Intel® oneAPI Data Analytics Library Developer Guide and Reference

Contents

Chapter 1: Intel® oneAPI Data Analytics Library (oneDAL)

Intel® oneAPI Data Analytics Library (oneDAL) 1

Intel® oneAPI Data Analytics Library (oneDAL) is a library that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation. The library provides two different sets of C++ interfaces: oneAPI and DAAL.

For general information, refer to Intel® oneAPI Data Analytics Library official page.

oneAPI vs. DAAL Interfaces

- oneAPI Interfaces are based on open oneDAL specification and are currently under an active development. They work on various hardware but only a limited set of algorithms is available at the moment.
- DAAL Interfaces are CPU-only interfaces that provide implementations for a wide range of algorithms.

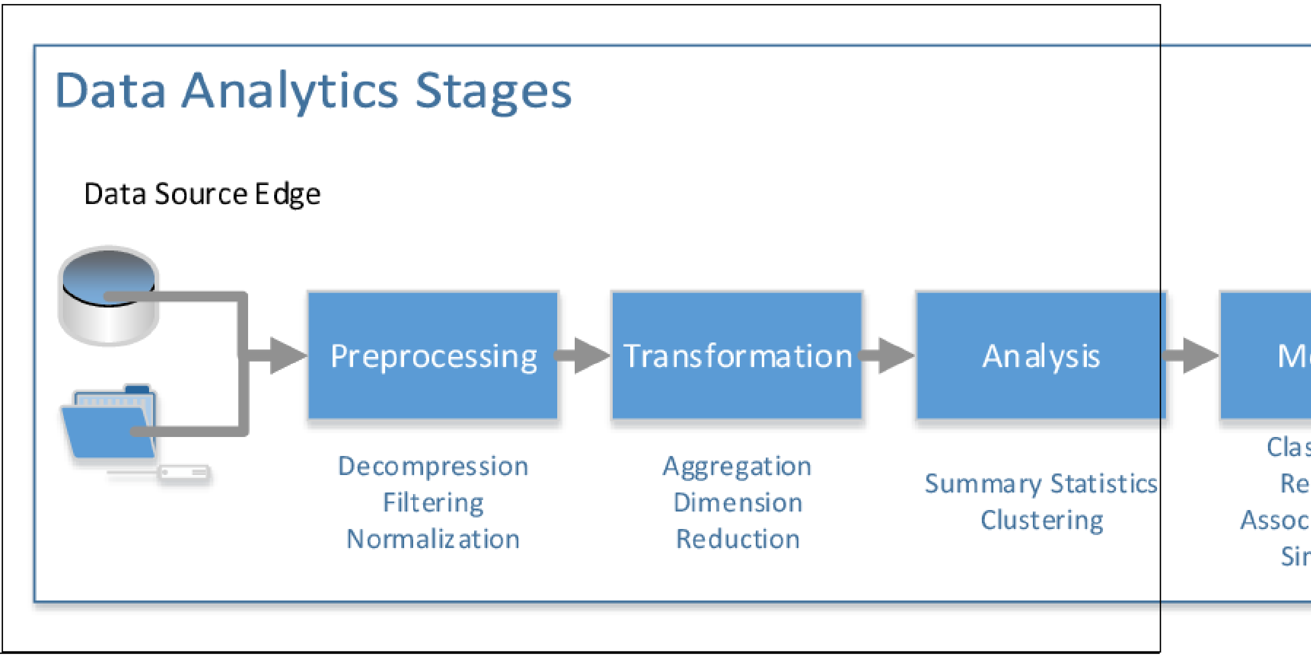
Developer Guide

API Reference

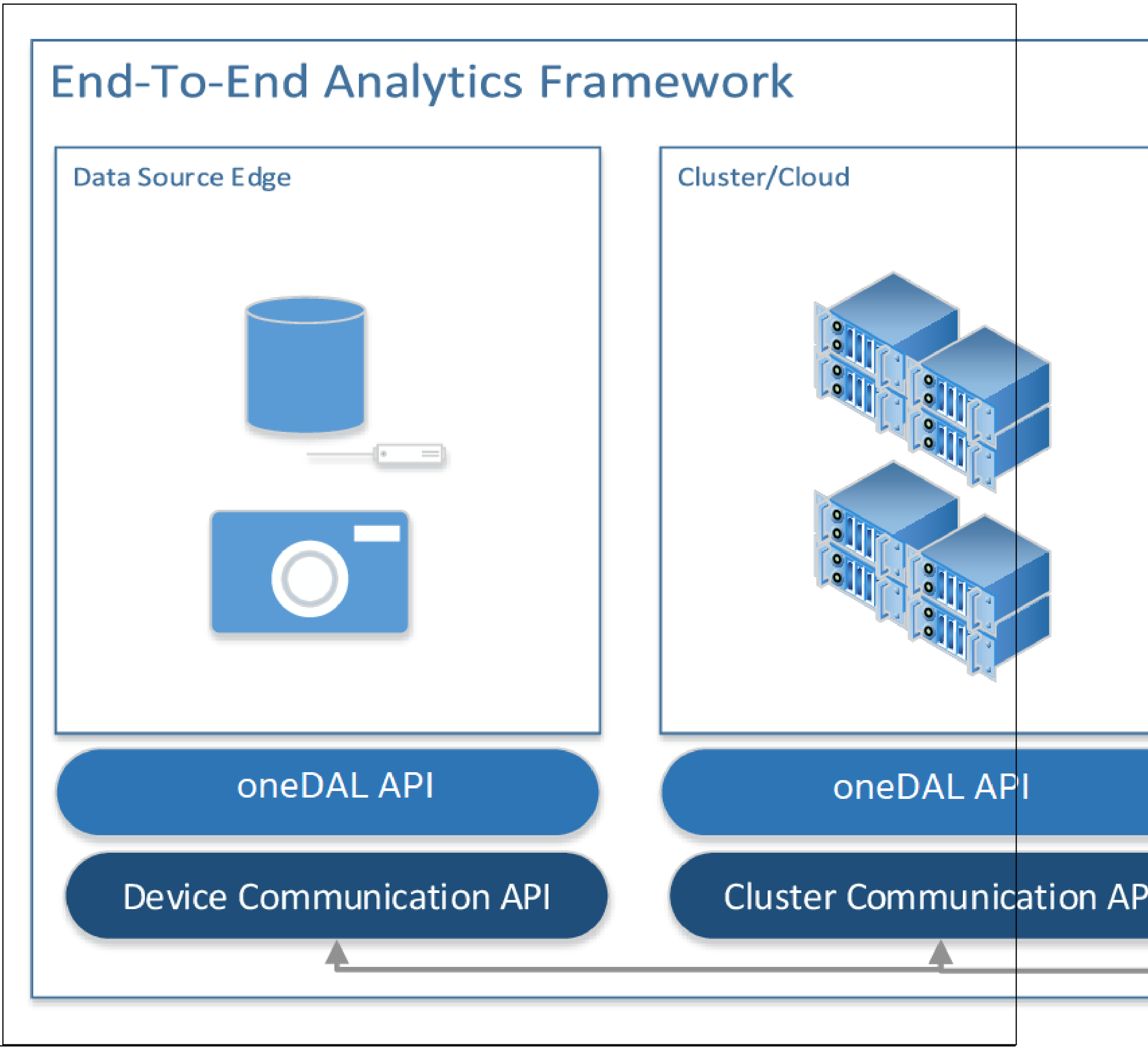
Introduction

Data Analytics Pipeline

Intel® oneAPI Data Analytics Library (oneDAL) is a library that provides building blocks covering all stages of data analytics: data acquisition from a data source, preprocessing, transformation, data mining, modeling, validation, and decision making.



oneDAL supports the concept of the end-to-end analytics when some of data analytics stages are performed on the edge devices (close to where the data is generated and where it is finally consumed). Specifically, oneDAL Application Programming Interfaces (APIs) are agnostic about a particular cross-device communication technology and, therefore, can be used within different end-to-end analytics frameworks.



Installation

You can obtain the latest version of oneDAL from oneDAL home page as a part of Intel® oneAPI Base Toolkit.

System Requirements

Refer to sysem requirements page.

oneAPI Interfaces

Introduction

- Build applications with oneDAL
- Glossary
- Mathematical Notations

Build applications with oneDAL

This section contains instructions for building applications with oneDAL for SYCL*.

- Applications on Linux* OS
- Applications on Windows* OS

Applications on Linux* OS

1. Install oneDAL.
2. Set environment variables by calling `<install dir>/setvars.sh`.
3. Build the application using `icpx` (Linux* OS) and `icx-cl` (Windows* OS) commands:
   - Add oneDAL `includes` folder:

```
-I<install dir>/dal/latest/include
```

- Add oneDAL libraries. Choose the appropriate oneDAL libraries based on oneDAL threading mode and linking method:

oneDAL libraries for Linux

	Single-threaded (non-threaded)	**Multi-threaded (internally threaded)**
Static linking	libonedal_core.a, libonedal_dpc.a,	libonedal_core.a, libonedal_dpc.a, libonedal_thread.a
Dynamic linking	libonedal_core.so, libonedal_dpc.so,	libonedal_core.so, libonedal_dpc.so, libonedal_thread.so

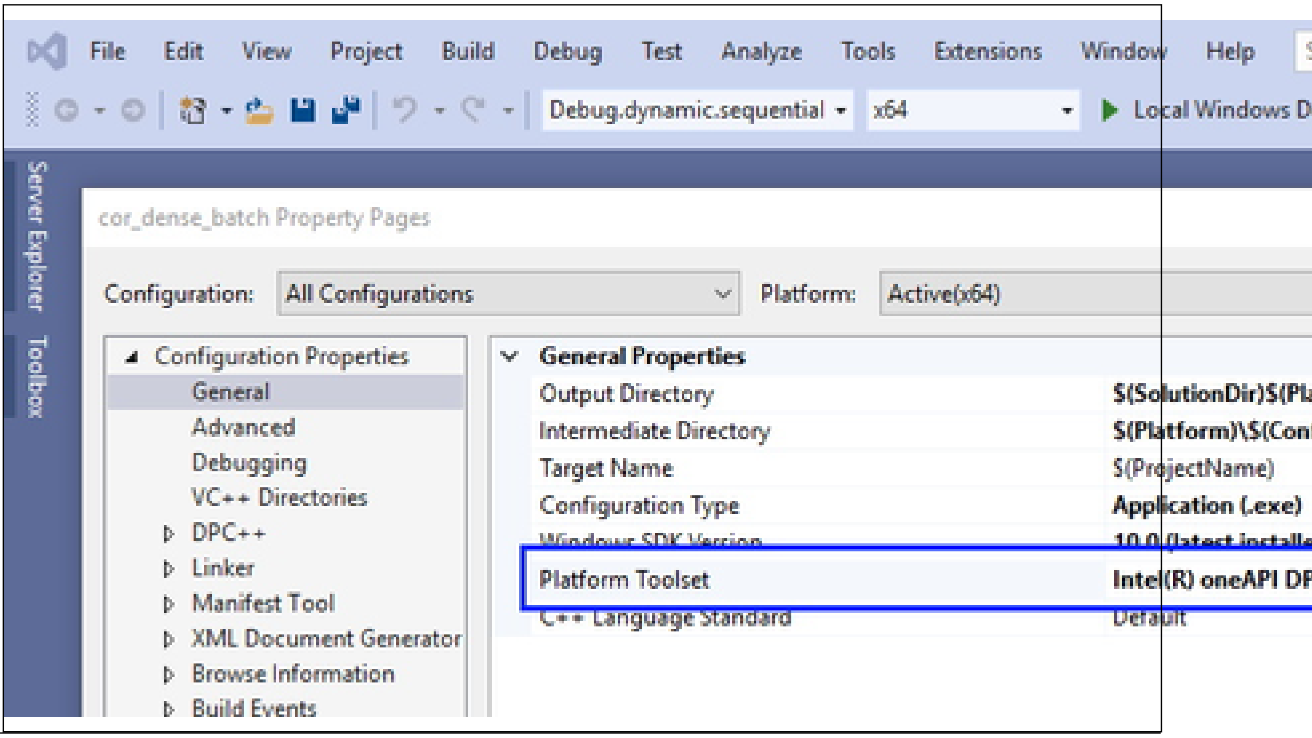
- Add an additional oneDAL library:

```
<install dir>/dal/latest/libintel64/libonedal_sycl.a
```

Applications on Windows* OS

1. Install oneDAL.
2. In Microsoft Visual Studio* Integrated Development Environment (IDE), open or create a C++ project for your oneDAL application to build.
3. In project properties:

- Set Intel® oneAPI DPC++/C++ Compiler platform toolset:

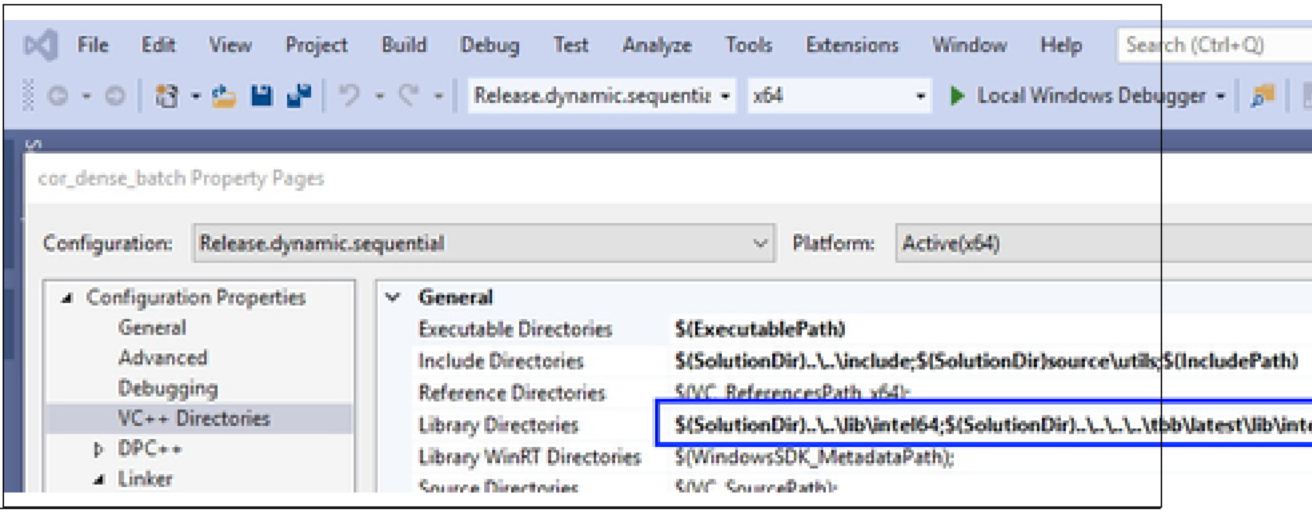


Configuration: All Configurations
Platform: Active(x64)

General Properties

Output Directory	$(SolutionDir)$(Pl
Intermediate Directory	$(Platform)\$(Con
Target Name	$(ProjectName)
Configuration Type	Application (.exe)
Windows SDK Version	10.0 (latest installe
Platform Toolset	Intel(R) oneAPI DP
C++ Language Standard	Default

- Add oneDAL `includes` folder to Additional Include Directories.
- Add folders with oneDAL and oneTBB libraries to Library Directories:



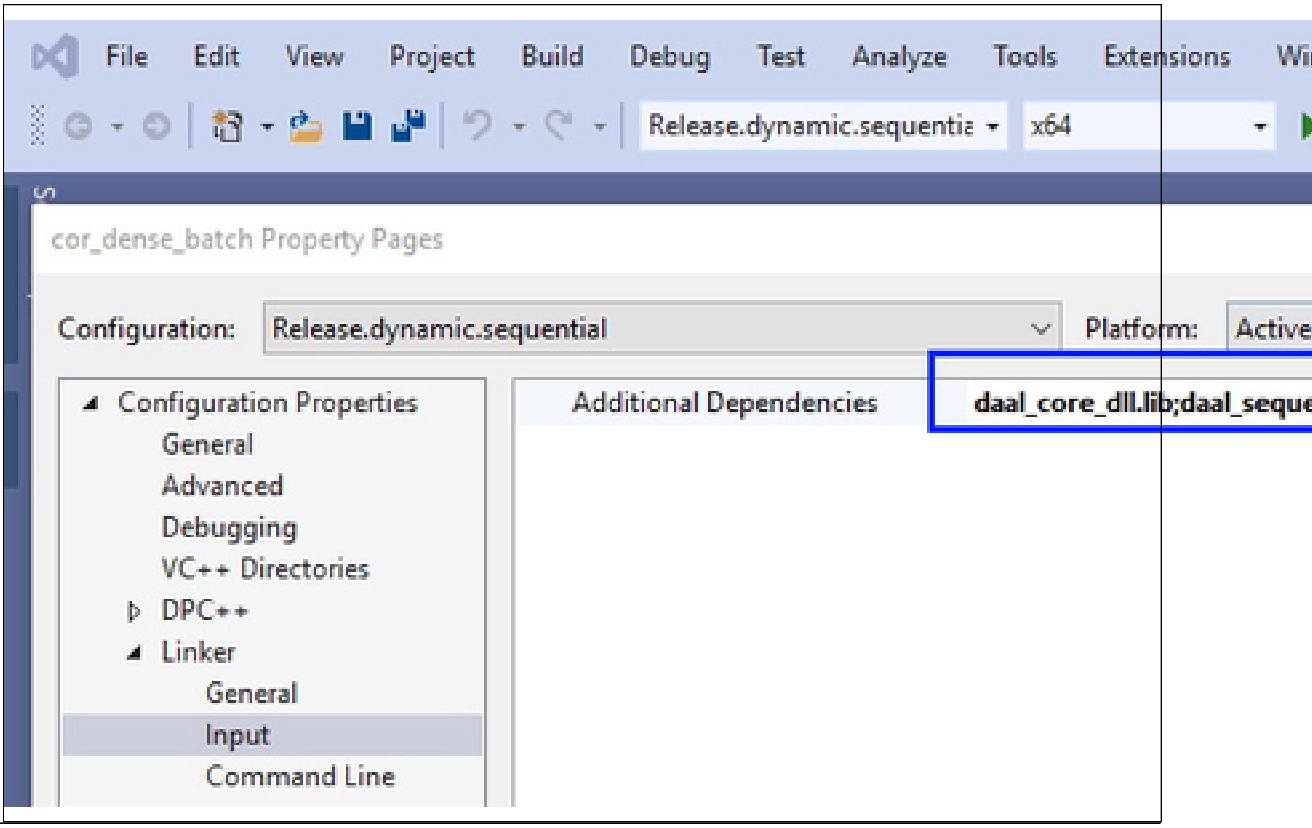
Configuration: Release.dynamic.sequential

Platform: Active(x64)

General

Executable Directories	$(ExecutablePath)
Include Directories	$(SolutionDir)..\..\include;$(SolutionDir)source\utils;$(IncludePath)
Reference Directories	$(VC_ReferencesPath_x64);
Library Directories	$(SolutionDir)..\..\lib\intel64;$(SolutionDir)..\..\..\..\tbb\latest\lib\inte
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);

- Add oneDAL and OpenCL libraries to Additional Dependencies:



Configuration: Release.dynamic.sequential

Platform: Active

Additional Dependencies: daal_core_dll.lib;daal_seque

4. Add the appropriate libraries to your project based on oneDAL threading mode and linking method:

oneDAL libraries for Windows

	Single-threaded (non-threaded)	Multi-threaded (internally threaded)
Static linking	onedal_core.lib,	onedal_core.lib, onedal_thread.lib
Dynamic linking	onedal_core_dll.lib	onedal_core_dll.lib

You may also add debug versions of the libraries based on the threading mode and linking method:

oneDAL debug libraries for Windows

	Single-threaded (non-threaded)	Multi-threaded (internally threaded)
Static linking	onedal_cored.lib, onedald.lib, onedal_dpcd.lib, onedal_sycld.lib,	onedal_cored.lib, onedald.lib, onedal_dpcd.lib, onedal_sycld.lib, onedal_threadd.lib
Dynamic linking	onedal_cored_dll.lib (onedal_cored_dll.1.lib), onedald_dll.lib (onedald_dll.1.lib), onedal_dpcd_dll.lib (onedal_dpcd_dll.1.lib), onedald.1.dll, onedal_cored.1.dll, onedal_dpcd.1.dll,	onedal_cored_dll.lib (onedal_cored_dll.1.lib), onedald_dll.lib (onedald_dll.1.lib), onedal_dpcd_dll.lib (onedal_dpcd_dll.1.lib), onedald.1.dll, onedal_cored.1.dll, onedal_dpcd.1.dll, onedal_threadd.1.dll

Examples

Dynamic linking, Multi-threaded oneDAL:

- Linux* OS:

```
icpx -fsycl my_first_dal_program.cpp -Wl,
--start-group -L<install dir>/dal/latest/lib/intel64 -lonedal_core -
lonedal_dpc -lonedal_thread -lpthread -ldl -lOpenCL -L<install dir>/tbb/
latest/lib/intel64/gcc4.8 -ltbb -ltbbmalloc <install dir>/dal/latest/lib/
intel64/libonedal_sycl.a -Wl,--end-group
```

- Windows* OS:

```
icx-cl -fsycl my_first_dal_program.cpp -Wl,
--start-group -L<install dir>/dal/latest/lib/intel64 -lonedal_core -
lonedal_dpc -lonedal_thread -lpthread -ldl -lOpenCL -L<install dir>/tbb/
latest/lib/intel64/gcc4.8 -ltbb -ltbbmalloc <install dir>/dal/latest/lib/
intel64/libonedal_sycl.a -Wl,--end-group
```

Static linking, Single-threaded oneDAL:

- Linux* OS:

```
icpx -fsycl my_first_dal_program.cpp -Wl,
--start-group <install dir>/dal/latest/lib/intel64/libonedal_core.a <install
dir>/dal/latest/lib/intel64/libonedal_dpc.a -lpthread -ldl -lOpenCL <install
dir>/dal/latest/lib/intel64/libonedal_sycl.a -Wl,--end-group
```

- Windows* OS:

```
icx-cl -fsycl my_first_dal_program.cpp -Wl,
--start-group <install dir>/dal/latest/lib/intel64/libonedal_core.a <install
dir>/dal/latest/lib/intel64/libonedal_dpc.a -lpthread -ldl -lOpenCL <install
dir>/dal/latest/lib/intel64/libonedal_sycl.a -Wl,--end-group
```

Glossary

Machine learning terms

Categorical feature

A feature with a discrete domain. Can be nominal or ordinal.

Synonyms: discrete feature, qualitative feature

Classification

A supervised machine learning problem of assigning labels to feature vectors.

Examples: predict what type of object is on the picture (a dog or a cat?), predict whether or not an email is spam

Clustering

An unsupervised machine learning problem of grouping feature vectors into bunches, which are usually encoded as nominal values.

Example: find big star clusters in the space images

Continuous feature

A feature with values in a domain of real numbers. Can be interval or ratio

Synonyms: quantitative feature, numerical feature

Examples: a person's height, the price of the house

CSV file

A comma-separated values file (csv) is a type of a text file. Each line in a CSV file is a record containing fields that are separated by the delimiter. Fields can be of a numerical or a text format. Text usually refers to categorical values. By default, the delimiter is a comma, but, generally, it can be any character. For more details, see.

Dimensionality reduction

A problem of transforming a set of feature vectors from a high-dimensional space into a low-dimensional space while retaining meaningful properties of the original feature vectors.

Feature

A particular property or quality of a real object or an event. Has a defined type and domain. In machine learning problems, features are considered as input variable that are independent from each other.

Synonyms: attribute, variable, input variable

Feature vector

A vector that encodes information about real object, an event or a group of objects or events. Contains at least one feature.

Example: A rectangle can be described by two features: its width and height

Inference

A process of applying a trainedmodel to the dataset in order to predict response values based on input feature vectors.

Synonym: prediction

Inference set

A dataset used at the inference stage. Usually without responses.

Interval feature

A continuous feature with values that can be compared, added or subtracted, but cannot be multiplied or divided.

Examples: a time frame scale, a temperature in Celsius or Fahrenheit

Label

A response with categorical or ordinal values. This is an output in classification and clustering problems.

Example: the spam-detection problem has a binary label indicating whether the email is spam or not

Model

An entity that stores information necessary to run inference on a new dataset. Typically a result of a training process.

Example: in linear regression algorithm, the model contains weight values for each input feature and a single bias value

Nominal feature

A categorical feature without ordering between values. Only equality operation is defined for nominal features.

Examples: a person’s gender, color of a car

Nu-classification

An SVM-specific classification problem where ν parameter is used instead of C. ν is an upper bound on the fraction of training errors and a lower bound of the fraction of the support vector.

Nu-regression

An SVM-specific regression problem where ν parameter is used instead of ϵ. ν is an upper bound on the fraction of training errors and a lower bound of the fraction of the support vector.

Observation

A feature vector and zero or more responses.

Synonyms: instance, sample

Ordinal feature

A categorical feature with defined operations of equality and ordering between values.

Example: student’s grade

Outlier

Observation which is significantly different from the other observations.

Ratio feature

A continuous feature with defined operations of equality, comparison, addition, subtraction, multiplication, and division. Zero value element means the absence of any value.

Example: the height of a tower

Regression

A supervised machine learning problem of assigning continuousresponses for feature vectors.

Example: predict temperature based on weather conditions

Response

A property of some real object or event which dependency from feature vector need to be defined in supervised learning problem. While a feature is an input in the machine learning problem, the response is one of the outputs can be made by the model on the inference stage.

Synonym: dependent variable

Result options:

Result options are entities that mimic C++ enums. They are used to specify which results of an algorithm should be computed. The use of result options may alter the default algorithm flow and result in performance differences. In general, fewer results to compute means faster performance. An error is thrown when you use an invalid set of result options or try to access the results that are not yet computed.

Example: k-NN Classification algorithm can perform classification and also return indices and distances to the nearest observations as a result option.

Search

A kNN-specific optimization problem of finding the point in a given set that is the closest to the given points.

Supervised learning

Training process that uses a dataset with information about dependencies between features and responses. The goal is to get a model of dependencies between input feature vector and responses.

Training

A process of creating a model based on information extracted from a training set. Resulting model is selected in accordance with some quality criteria.

Training set

A dataset used at the training stage to create a model.

Unsupervised learning

Training process that uses a training set with no responses. The goal is to find hidden patters inside feature vectors and dependencies between them.

Graph analytics terms

Adjacency

A vertex *u* is adjacent to vertex *v* if they are joined by an edge.

Adjacency matrix

An $nimesn$ matrix A_G for a graph *G* whose vertices are explicitly ordered $(v_1, v_2, ..., v_n)$,

$$\mathrm{A_G} = \begin{cases} 1, \text{where } v_i \text{ and } v_j \text{ adjacent} \\ 0, \text{otherwise.} \end{cases}$$

Attribute

A value assigned to graph, vertex or edge. Can be numerical (weight), string or any other custom data type.

Component

A connectedsubgraph*H* of graph *G* such that no subgraph of *G* that properly contains *H* is connected [Gross2014].

Connected graph

A graph is connected if there is a walk between every pair of its vertices [Gross2014].

Directed graph

A graph where each edge is an ordered pair (u, v) of vertices. *v* is designated as the tail, and *u* is designated as the head.

Edge index

The index *i* of an edge e_i in an edge set $E = \{e_1, e_2, ..., e_m\}$ of graph*G*. Can be an integer value.

Graph

An object $G = (V; E)$ that consists of two sets, *V* and *E*, where *V* is a finite nonempty set, *E* is a finite set that may be empty, and the elements of *E* are two-element subsets of *V*. *V* is called a set of vertices, *E* is called a set of edges [Gross2014].

Induced subgraph on the edge set

Each subset $E' \subseteq E$ defines a unique subgraph$H' = (V'; E')$ of graph $G = (V; E)$, where V' consists of only those vertices that are the endpoints of the edges in E'. The subgraph *H* is called an induced subgraph of *G* on the edge set E'[Gross2014].

Induced subgraph on the vertex set

Each subset $V' \subseteq V$ defines a unique subgraph$H = (V'; E')$ of graph $G = (V; E)$, where E' consists of those edges whose endpoints are in V'. The subgraph *H* is called an induced subgraph of *G* on the vertex set V' [Gross2014].

Self-loop

An edge that joins a vertex to itself.

Subgraph

A graph $H = (V'; E')$ is called a subgraph of graph $G = (V; E)$ if $V' \subseteq V; E' \subseteq E$ and V' contains all the endpoints of all the edges in E' [Gross2014].

Topology

A graph without attributes.

Undirected graph

A graph where each edge is an unordered pair (u, v) of vertices.

Unweighted graph

A graph where all vertices and all edges has no weights.

Vertex index

The index *i* of a vertex v_i in a vertex set $V = \{v_1, v_2, ..., v_n\}$ of graph*G*. Can be an integer value.

Walk

An alternating sequence of vertices and edges such that for each edge, one endpoint precedes and the other succeeds that edge in the sequence [Gross2014].

Weight

A numerical value assigned to vertex, edge or graph.

Weighted graph

A graph where all vertices or all edges have weights

oneDAL terms

Accessor

A oneDAL concept for an object that provides access to the data of another object in the special data format. It abstracts data access from interface of an object and provides uniform access to the data stored in objects of different types.

Batch mode

The computation mode for an algorithm in oneDAL, where all the data needed for computation is available at the start and fits the memory of the device on which the computations are performed.

Builder

A oneDAL concept for an object that encapsulates the creation process of another object and enables its iterative creation.

Contiguous data

Data that are stored as one contiguous memory block. One of the characteristics of a data format.

CSR data

A compressed sparse row (csr) data is the sparse matrix representation. Data with values of a single data type and the same set of available operations defined on them. One of the characteristics of a data format.

Data format

Representation of the internal structure of the data.

Examples: data can be stored in array-of-structures or compressed-sparse-row format

Data layout

A characteristic of data format which describes the order of elements in a contiguous data block.

Example: row-major format, where elements are stored row by row

Data type

An attribute of data used by a compiler to store and access them. Includes size in bytes, encoding principles, and available operations (in terms of a programming language).

Examples:`int32_t`, `float`, `double`

Dataset

A collection of data in a specific data format.

Examples: a collection of observations, a graph

Flat data

A block of contiguoushomogeneous data.

Getter

A method that returns the value of the private member variable.

Example:

```
std::int64_t get_row_count() const;
```

Heterogeneous data

Data which contain values either of different data types or different sets of operations defined on them. One of the characteristics of a data format.

Example: A dataset with 100 observations of three interval features. The first two features are of float32 data type, while the third one is of float64 data type.

Homogeneous data

Data with values of single data type and the same set of available operations defined on them. One of the characteristics of a data format.

Example: A dataset with 100 observations of three interval features, each of type float32

Immutability

The object is immutable if it is not possible to change its state after creation.

Metadata

Information about logical and physical structure of an object. All possible combinations of metadata values present the full set of possible objects of a given type. Metadata do not expose information that is not a part of a type definition, e.g. implementation details.

Example:table object can contain three nominal features with 100 observations (logical part of metadata). This object can store data as sparse csr array and provides direct access to them (physical part)

Online mode

The computation mode for an algorithm in oneDAL, where the data needed for computation becomes available in parts over time.

Reference-counted object

A copy-constructible and copy-assignable oneDAL object which stores the number of references to the unique implementation. Both copy operations defined for this object are lightweight, which means that each time a new object is created, only the number of references is increased. An implementation is automatically freed when the number of references becomes equal to zero.

Setter

A method that accepts the only parameter and assigns its value to the private member variable.

Example:

```
void set_row_count(std::int64_t row_count);
```

Table

A oneDAL concept for a dataset that contains only numerical data, categorical or continuous. Serves as a transfer of data between user's application and computations inside oneDAL. Hides details of data format and generalizes access to the data.

Workload

A problem of applying a oneDAL algorithm to a dataset.

Common oneAPI terms

API

Application Programming Interface

DPC++

Data Parallel C++ (DPC++) is a high-level language designed for data parallel programming productivity. DPC++ is based on SYCL* from the Khronos* Group to support data parallelism and heterogeneous programming.

Host/Device

OpenCL [OpenCLSpec] refers to CPU that controls the connected GPU executing kernels.

JIT

Just in Time Compilation — compilation during execution of a program.

Kernel

Code written in OpenCL [OpenCLSpec] or SYCL and executed on a GPU device.

SPIR-V

Standard Portable Intermediate Representation - V is a language for intermediate representation of compute kernels.

SYCL

SYCL(TM) [SYCLSpec] — high-level programming model for OpenCL(TM) that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++.

Distributed computational mode terms

Communicator

A oneDAL concept for an object that is used to perform inter-process collective operations

Communicator backend

A particular library providing collective operations.

Examples: oneCCL, oneMPI

SPMD

Single Program, Multiple Data (SPMD) is a technique employed to achieve parallelism. In SPMD model, multiple autonomous processors simultaneously execute the same program at independent points.

Mathematical Notations

Notation	Definition
n or m	The number of observations in a tabular dataset. Typically n is used, but sometimes m is required to distinguish two datasets, e.g., the training set and the inference set.
p or r	The number of features in a tabular dataset. Typically p is used, but sometimes r is required to distinguish two datasets.
$a \times b$	The dimensionality of a matrix (dataset) has a rows (observations) and b columns (features).
V	The vertex set in a graph.
E	The edge set in a graph.
u, v or w	The vertex in a graph.
(u, v)	The edge in a graph.
$\|A\|$	Depending on the context may be interpreted as follows:

Notation	Definition
	• If *A* is a set, this denotes its cardinality, i.e., the number of elements in the set *A*. • If *A* is a real number, this denotes the absolute value of *A*.
$\|\|x\|\|$	The L_2-norm of a vector $x \in \mathbb{R}^d$, $$\|\|x\|\| = \sqrt{x_1^2 + x_2^2 + \dots + x_d^2}.$$
$\operatorname{sgn}(x)$	Sign function for $x \in \mathbb{R}$, $$\operatorname{sgn}(x) = \begin{cases} -1, x < 0, \\ 0, x = 0, \\ 1, x > 0. \end{cases}$$
x_i	In the description of an algorithm, this typically denotes the *i*-th feature vector in the training set.
x'_i	In the description of an algorithm, this typically denotes the *i*-th feature vector in the inference set.
y_i	In the description of an algorithm, this typically denotes the *i*-th response in the training set.
y'_i	In the description of an algorithm, this typically denotes the *i*-th response that needs to be predicted by the inference algorithm given the feature vector x'_i from the inference set.

Computational Modes

Batch

In the batch processing mode, the algorithm works with the entire data set to produce the final result. A more complex scenario occurs when the entire data set is not available at the moment or the data set does not fit into the device memory.

Online

In the online processing mode, the algorithm processes a data set in blocks streamed into the device's memory. Partial results are updated incrementally and finalized when the last data block is processed.

Distributed

In the distributed processing mode, the algorithm operates on a data set distributed across several devices (compute nodes). On each node, the algorithm produces partial results that are later merged into the final result on the main node.

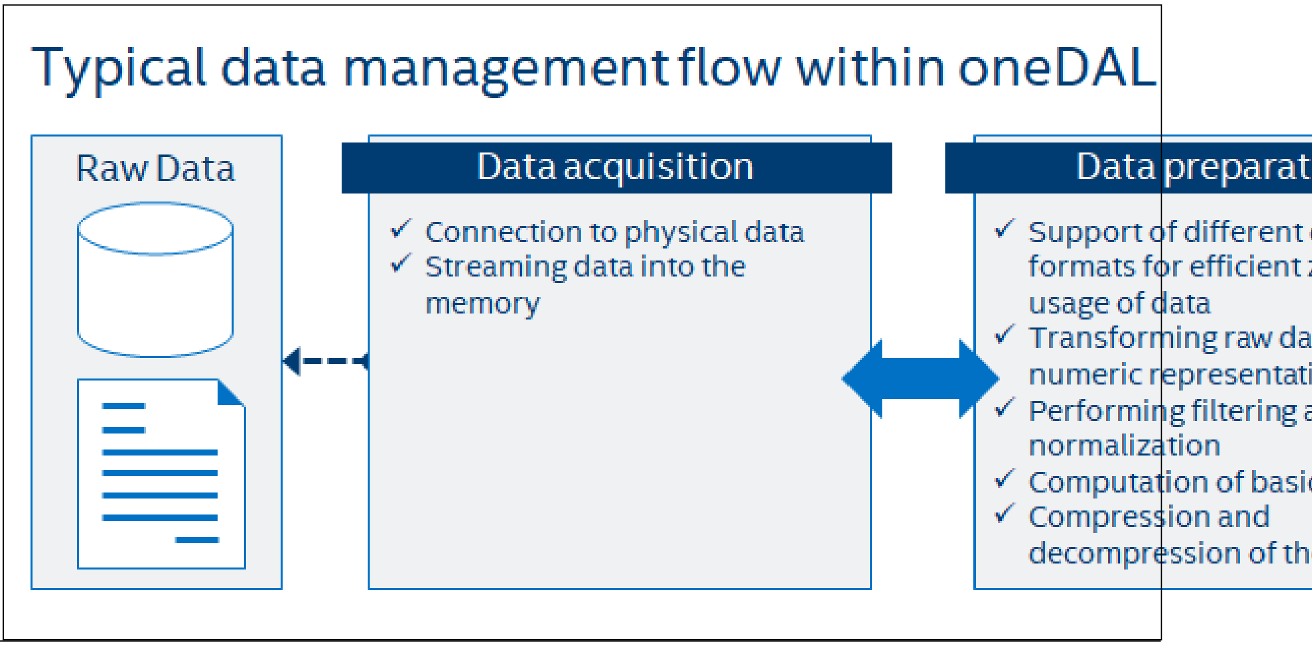
Data Management

This section includes concepts and objects that operate on data. For oneDAL, such set of operations, or **data management**, is distributed between different stages of the data analytics pipeline. From a perspective of data management, this pipeline contains three main steps of data acquisition, preparation, and computation (see the picture below):

1. Raw data acquisition
   - Transfer out-of-memory data from various sources (databases, files, remote storage) into an in-memory representation.
2. Data preparation
   - Support different in-memory data formats.
   - Compress and decompress the data.
   - Convert the data into numeric representation.
   - Recover missing values.
   - Filter the data and perform data normalization.
   - Compute various statistical metrics for numerical data, such as mean, variance, and covariance.
3. Algorithm computation
   - Stream in-memory numerical data to the algorithm.

In complex usage scenarios, data flow goes through these three stages back and forth. For example, when the data are not fully available at the start of the computation, it can be done step-by-step using blocks of data. After the computation on the current block is completed, the next block should be obtained and prepared.

Data Management Flow in oneDAL

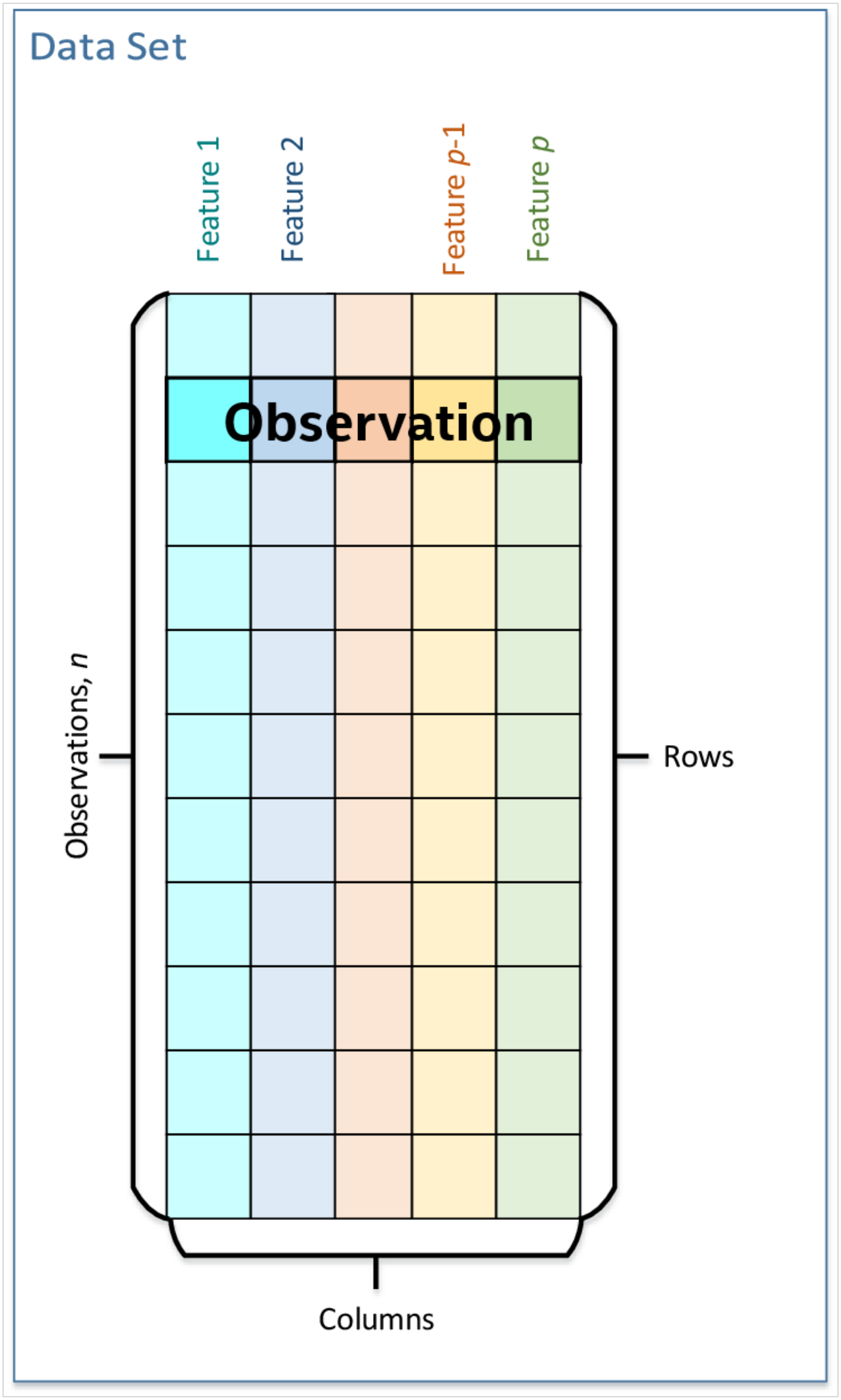


Key concepts

oneDAL provides a set of concepts to operate on out-of-memory and in-memory data during different stages of the data analytics pipeline.

Dataset

The main data-related concept that oneDAL works with is a dataset. It is a collection of data in a specific data format.


Data Set
Feature 1
Feature 2
Feature p-1
Feature p
Observation
Observations, n
Rows
Columns

The dataset is used across all stages of the data analytics pipeline. For example:

1. At the acquisition stage, it is downloaded into the local memory.
2. At the preparation stage, it is converted into a numerical representation.
3. At the computation stage, it is used as one of the inputs or results of an algorithm or a descriptor properties.

Data source

Data source is a concept of an out-of-memory storage for a dataset. It is used at the data acquisition and data preparation stages to:

- Extract datasets from external sources such as databases, files, remote storage.
- Load datasets into the device's local memory. Data do not always fit the local memory, especially when processing with accelerators. A data source provides the ability to load data by batches and extracts it directly into the device's local memory. Therefore, a data source enables complex data analytics scenarios, such as online computations.
- Transform datasets into their numerical representation. Data source automatically transforms non-numeric categorical and continuous data values into one of the numeric data formats.

For details, see dm data sources section.

Table

Table is a concept of in-memory numerical data that are organized in a tabular view with several rows and columns. It is used at the data preparation and data processing stages to:

- Be an in-memory representation of a dataset or another tabular data (for example, matrices, vectors, and scalars).
- Store heterogeneous data in various data formats, such as dense, sparse, chunked, contiguous.
- Avoid unnecessary data copies during conversion from external data representations.
- Transfer memory ownership of the data from user application to the table, or share it between them.
- Connect with the data source to convert data from an out-of-memory into an in-memory representation.
- Support streaming of the data to the algorithm.
- Access the underlying data on a device in a required data format, e.g. by blocks with the defined data layout.

> **NOTE** For thread-safety reasons and better integration with external entities, a table provides a read-only access to the data within it, thus, table object is immutable.

This concept has different logical organization and physical format of the data:

- Logically, a table contains *n* rows and *p* columns. Every column may have its own type of data values and a set of allowed operations.
- Physically, a table can be organized in different ways: as a homogeneous, contiguous array of bytes, as a heterogeneous list of arrays of different data types, in a compressed-sparse-row format. The number of bytes needed to store the data differs from the number of elements $nimesp$ within a table.

For details, see dm tables section.

Table metadata

Table metadata concept provides an additional information about data in the table:

1. The data types of the columns.
2. The logical types of data in the columns: nominal, ordinal, interval, or ratio.

Only the properties of data that do not affect table concept definition is a part of metadata concept.

Accessor

Accessor is a concept that defines a single way to extract the data from a table. It allows to:

- Have unified access to the data from table objects of different types, without exposing their implementation details.
- Provide a flat view on the data blocks of a table for better data locality. For example, the accessor returns a column of the table stored in row-major format as a contiguous array.
- Acquire data in a desired data format for which a specific set of operations is defined.
- Have read-only access to the data.

For details, see dm accessors section.

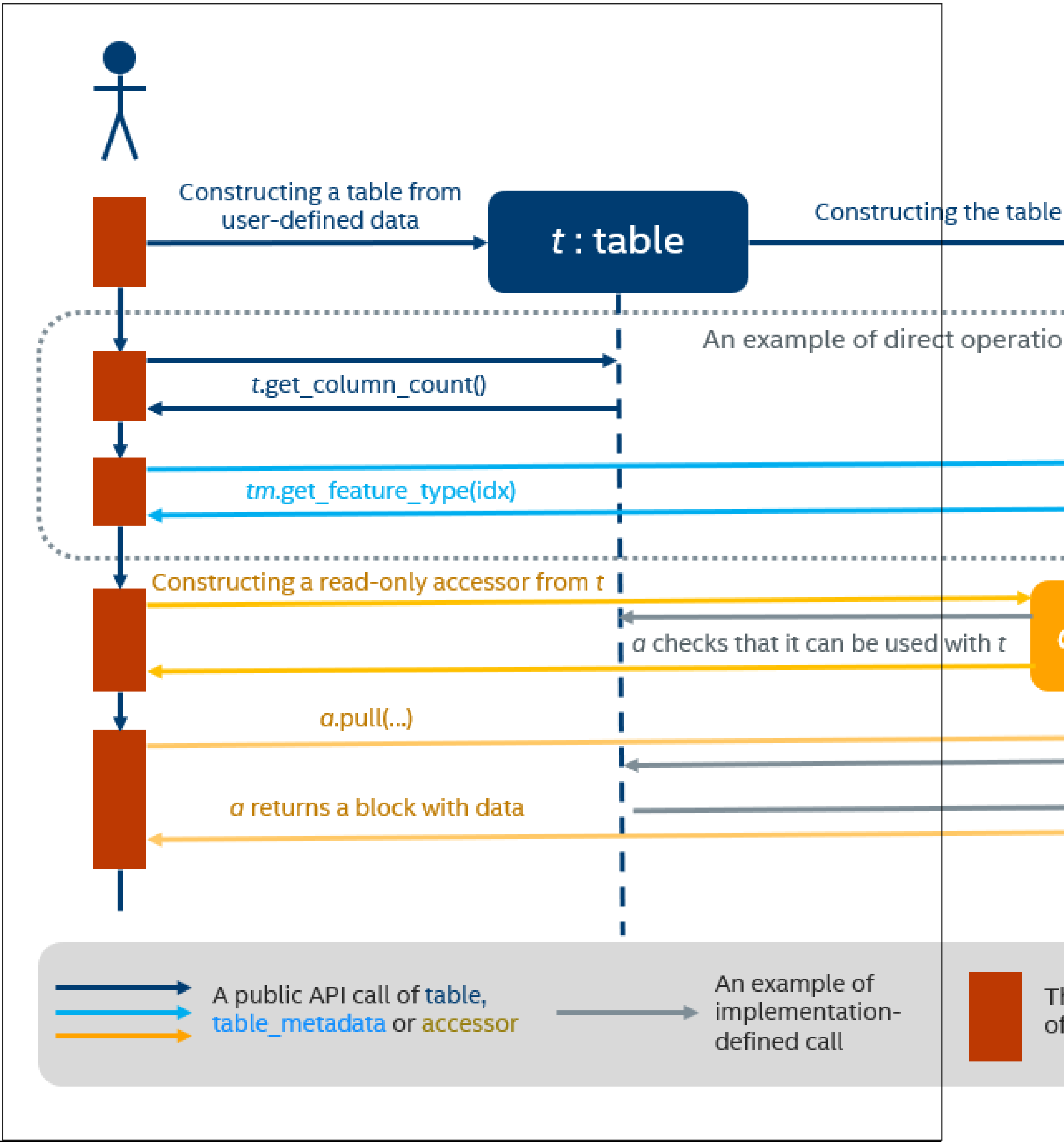
Example of interaction between table and accessor objects

This section provides a basic usage scenario of the table and accessor concepts and demonstrates the relations between them. The following diagram shows objects of these concepts, which are highlighted by colors:

- table object is dark blue
- accessor is orange

- table metadata is light blue

Sequence diagram of accessor-builder-table relations


Constructing a table from user-defined data
t : table
Constructing the table
An example of direct operatio
t.get_column_count()
tm.get_feature_type(idx)
Constructing a read-only accessor from t
a checks that it can be used with t
a.pull(...)
a returns a block with data
A public API call of table, table_metadata or accessor
An example of implementation-defined call
Th
of

To perform computations on a dataset, you have to create a table object first. It can be done either using a data source or directly from user-defined memory. The diagram shows the creation of a table object **t** from the data provided by user (not shown on the diagram). During a table creation, an object **tm** of table metadata is constructed and initialized using the data.

Once a table object is created, it can be used as an input in computations or as a parameter of some algorithm. The data in the table can be accessed via its own interface or via read-only accessor as shown on the diagram.

Graph

A graph is a concept of in-memory structured data that is organized as a graph with several vertices and edges. Graphs can be directed, undirected, weighted and attributed. Graphs are used at the data preparation and data processing stages to:

- Be an in-memory representation of a dataset.
- Store graph data in sparse data formats.
- Avoid unnecessary data copies during conversion from external data representations.
- Connect with the data source to convert data from an out-of-memory representation into an in-memory representation.

> **NOTE** For thread-safety reasons and better integration with external entities, a graph provides a read-only access to the data within it, thus, a graph object is immutable.

The logical organization of a graph and the physical format of the data are different:

- Logically, a graph contains $|V|$ vertices and $|E|$ edges. All vertices of the graph are described with the same data type and respective operations on it. Similarly, the same is true for edges and attributes of the graph. The data types of vertices, edges, and attributes can be different.
- Physically, the topology of a graph can be organized in CSR and others data formats.

For details, see dm graphs section.

Details

This section includes the detailed descriptions of all data management objects in oneDAL.

Array

The array is a simple concept over the data in oneDAL. It represents a storage that:

1. Holds the data allocated inside it or references to the external data. The data are organized as one homogeneous and contiguous memory block.
2. Contains information about the memory block’s size.
3. Supports both immutable and mutable data.
4. Provides an ability to change the data state from immutable to mutable one.
5. Holds ownership information on the data (see the data ownership requirements section).
6. Ownership information on the data can be shared between several arrays. It is possible to create a new array from another one without any data copies.

Usage example

The following listing provides a brief introduction to the array API and an example of basic usage scenario:

```
#include <sycl/sycl.hpp>
#include <iostream>
#include <string>
#include "oneapi/dal/array.hpp"

using namespace oneapi;

void print_property(const std::string& description, const auto& property) {
   std::cout << description << ": " << property << std::endl;
}

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr std::int64_t data_count = 4;
   const float data[] = { 1.0f, 2.0f, 3.0f, 4.0f };

   // Creating an array from immutable user-defined memory
   auto arr_data = dal::array<float>::wrap(data, data_count);

   // Creating an array from internally allocated memory filled by ones
   auto arr_ones = dal::array<float>::full(queue, data_count, 1.0f);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_ones mutable", arr_ones.has_mutable_data()); // true

   // Creating new array from arr_data without data copy - they share ownership information.
   dal::array<float> arr_mdata = arr_data;

   print_property("arr_mdata elements count", arr_mdata.get_count()); // equal to data_count
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // false
   /// Copying data inside arr_mdata to new mutable memory block.
   /// arr_data still refers to the original data pointer.
   arr_mdata.need_mutable_data(queue);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // true

   queue.submit([&](sycl::handler& cgh){
      auto mdata = arr_mdata.get_mutable_data();
      auto cones = arr_ones.get_data();
      cgh.parallel_for<class array_addition>(sycl::range<1>(data_count), [=](sycl::id<1> idx) {
         mdata[idx[0]] += cones[idx[0]];
      });
   }).wait();

   std::cout << "arr_mdata values: ";
   for(std::int64_t i = 0; i < arr_mdata.get_count(); i++) {
      std::cout << arr_mdata[i] << ", ";
   }
   std::cout << std::endl;

   return 0;
}
```

Data ownership requirements

The array supports the following requirements on the internal data management:

1. An array owns two properties representing raw pointers to the data:
   - `data` for a pointer to immutable data block
   - `mutable_data` for a pointer to mutable data block (see the api array)
2. If an array owns mutable data, both properties point to the same memory block.
3. If an array owns immutable data, `mutable_data` is `nullptr`.
4. An array stores the number of elements in the block it owns and updates the `count` property when a new memory block is assigned to the array.
5. An array stores a pointer to the **ownership structure** of the data:
   - The **reference count** indicating how many array objects refer to the same memory block.
   - The **deleter** object used to free the memory block when reference count is zero.
6. An array creates the ownership structure for a new memory block not associated with such structure.
7. An array decrements the number of references to the memory block when the array goes out of the scope. If the number of references is zero, the array calls the deleter on this memory block and free the ownership structure.
8. An array stores the pointer to the ownership structure created by another array when they share the data. An array increments the reference count for it to be equal to the number of array objects sharing the same data.

Programming interface

Refer to API Reference: Array.

Accessors

This section defines requirements to an accessor implementation and introduces several accessor types.

Requirements

Each accessor implementation:

1. Defines a single format of the data for the access. Every accessor type returns and use only one data format.
2. Provides read-only access to the data in the table types.
3. Provides the `pull()` method for obtaining the values from the table.
4. Is lightweight. Its constructors do not have computationally intensive operations such data copy, reading, or conversion. These operations are performed by method `pull()`.
5. The `pull()` method avoids data copy and conversion when it is possible to return the pointer to the memory block in the table. This is applicable for cases such as when the data format and data types of the data within the table are the same as the data format and data type for the access.

Accessor Types

oneDAL defines a set of accessor classes. Each class supports one specific way of obtaining data from the table.

All accessor classes in oneDAL are listed below:

Accessor Types

Accessor type	Description	List of supported types
row accessor	Provides access to the range of rows as one contiguoushomogeneous block of memory.	homogen table
column accessor	Provides access to the range of values within a single column as one contiguoushomogeneous block of memory.	homogen table

Details

- Column accessor
  - Usage example
  - Programming interface
- Row accessor
  - Usage example
  - Programming interface

Column accessor

The `column_accessor` class provides a read-only access to the column values of the table as contiguoushomogeneous array.

Usage example

```
#include <sycl/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/column_accessor.hpp"

using namespace oneapi;
```

```
int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count * column_count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing whole elements in a first column
   dal::column_accessor<const float> acc { t };

   auto block = acc.pull(queue, 0);
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

Refer to API Reference: Column accessor.

Row accessor

The `row_accessor` class provides a read-only access to the rows of the table as contiguoushomogeneous array.

Usage example

```
#include <sycl/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/row_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };
   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count * column_count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing second and third rows of the table
   dal::row_accessor<const float> acc { t };

   auto block = acc.pull(queue, {1, 3});
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

Refer to API Reference: Row accessor.

Data Sources

This section describes the types related to the data source concept.

Read

Read operation is a function that transforms a data source and other arguments represented via an args object to a result object. The operation is responsible for:

- Executing all of the data retrieval and transformation routines of the data source.
- Passing a SYCL* queue to the data retrieval and transformation routines.

Read operation definition

The following code sample shows the declaration for a read operation.

```
namespace oneapi::dal {

template <typename Object, typename DataSource>
using read_args_t = /* implementation defined */;

template <typename Object, typename DataSource>
using read_result_t = Object;

template <typename Object, typename DataSource>
read_result_t<Object, DataSource> read(
   sycl::queue& queue,
   const DataSource& data_source,
   const read_args_t<Object, DataSource>& args);

} // namespace oneapi::dal
```

Each operation satisfies the following requirements:

- An operation accepts three parameters in the following order:
  - The SYCL* queue object.
  - The data source.
  - The args object.
- An operation returns the result object.
- The `read_args_t` and `read_result_t` alias templates is used for inference of the args and return types.

Read operation shortcuts

In order to make the code on user side less verbose, oneDAL defines the following overloaded functions called *shortcuts* for a read operation in addition to the general one described in section Read operation definition.

- A shortcut for execution on host. Performs the same operation as the general function on host, but does not require passing the queue explicitly.

```
template <typename Object, typename DataSource>
read_result_t<Object, DataSource> read(
   const DataSource& data_source,
   const read_args_t<Object, DataSource>& args);
```

- A shortcut that allows omitting explicit args creation.

```
template <typename Object, typename DataSource, typename... Args>
read_result_t<Object, DataSource> read(
   sycl::queue& queue,
   const DataSource& data_source,
   Args&&... args);
```

- A shortcut that allows omitting explicit queue and args creation. This is a combination of two previous shortcuts.

```
template <typename Object, typename DataSource, typename... Args>
read_result_t<Object, DataSource> read(
   const DataSource& data_source,
   Args&&... args);
```

Args

- The string `%DATA_SOURCE%` should be substituted with the name of the data source, for example, `csv`.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` should be substituted with the name and the type of one of the data source args properties.

```
namespace oneapi::dal::%DATA_SOURCE% {

template <typename Object, typename DataSource>
class read_args {
public:
   read_args(
      const %PROPERTY_TYPE_1%& property_name_1,
      const %PROPERTY_TYPE_2%& property_name_2,
      /* more properties */
   )
   /* Getter & Setter for the property called `%PROPERTY_NAME_1%` */
   descriptor& set_%PROPERTY_NAME_1%(%PROPERTY_TYPE_1% value);
   %PROPERTY_TYPE_1% get_%PROPERTY_NAME_1%() const;
   /* Getter & Setter for the property called `%PROPERTY_NAME_2%` */
   descriptor& set_%PROPERTY_NAME_2%(%PROPERTY_TYPE_2% value);
   %PROPERTY_TYPE_2% get_%PROPERTY_NAME_2%() const;
   /* more properties */
};
} // namespace oneapi::dal::%DATA_SOURCE%
```

Result

The result of a **read** operation is an instance of an in-memory object with `Object` type.

Data Source Types

oneDAL defines a set of classes.

Data Source Types

Data source type	Description
CSV data source	Data source that allows reading data from a text file into a table.

Details

- CSV data source
  - Usage example
  - Programming Interface

CSV data source

Class `csv::data_source` is an API for accessing the data source represented as a csv file. CSV data source is used with **read** operation to extract data in text format from the given input file, process it using provided parameters (such as delimiter and read options), transform it into numerical representation, and store it as an in-memory dataset of a chosen type.

Supported type of in-memory object for **read** operation with CSV data source is **oneapi::dal::table**.

CSV data source requires input file name to be set in the constructor, while the other parameters of the constructor such as delimiter and read options rely on default values.

Usage example

```
using namespace oneapi;

const auto data_source = dal::csv::data_source("data.csv", ',');

const auto table = dal::read<dal::table>(data_source);
```

Programming Interface

Refer to API Reference: CSV data source.

Graphs

This section describes the types and functions related to the graph concept.

Type	Description
Undirected adjacency vector graph	An implementation of the undirected graph concept.
Directed adjacency vector graph	An implementation of the directed graph concept.
Graph traits	A standartized way to access various properties of the graph.

Requirements on graph types

Each implementation of graph concept:

1. Follows the definition of the graph concept and its restrictions (for example, immutability)
2. Is reference-counted
3. Defines graph_traits data type.

Graph types

Graph type	Description
undirected adjacency vector graph	A sparse undirectedweighted or unweighted graph that contains graph in CSR data format.
directed adjacency vector graph	A sparse directedweighted or unweighted graph that contains graph in CSR data format.

Undirected adjacency vector graph

Class `undirected_adjacency_vector_graph` is the implementation of undirectedweighted sparse graph concept with adjacency matrix underneath for which the following is true:

- The data within the graph is sparse and stored in CSR.
- The specific graph traits are defined for this class.

Programming interface

Refer to API Reference: Undirected Adjacency Vector Graph.

Directed adjacency vector graph

Class `directed_adjacency_vector_graph` is the implementation of directedweighted sparse graph concept with adjacency matrix underneath for which the following is true:

- The data within the graph is sparse and stored in CSR format.
- The specific graph traits are defined for this class.

Programming interface

Refer to API Reference: Directed Adjacency Vector Graph.

Tables

This section describes the types related to the table concept.

Table Types

Type	Description
table	A common implementation of the table concept. Base class for other table types.
table_metadata	An implementation of table metadata concept.
Data layout	An enumeration of data layouts used to store contiguous data blocks inside the table.
Feature type	An enumeration of feature types used in oneDAL to define set of available operations onto the data.

Requirements on table types

Each implementation of table concept:

1. Follows the definition of the table concept and its restrictions (e.g., immutability).
2. Is derived from the **oneapi::dal::table** class. The behavior of this class can be extended, but cannot be weaken.
3. Is reference-counted.
4. Defines a unique id number: the “kind” that represents objects of that type in runtime.

The following listing provides an example of table API to illustrate table kinds and copy-assignment operation:

```
using namespace onedal;

// Creating homogen_table sub-type.
dal::homogen_table table1 = homogen_table::wrap(queue, data_ptr, row_count, column_count);

// table1 and table2 share the same data (no data copy is performed)
dal::table table2 = table1;

// Creating an empty table
dal::table table3;

std::cout << table1.get_kind()     == table2.get_kind() << std::endl; // true
std::cout << homogen_table::kind() == table2.get_kind() << std::endl; // true
std::cout << table2.get_kind()     == table3.get_kind() << std::endl; // false

// Referring table3 to the table2.
table3 = table2;
std::cout << table2.get_kind() == table3.get_kind() << std::endl; // true
```

Table types

oneDAL defines a set of classes that implement the table concept for a specific data format:

Table Types for specific data formats

Table type	Description
homogen table	A dense table that contains contiguoushomogeneous data.

Programming interface

Refer to API: Tables.

Homogeneous table

Class `homogen_table` is a subtype of a table type for which the following is true:

- The data within the table are dense and stored as one contiguous memory block.
- All the columns have the same data type.

Programming interface

Refer to API Reference: Homogeneous table.

Algorithms

The Algorithms component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include matrix decompositions, clustering, classification, and regression algorithms, as well as association rules discovery.

- Clustering
  - DBSCAN
  - K-Means
  - K-Means initialization
- Covariance
  - Covariance
- Decomposition
  - Principal Components Analysis (PCA)
- Ensembles
  - Decision Forest Classification and Regression (DF)
- Graph
  - Subgraph Isomorphism
  - Connected Components
- Kernel Functions
  - Linear kernel
  - Polynomial kernel
  - Radial Basis Function (RBF) kernel
  - Sigmoid kernel
- Nearest Neighbors (kNN)
  - k-Nearest Neighbors Classification and Search (k-NN)
- Pairwise Distances
  - Minkowski distance
  - Chebyshev distance
  - Cosine distance
- Statistics
  - Basic Statistics
- Support Vector Machines
  - Support Vector Machine Classifier and Regression (SVM)

Clustering

This chapter describes clustering algorithms implemented in oneDAL:

- DBSCAN
- K-Means
- K-Means initialization

Examples: DBSCAN

oneAPI DPC++

Batch Processing:

- dpc_dbscan_brute_force_batch.cpp

oneAPI C++

Batch Processing:

- cpp_dbscan_brute_force_batch.cpp

Python* with DPC++ support

Batch Processing:

- dbscan_batch.py

Examples: K-Means

oneAPI DPC++

Batch Processing:

- dpc_kmeans_lloyd_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_lloyd_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- kmeans_batch.py

Examples: K-Means Initialization

oneAPI DPC++

Batch Processing:

- dpc_kmeans_init_dense.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_init_dense.cpp

DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed in [Ester96]. It is a density-based clustering non-parametric algorithm: given a set of observations in some space, it groups together observations that are closely packed together (observations with many nearby neighbors), marking as outliers observations that lie alone in low-density regions (whose nearest neighbors are too far away).

Operation	Computational methods	Programming Interface		
Compute	Default method	compute(...)	compute_input	compute_result

Mathematical formulation

Computation

Given the set $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors (further referred as observations), a positive floating-point number `epsilon` and a positive integer `minObservations`, the problem is to get clustering assignments for each input observation, based on the definitions below [Ester96]: two observations *x* and *y* are considered to be in the same cluster if there is a core observation*z*, and *x* and *y* are both reachable from *z*.

Each cluster gets a unique identifier, an integer number from **0** to $\text{total number of clusters} - 1$. Each observation is assigned an identifier of the cluster it belongs to, or **-1** if the observation considered to be a noise observation.

Programming Interface

Refer to API Reference: DBSCAN.

Distributed mode

The algorithm supports distributed execution in SMPD mode (only on GPU).

Usage example

Compute

```
void run_compute(const table& data,
                           const table& weights) {
   double epsilon = 1.0;
   std::int64_t max_observations = 5;
   const auto dbscan_desc = kmeans::descriptor<float>{epsilon, max_observations}
      .set_result_options(dal::dbscan::result_options::responses);

   const auto result = compute(dbscan_desc, data, weights);

   print_table("responses", result.get_responses());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_dbscan_brute_force_batch.cpp

oneAPI C++

Batch Processing:

- cpp_dbscan_brute_force_batch.cpp

Python* with DPC++ support

Batch Processing:

- dbscan_batch.py

K-Means

The K-Means algorithm solves clustering problem by partitioning *n* feature vectors into *k* clusters minimizing some criterion. Each cluster is characterized by a representative point, called *a centroid*.

Operation	Computational methods	Programming Interface		
Training	Lloyd’s	train(…)	train_input	train_result
Inference	Lloyd’s	infer(…)	infer_input	infer_result

Mathematical formulation

Training

Given the training set $X = \{x_1, \ldots, x_n\}$ of *p*-dimensional feature vectors and a positive integer *k*, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of *p*-dimensional centroids that minimize the objective function

$$\Phi_X(C) = \sum_{i=1}^{n} d^2(x_i, C),$$

where $d^2(x_i, C)$ is the squared Euclidean distance from x_i to the closest centroid in *C*,

$$d^2(x_i, C) = \min_{1 \le j \le k} \|x_i - c_j\|^2, \quad 1 \le i \le n.$$

Expression $\|\cdot\|$ denotes L_2norm.

> **NOTE** In the general case, *d* may be an arbitrary distance function. Current version of the oneDAL spec defines only Euclidean distance case.

Training method: *Lloyd’s*

The Lloyd’s method [Lloyd82] consists in iterative updates of centroids by applying the alternating *Assignment* and *Update* steps, where *t* denotes a index of the current iteration, e.g., $C^{(t)} = \{c_1^{(t)}, \ldots, c_k^{(t)}\}$ is the set of centroids at the *t*-th iteration. The method requires the initial centroids $C^{(1)}$ to be specified at the beginning of the algorithm ($t = 1$).

(1) Assignment step: Assign each feature vector x_i to the nearest centroid. $y_i^{(t)}$ denotes the assigned label (cluster index) to the feature vector x_i.

$$y_i^{(t)} = \arg\min_{1 \le j \le k} \|x_i - c_j^{(t)}\|^2, \quad 1 \le i \le n.$$

Each feature vector from the training set *X* is assigned to exactly one centroid so that *X* is partitioned to *k* disjoint sets (clusters)

$$S_j^{(t)} = \{\; x_i \in X \;:\; y_i^{(t)} = j \;\}, \quad 1 \le j \le k.$$

(2) Update step: Recalculate centroids by averaging feature vectors assigned to each cluster.

$$c_j^{(t+1)} = \frac{1}{|S_j^{(t)}|} \sum_{x \in S_j^{(t)}} x, \quad 1 \leq j \leq k.$$

The steps (1) and (2) are performed until the following **stop condition**,

$$\sum_{j=1}^{k} \left\| c_j^{(t)} - c_j^{(t+1)} \right\|^2 < \varepsilon,$$

is satisfied or number of iterations exceeds the maximal value *T* defined by the user.

Inference

Given the inference set $X' = \{x'_1, \ldots, x'_m\}$ of *p*-dimensional feature vectors and the set $C = \{c_1, \ldots, c_k\}$ of centroids produced at the training stage, the problem is to predict the index $y'_j \in \{0, \ldots, k-1\}$, $1 \leq j \leq m$, of the centroid in accordance with a method-defined rule.

Inference method: *Lloyd's*

Lloyd's inference method computes the y'_j as an index of the centroid closest to the feature vector x'_j,

$$y'_j = \arg\min_{1 \leq l \leq k} \|x'_j - c_l\|^2, \quad 1 \leq j \leq m.$$

Programming Interface

Refer to API Reference: K-Means.

Usage example

Training

```
kmeans::model<> run_training(const table& data,
                             const table& initial_centroids) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(10)
      .set_max_iteration_count(50)
      .set_accuracy_threshold(1e-4);

   const auto result = train(kmeans_desc, data, initial_centroids);

   print_table("labels", result.get_labels());
   print_table("centroids", result.get_model().get_centroids());
   print_value("objective", result.get_objective_function_value());

   return result.get_model();
}
```

Inference

```
table run_inference(const kmeans::model<>& model,
                    const table& new_data) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(model.get_cluster_count());
```

```
    const auto result = infer(kmeans_desc, model, new_data);

    print_table("labels", result.get_labels());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_kmeans_lloyd_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_lloyd_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- kmeans_batch.py

K-Means initialization

The K-Means initialization algorithm receives *n* feature vectors as input and chooses *k* initial centroids. After initialization, K-Means algorithm uses the initialization result to partition input data into *k* clusters.

Operation	Computational methods	Programming Interface		
Computing	Dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given the training set $X = \{x_1, \ldots, x_n\}$ of *p*-dimensional feature vectors and a positive integer *k*, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of *p*-dimensional initial centroids.

Computing method: *dense*

The method chooses first *k* feature vectors from the training set *X*.

Programming Interface

Refer to API Reference: K-Means initialization.

Usage example

Computing

```
table run_compute(const table& data) {
   const auto kmeans_desc = kmeans_init::descriptor<float,
                                                       kmeans_init::method::dense>{}
      .set_cluster_count(10)

   const auto result = compute(kmeans_desc, data);

   print_table("centroids", result.get_centroids());

   return result.get_centroids();
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_kmeans_init_dense.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_init_dense.cpp

Covariance

- Covariance

Examples: Covariance

oneAPI DPC++

Batch Processing:

- dpc_cor_dense_batch.cpp
- dpc_cov_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_cor_dense_batch.cpp
- cpp_cov_dense_batch.cpp

Covariance

Covariance algorithm computes the following set of quantitative dataset characteristics:

- means
- covariance
- correlation

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given a set *X* of *np*-dimensional feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$, the problem is to compute the sample means or the covariance matrix or the correlation matrix:

Statistic	Definition
Means	$M = (m(1), \ldots, m(p))$, where $m(j) = \frac{1}{n} \sum_i x_{ij}$
Covariance matrix	$Cov = (v_{ij})$, where $v_{ij} = \frac{1}{n-1} \sum_{k=1}^{n} (x_{ki} - m(i))(x_{kj} - m(j))$, $i = \overline{1, p}$, $j = \overline{1, p}$
Correlation matrix	$Cor = (c_{ij})$, where $c_{ij} = \frac{v_{ij}}{\sqrt{v_{ii} \cdot v_{jj}}}$, $i = \overline{1, p}$, $j = \overline{1, p}$

Computation method: *dense*

The method computes the means or the variance-covariance matrix or the correlation matrix

Programming Interface

Refer to API Reference: Covariance.

Distributed mode

The algorithm supports distributed execution in SMPD mode (only on GPU).

Decomposition

- Principal Components Analysis (PCA)

Examples: PCA

oneAPI DPC++

Batch Processing:

- dpc_pca_cor_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_pca_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- pca_batch.py

Principal Components Analysis (PCA)

Principal Component Analysis (PCA) is an algorithm for exploratory data analysis and dimensionality reduction. PCA transforms a set of feature vectors of possibly correlated features to a new set of uncorrelated features, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Operation	Computational methods	Programming Interface			
Training	Covariance	SVD	train(...)	train_input	train_result
Inference	Covariance	SVD	infer(...)	infer_input	infer_result

Mathematical formulation

Training

Given the training set $X = \{x_1, \dots, x_n\}$ of *p*-dimensional feature vectors and the number of principal components *r*, the problem is to compute *r* principal directions (*p*-dimensional eigenvectors [Lang87]) for the training set. The eigenvectors can be grouped into the $r \times p$ matrix *T* that contains one eigenvector in each row.

Training method: *Covariance*

This method uses eigenvalue decomposition of the covariance matrix to compute the principal components of the datasets. The method relies on the following steps:

1. Computation of the covariance matrix
2. Computation of the eigenvectors and eigenvalues
3. Formation of the matrices storing the results

Covariance matrix computation is performed in the following way:

1. Compute the vector-column of sums $s_i = \sum_{j=1}^{n} x_{i,j}, \quad 1 \leq i \leq p$.
2. Compute the cross-product $P = X^T X - s^T s$.
3. Compute the covariance matrix $\Sigma = \frac{1}{n-1} P$.

To compute eigenvalues λ_i and eigenvectors v_i, the implementer can choose an arbitrary method such as [Ping14].

The final step is to sort the set of pairs (λ_i, v_i) in the descending order by λ_i and form the resulting matrix $T = (v_{i,1}, \cdots, v_{i,r}), \quad 1 \leq i \leq p$. Additionally, the means and variances of the initial dataset are returned.

Training method: *SVD*

This method uses singular value decomposition of the dataset to compute its principal components. The method relies on the following steps:

1. Computation of the singular values and singular vectors
2. Formation of the matrices storing the results

To compute singular values λ_i and singular vectors u_i and v_i, the implementer can choose an arbitrary method such as [Demmel90].

The final step is to sort the set of pairs (λ_i, v_i) in the descending order by λ_i and form the resulting matrix $T = (v_{i,1}, \cdots, v_{i,r}), \quad 1 \leq i \leq p$. Additionally, the means and variances of the initial dataset are returned.

Sign-flip technique

Eigenvectors computed by some eigenvalue solvers are not uniquely defined due to sign ambiguity. To get the deterministic result, a sign-flip technique should be applied. One of the sign-flip techniques proposed in [Bro07] requires the following modification of matrix *T*:

$$\hat{T}_i = T_i \cdot \operatorname{sgn}(\max_{1 \le j \le p} |T_{ij}|), \quad 1 \le i \le r,$$

where T_i is *i*-th row, T_{ij} is the element in the *i*-th row and *j*-th column, $\operatorname{sgn}(\cdot)$ is the signum function,

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

Inference

Given the inference set $X' = \{x'_1, \ldots, x'_m\}$ of *p*-dimensional feature vectors and the $r \times p$ matrix *T* produced at the training stage, the problem is to transform X' to the set $X'' = \{x''_1, \ldots, x''_m\}$, where x''_j is an *r*-dimensional feature vector, $1 \le j \le m$.

The feature vector x''_j is computed through applying linear transformation [Lang87] defined by the matrix *T* to the feature vector x'_j,

$$x''_j = T x'_j, \quad 1 \le j \le m.$$

Inference methods: *Covariance* and *SVD*

Covariance and SVD inference methods compute x''_j according to (1).

Programming Interface

Refer to API Reference: Principal Components Analysis.

Distributed mode

The algorithm supports distributed execution in SMPD mode (only on GPU).

Usage example

Training

```
pca::model<> run_training(const table& data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(5)
      .set_deterministic(true);

   const auto result = train(pca_desc, data);

   print_table("means", result.get_means());
   print_table("variances", result.get_variances());
   print_table("eigenvalues", result.get_eigenvalues());
   print_table("eigenvectors", result.get_eigenvectors());
   return result.get_model();
}
```

Inference

```
table run_inference(const pca::model<>& model,
                  const table& new_data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(model.get_component_count());

   const auto result = infer(pca_desc, model, new_data);

   print_table("labels", result.get_transformed_data());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_pca_cor_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_pca_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- pca_batch.py

Ensembles

- Decision Forest Classification and Regression (DF)

Examples: Decifion Forest Classification

oneAPI DPC++

Batch Processing:

- dpc_df_cls_hist_batch.cpp

oneAPI C++

Batch Processing:

- cpp_df_cls_dense_batch.cpp

Examples: Decifion Forest Regression

oneAPI DPC++

Batch Processing:

- dpc_df_reg_hist_batch.cpp

oneAPI C++

Batch Processing:

- cpp_df_reg_dense_batch.cpp

Decision Forest Classification and Regression (DF)

Decision Forest (DF) classification and regression algorithms are based on an ensemble of tree-structured classifiers, which are known as decision trees. Decision forest is built using the general technique of bagging, a bootstrap aggregation, and a random choice of features. For more details, see [Breiman84] and [Breiman2001].

Operation	Computational methods	Programming Interface			
Training	Dense	Hist	train(...)	train_input	train_result
Inference	Dense	Hist	infer(...)	infer_input	infer_result

Mathematical formulation

Training

Given *n* feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of size *p*, their non-negative observation weights $W = \{w_1, \ldots, w_n\}$ and *n* responses $Y = \{y_1, \ldots, y_n\}$,

Classification

- $y_i \in \{0, \ldots, C - 1\}$, where *C* is the number of classes

Regression

- $y_i \in \mathbb{R}$

the problem is to build a decision forest classification or regression model.

The library uses the following algorithmic framework for the training stage. Let $S = (X, Y)$ be the set of observations. Given positive integer parameters, such as the number of trees *B*, the bootstrap parameter $N = f * n$, where *f* is a fraction of observations used for a training of each tree in the forest, and the number of features per node *m*, the algorithm does the following for $b = 1, \ldots, B$:

- Selects randomly with replacement the set D_b of *N* vectors from the set *S*. The set D_b is called a *bootstrap* set.
- Trains a decision tree classifier T_b on D_b using parameter *m* for each tree.

Decision tree*T* is trained using the training set *D* of size *N*. Each node *t* in the tree corresponds to the subset D_t of the training set *D*, with the root node being *D* itself. Each internal node *t* represents a binary test (split) that divides the subset X_t in two subsets, X_{t_L} and X_{t_R}, corresponding to their children, t_L and t_R.

Training method: *Dense*

In *dense* training method, all possible splits for each feature are taken from the subset of selected features for the current node and evaluated for best split computation.

Training method: *Hist*

In *hist* training method, only a selected subset of splits is considered for best split computation. This subset of splits is computed for each feature at the initialization stage of the algorithm. After computing the subset of splits, each value from the initially provided data is substituted with the value of the corresponding bin. Bins are continuous intervals between selected splits.

Split Criteria

The metric for measuring the best split is called *impurity*, *i(t)*. It generally reflects the homogeneity of responses within the subset D_t in the node *t*.

Classification

Gini index is an impurity metric for classification, calculated as follows:

$$I_{Gini}(D) = 1 - \sum_{i=0}^{C-1} p_i^2$$

where

- *D* is a set of observations that reach the node;
- p_i is specified in the table below:

Decision Forest Split Criteria Calculation

Without sample weights	With sample weights
p_i is the observed fraction of observations that belong to class *i* in *D*	p_i is the observed weighted fraction of observations that belong to class *i* in *D*: $p_i = \frac{\sum_{d \in \{d \in D \mid y_d = i\}} W_d}{\sum_{d \in D} W_d}$

Regression

MSE is an impurity metric for regression, calculated as follows:

MSE Impurity Metric

Without sample weights	With sample weights
$I_{\text{MSE}}(D) = \frac{1}{W(D)} \sum_{i=1}^{W(D)} \left(y_i - \frac{1}{W(D)} \sum_{j=1}^{W(D)} y_j\right)^2$	$I_{\text{MSE}}(D) = \frac{1}{W(D)} \sum_{i \in D} w_i \left(y_i - \frac{1}{W(D)} \sum_{j \in D} w_j y_j\right)$
$W(S) = \sum_{s \in S} 1$, which is equivalent to the number of elements in *S*	$W(S) = \sum_{s \in S} w_s$

Let the *impurity decrease* in the node *t* be

$$\Delta i(t) = i(t) - \frac{|D_{t_L}|}{|D_t|} i(t_L) - \frac{|D_{t_R}|}{|D_t|} i(t_R).$$

Termination Criteria

The library supports the following termination criteria of decision forest training:

Minimal number of observations in a leaf node	Node *t* is not processed if $\|D_t\|$ is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.
Minimal number of observations in a split node	Node *t* is not processed if $\|D_t\|$ is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.

Minimum weighted fraction of the sum total of weights of all the input observations required to be at a leaf node	Node *t* is not processed if $|D_t|$ is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.
Maximal tree depth	Node *t* is not processed if its depth in the tree reached the predefined value.
Impurity threshold	Node *t* is not processed if its impurity is smaller than the predefined threshold.
Maximal number of leaf nodes	Grow trees with positive maximal number of leaf nodes in a best-first fashion. Best nodes are defined by relative reduction in impurity. If maximal number of leaf nodes equals zero, then this criterion does not limit the number of leaf nodes, and trees grow in a depth-first fashion.

Tree Building Strategies

Maximal number of leaf nodes defines the strategy of tree building: depth-first or best-first.

Depth-first Strategy

If maximal number of leaf nodes equals zero, a decision tree is built using depth-first strategy. In each terminal node *t*, the following recursive procedure is applied:

- Stop if the termination criteria are met.
- Choose randomly without replacement *m* feature indices $J_t \in \{0, 1, \ldots, p-1\}$.
- For each $j \in J_t$, find the best split $s_{j,t}$ that partitions subset D_t and maximizes impurity decrease $\Delta i(t)$.
- A node is a split if this split induces a decrease of the impurity greater than or equal to the predefined value. Get the best split s_t that maximizes impurity decrease Δi in all $s_{j,t}$ splits.
- Apply this procedure recursively to t_L and t_R.

Best-first Strategy

If maximal number of leaf nodes is positive, a decision tree is built using best-first strategy. In each terminal node *t*, the following steps are applied:

- Stop if the termination criteria are met.
- Choose randomly without replacement *m* feature indices $J_t \in \{0, 1, \ldots, p-1\}$.
- For each $j \in J_t$, find the best split $s_{j,t}$ that partitions subset D_t and maximizes impurity decrease $\Delta i(t)$.
- A node is a split if this split induces a decrease of the impurity greater than or equal to the predefined value and the number of split nodes is less or equal to $\mathrm{maxLeafNodes}-1$. Get the best split s_t that maximizes impurity decrease Δi in all $s_{j,t}$ splits.
- Put a node into a sorted array, where sort criterion is the improvement in impurity $\Delta i(t)|D_t|$. The node with maximal improvement is the first in the array. For a leaf node, the improvement in impurity is zero.
- Apply this procedure to t_L and t_R and grow a tree one by one getting the first element from the array until the array is empty.

Inference

Given decision forest classification or regression model and vectors $x_1, \ldots, x_r$, the problem is to calculate the responses for those vectors.

Inference methods: *Dense* and *Hist*

Dense and *hist* inference methods perform prediction in the same way. To solve the problem for each given query vector x_i, the algorithm does the following:

Classification

For each tree in the forest, it finds the leaf node that gives x_i its label. The label *y* that the majority of trees in the forest vote for is chosen as the predicted label for the query vector x_i.

Regression

For each tree in the forest, it finds the leaf node that gives x_i the response as the mean of dependent variables. The mean of responses from all trees in the forest is the predicted response for the query vector x_i.

Additional Characteristics Calculated by the Decision Forest

Decision forests can produce additional characteristics, such as an estimate of generalization error and an importance measure (relative decisive power) of each of p features (variables).

Out-of-bag Error

The estimate of the generalization error based on the training data can be obtained and calculated as follows:

Classification

- For each vector x_i in the dataset *X*, predict its label $\hat{y_i}$ by having the majority of votes from the trees that contain x_i in their OOB set, and vote for that label.
- Calculate the OOB error of the decision forest *T* as the average of misclassifications:

$$OOB(T) = \frac{1}{|D'|}\sum_{y_i \in D'} I\{y_i \neq \hat{y}_i\}, \text{where } D' = \bigcup_{b=1}^{B} \overline{D_b}.$$

- If OOB error value per each observation is required, then calculate the prediction error for x_i: $OOB(x_i) = I\{y_i \neq \hat{y}_i\}$

Regression

- For each vector x_i in the dataset *X*, predict its response $\hat{y_i}$ as the mean of prediction from the trees that contain x_i in their OOB set:

$\hat{y_i} = \frac{1}{|B_i|}\sum_{b=1}^{|B_i|}\hat{y_{ib}}$, where $B_i = \bigcup T_b : x_i \in \overline{D_b}$ and $\hat{y_{ib}}$ is the result of prediction x_i by T_b.

- Calculate the OOB error of the decision forest *T* as the Mean-Square Error (MSE):

$$OOB(T) = \frac{1}{|D'|}\sum_{y_i \in D'} \sum (y_i - \hat{y}_i)^2, \text{where } D' = \bigcup_{b=1}^{B} \overline{D_b}$$

- If OOB error value per each observation is required, then calculate the prediction error for x_i:

$$OOB(x_i) = (y_i - \hat{y}_i)^2$$

Variable Importance

There are two main types of variable importance measures:

- *Mean Decrease Impurity* importance (MDI)

  Importance of the *j*-th variable for predicting *Y* is the sum of weighted impurity decreases $p(t)\Delta i(s_t, t)$ for all nodes *t* that use x_j, averaged over all *B* trees in the forest:

  $$MDI\left(j\right) = \frac{1}{B}\sum_{b=1}^{B} \sum_{t\in T_b: v(s_t)=j} p\left(t\right)\Delta i\left(s_t, t\right),$$

  where $p\left(t\right) = \frac{|D_t|}{|D|}$ is the fraction of observations reaching node *t* in the tree T_b, and $v(s_t)$ is the index of the variable used in split s_t.
- *Mean Decrease Accuracy* (MDA)

  Importance of the *j*-th variable for predicting *Y* is the average increase in the OOB error over all trees in the forest when the values of the *j*-th variable are randomly permuted in the OOB set. For that reason, this latter measure is also known as *permutation importance*.

  In more details, the library calculates MDA importance as follows:

  - Let $\pi(X, j)$ be the set of feature vectors where the *j*-th variable is randomly permuted over all vectors in the set.
  - Let E_b be the OOB error calculated for $T_b:$ on its out-of-bag dataset $\overline{D_b}$.
  - Let $E_{b,j}$ be the OOB error calculated for $T_b:$ using $\pi\left(\overline{X_b}, j\right)$, and its out-of-bag dataset $\overline{D_b}$ is permuted on the *j*-th variable. Then
    - $\delta_{b,j} = E_b - E_{b,j}$ is the OOB error increase for the tree T_b.
    - $RawMDA\left(j\right) = \frac{1}{B}\sum_{b=1}^{B}\delta_{b,j}$ is MDA importance.
    - $ScaledMDA\left(j\right) = \frac{Raw\ MDA(x_j)}{\frac{\sigma_j}{\sqrt{B}}}$, where σ_j^2 is the variance of $D_{b,j}$

Programming Interface

Refer to API Reference: Decision Forest Classification and Regression.

Distributed mode

The algorithm supports distributed execution in SMPD mode (only on GPU).

Graph

This chapter describes graph algorithms implemented in oneDAL:

- Subgraph Isomorphism
- Connected Components

Examples: Subgraph Isomorphism

oneAPI C++

Batch Processing:

- cpp_subgraph_isomorphism_batch.cpp

Examples: Connected Components

oneAPI C++

Batch Processing:

- cpp_connected_components_batch.cpp

Subgraph Isomorphism

Subgraph Isomorphism algorithm receives a target graph *G* and a pattern graph *H* as input and searches the target graph for subgraphs that are isomorphic to the pattern graph. The algorithm returns the mappings of the pattern graph vertices onto the target graph vertices.

Operation	**Computational methods**	**Programming Interface**		
Computing	fast	graph_matching(…)	graph_matching_input	graph_matching_result

Mathematical formulation

Subgraphs definition

A graph $H = (V'; E')$ is called a subgraph of graph $G = (V; E)$ if $V' \subseteq V; E' \subseteq E$ and V' contains all the endpoints of all the edges in E' [Gross2014].

Further we denote the induced subgraph on the vertex set as **induced** subgraph, the induced subgraph on the edge set as **non-induced** subgraph.

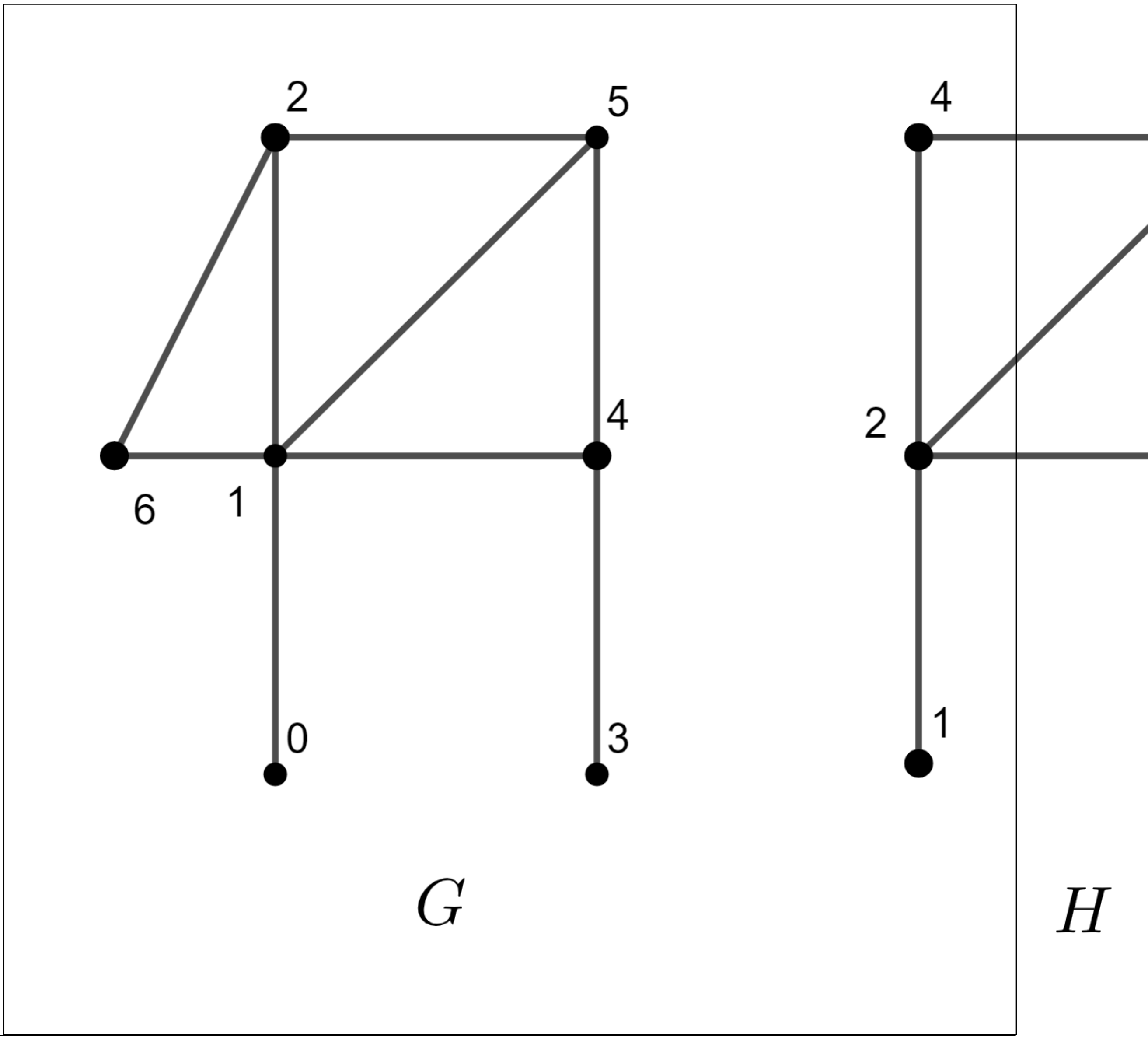
Computing

Given two graphs *G* and *H*, the problem is to determine whether graph *G* contains a subgraph isomorphic to graph *H* and find the exact mapping of subgraph *H* in graph *G*.

G is called **target** graph, *H* is called **pattern** graph.

Mapping is a bijection or one-to-one correspondence between vertices of *H* and a subgraph of graph *G*. Two vertices are adjacent if there is an existing edge between them, and non-adjacent otherwise. Induced subgraph isomorphism preserves both adjacency and non-adjacency relationships between vertices, while non-induced subgraph isomorphism preserves only adjacency relationship.

Example



For the example above, the mappings for subgraph *H* in graph *G* are:

- Induced: [3, 0, 1, 4, 2, 5]
- Non-induced: [3, 0, 1, 4, 2, 5], [3, 6, 1, 4, 2, 5], [6, 0, 1, 2, 4, 5], and [4, 0, 1, 5, 6, 2]

The notation [3, 0, 1, 4, 2, 5] means that:

- pattern vertex with id 0 is mapped on vertex in target graph with id 3
- pattern vertex with id 1 is mapped on vertex in target graph with id 0
- pattern vertex with id 2 is mapped on vertex in target graph with id 1
- pattern vertex with id 3 is mapped on vertex in target graph with id 4
- pattern vertex with id 4 is mapped on vertex in target graph with id 2
- pattern vertex with id 5 is mapped on vertex in target graph with id 5

Computation method: *fast*

The method defines VF3-light algorithms with Global State Stack parallelization method and supports induced and non-induced cases.

For more details, see [Carletti2021].

Programming Interface

Refer to API Reference: Subgraph Isomorphism.

Examples

oneAPI C++

Batch Processing:

- cpp_subgraph_isomorphism_batch.cpp

Connected Components

Connected components algorithm receives an undirected graph *G* as an input and searches for connected components in *G*. For each vertex in *G*, the algorithm returns the label of the component this vertex belongs to. The result of the algorithm is a set of labels for all vertices in *G*.

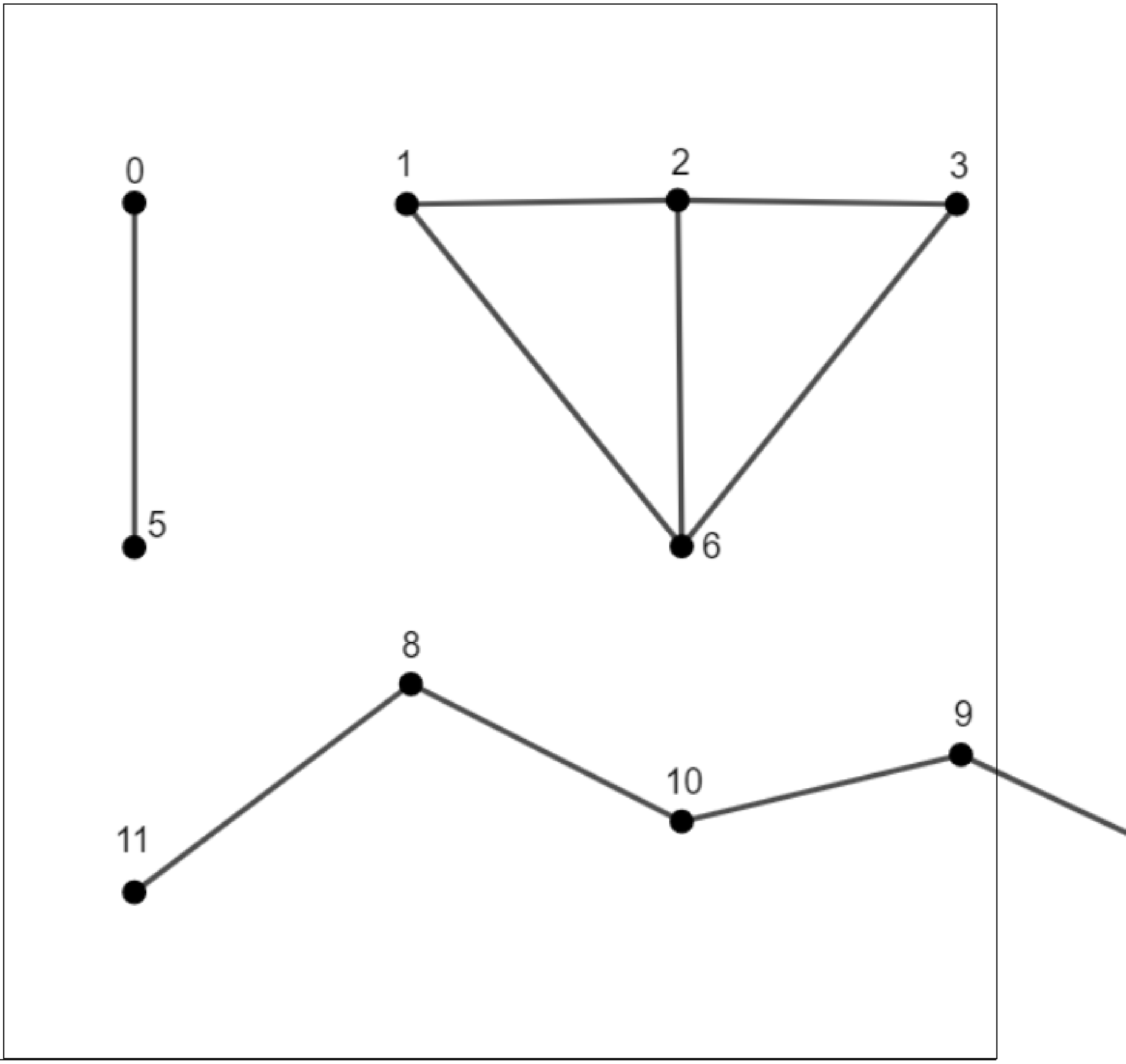
Operation	Computational methods	Programming Interface		
Computing	afforest	vertex_partitioning(...)	vertex_partitioning_input	vertex_partitioning_result

Mathematical formulation

Computing

Given an undirected graph*G*, the problem is to find connected components in *G*, determine their quantity, and label vertices so that vertices from the same component have the same label.

Example



Components are labeled from **0** to *k-1*, where *k* is the number of components. For the example above, the labels for vertices are [0, 1, 1, 1, 2, 0, 1, 3, 4, 4, 4, 4, 4].

This notation means that:

- vertices with ids 0 and 5 belong to the connected component with id 0
- vertices with ids 1, 2, 3, and 6 belong to the connected component with id 1
- vertex with id 4 belongs to the connected component with id 2
- vertex with id 7 belongs to the connected component with id 3
- vertices with ids 8, 9, 10, 11, and 12 belong to the connected component with id 4

Computation method: *afforest*

The method defines Afforest algorithm and solves the problem of connected components identification in an undirected graph.

This algorithm expands the Shiloach-Vishkin connected components algorithm and uses component approximation to decrease redundant edge processing. The method consists of the following steps:

1. Process a fixed number of edges for each vertex (Vertex Neighbor Sampling optimization).
2. Identify the largest intermediate component using probabilistic method.
3. Process the rest of the neighborhoods only for the vertices that do not belong to the largest component (Large Component Skipping optimization).

For more details, see [Sutton2018].

Programming Interface

Refer to API Reference: Connected Components.

Examples

oneAPI C++

Batch Processing:

- cpp_connected_components_batch.cpp

Kernel Functions

- Linear kernel
- Polynomial kernel
- Radial Basis Function (RBF) kernel
- Sigmoid kernel

Examples: Linear Kernel

oneAPI DPC++

Batch Processing:

- dpc_linear_kernel_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_linear_kernel_dense_batch.cpp

Examples: Polynomial Kernel

oneAPI C++

Batch Processing:

- cpp_polynomial_kernel_dense_batch.cpp

Examples: RBF Kernel

oneAPI DPC++

Batch Processing:

- dpc_rbf_kernel_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_rbf_kernel_dense_batch.cpp

Examples: Sigmoid Kernel

oneAPI C++

Batch Processing:

- cpp_sigmoid_kernel_dense_batch.cpp

Linear kernel

The linear kernel is the simplest kernel function for pattern analysis.

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \dots, x_{1p}), \dots, x_n = (x_{n1}, \dots, x_{np})$ of dimension *p* and a set *Y* of *m* feature vectors $y_1 = (y_{11}, \dots, y_{1p}), \dots, y_m = (y_{m1}, \dots, y_{mp})$, the problem is to compute the linear kernel function $K(x_i, y_i)$ for any pair of input vectors:

$$K(x_i, y_i) = k{X_i}^T y_i + b$$

Computation method: *dense*

The method computes the linear kernel function $K(X, Y)$ for *X* and *Y* matrices.

Programming Interface

Refer to API Reference: Linear kernel.

Polynomial kernel

The Polynomial kernel is a popular kernel function used in kernelized learning algorithms. It represents the similarity of training samples in a feature space of polynomials of the original data and allows to fit non-linear models.

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \dots, x_{1p}), \dots, x_n = (x_{n1}, \dots, x_{np})$ of dimension *p* and a set *Y* of *m* feature vectors $y_1 = (y_{11}, \dots, y_{1p}), \dots, y_m = (y_{m1}, \dots, y_{mp})$, the problem is to compute the polynomial kernel function $K(x_i, y_j)$ for any pair of input vectors:

$$K(x_i, y_j) = (k{x_i}^T y_j + b)^d,$$

where $k \in \mathbb{R},\ \ b \in \mathbb{R},\ \ d \in \{0,\ 1,\ 2,\ \dots\},\quad 1 \le i \le n,\quad 1 \le j \le m$.

Computation method: *dense*

The method computes the polynomial kernel function $K(X, Y)$ for *X* and *Y* matrices.

Programming Interface

Refer to API Reference: Polynomial kernel.

Radial Basis Function (RBF) kernel

The Radial Basis Function (RBF) kernel is a popular kernel function used in kernelized learning algorithms.

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p* and a set *Y* of *m* feature vectors $y_1 = (y_{11}, \ldots, y_{1p}), \ldots, y_m = (y_{m1}, \ldots, y_{mp})$, the problem is to compute the RBF kernel function $K(x_i, y_i)$ for any pair of input vectors:

$$K\left(x_i, y_j\right) = exp\left(-\frac{\left(\|x_i - y_j\|\right)^2}{2\sigma^2}\right)$$

Computation method: *dense*

The method computes the rbf kernel function $K(X, Y)$ for *X* and *Y* matrices.

Programming Interface

Refer to API Reference: Radial Basis Function (RBF) kernel.

Sigmoid kernel

The Sigmoid kernel is a popular kernel function used in kernelized learning algorithms.

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p* and a set *Y* of *m* feature vectors $y_1 = (y_{11}, \ldots, y_{1p}), \ldots, y_m = (y_{m1}, \ldots, y_{mp})$, the problem is to compute the sigmoid kernel function $K(x_i, y_j)$ for any pair of input vectors:

$$K(x_i, y_j) = \tanh(k{x_i}^T y_j + b),$$

where $k \in \mathbb{R},\ b \in \mathbb{R},\quad 1 \leq i \leq n,\quad 1 \leq j \leq m$.

Computation method: *dense*

The method computes the sigmoid kernel function $K(X, Y)$ for *X* and *Y* matrices.

Programming Interface

Refer to API Reference: Sigmoid kernel.

Nearest Neighbors (kNN)

- k-Nearest Neighbors Classification and Search (k-NN)

Examples: k-Nearest Neighbors

oneAPI DPC++

Batch Processing:

- dpc_knn_cls_brute_force_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_knn_cls_brute_force_dense_batch.cpp
- cpp_knn_cls_kd_tree_dense_batch.cpp
- cpp_knn_search_brute_force_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- bf_knn_classification_batch.py

k-Nearest Neighbors Classification and Search (k-NN)

k-NN classification and search algorithms are based on finding the *k* nearest observations to the training set. For classification, the problem is to infer the class of a new feature vector by computing the majority vote of its *k* nearest observations from the training set. For search, the problem is to infer *k* nearest observations from the training set to a new feature vector. The nearest observations are computed based on the chosen distance metric.

Operation	Computational methods	Programming Interface			
Training	Brute-force	k-d tree	train(...)	train_input	train_result
Inference	Brute-force	k-d tree	infer(...)	infer_input	infer_result

Mathematical formulation

Training

Classification

Let $X = \{x_1, \ldots, x_n\}$ be the training set of *p*-dimensional feature vectors, let $Y = \{y_1, \ldots, y_n\}$ be the set of class labels, where $y_i \in \{0, \ldots, C-1\}$, $1 \leq i \leq n$, and *C* is the number of classes. Given *X*, *Y*, and the number of nearest neighbors *k*, the problem is to build a model that allows distance computation between the feature vectors in training and inference sets at the inference stage.

Search

Let $X = \{x_1, \ldots, x_n\}$ be the training set of *p*-dimensional feature vectors. Given *X* and the number of nearest neighbors *k*, the problem is to build a model that allows distance computation between the feature vectors in training and inference sets at the inference stage.

Training method: *brute-force*

The training operation produces the model that stores all the feature vectors from the initial training set *X*.

Training method: *k-d tree*

The training operation builds a *k-d* tree that partitions the training set *X* (for more details, see k-d Tree).

Inference

Classification

Let $X' = \{x'_1, \ldots, x'_m\}$ be the inference set of *p*-dimensional feature vectors. Given X', the model produced at the training stage, and the number of nearest neighbors *k*, the problem is to predict the label y'_j from the *Y* set for each x'_j, $1 \leq j \leq m$, by performing the following steps:

1. Identify the set $N(x'_j) \subseteq X$ of *k* feature vectors in the training set that are nearest to x'_j with respect to the Euclidean distance, which is chosen by default. The distance can be customized with the predefined set of pairwise distances: Minkowski distances with fractional degree (including Euclidean distance), Chebyshev distance, and Cosine distance.
2. Estimate the conditional probability for the *l*-th class as the fraction of vectors in $N(x'_j)$ whose labels y_j are equal to *l*:

$$P_{jl} = \frac{1}{|N(x'_j)|} \Big| \{x_r \in N(x'_j) : y_r = l\} \Big|, \quad 1 \leq j \leq m, \; 0 \leq l < C.$$

3. Predict the class that has the highest probability for the feature vector x'_j:

$$y'_j = \arg\max_{0 \leq l < C} P_{jl}, \quad 1 \leq j \leq m.$$

Search

Let $X' = \{x'_1, \ldots, x'_m\}$ be the inference set of *p*-dimensional feature vectors. Given X', the model produced at the training stage, and the number of nearest neighbors *k*:

1. Identify the set $N(x'_j) \subseteq X$ of *k* feature vectors in the training set that are nearest to x'_j with respect to the Euclidean distance, which is chosen by default. The distance can be customized with the predefined set of pairwise distances: Minkowski distances with fractional degree (including Euclidean distance), Chebyshev distance, and Cosine distance.

Inference method: *brute-force*

Brute-force inference method determines the set $N(x'_j)$ of the nearest feature vectors by iterating over all the pairs (x'_j, x_i) in the implementation defined order, $1 \leq i \leq n$, $1 \leq j \leq m$.

Inference method: *k-d tree*

K-d tree inference method traverses the *k-d* tree to find feature vectors associated with a leaf node that are closest to x'_j, $1 \leq j \leq m$. The set $\tilde{n}(x'_j)$ of the currently known nearest *k* neighbors is progressively updated during the tree traversal. The search algorithm limits exploration of the nodes for which the distance between the x'_j and respective part of the feature space is not less than the distance between x'_j and the most distant feature vector from $\tilde{n}(x'_j)$. Once tree traversal is finished, $\tilde{n}(x'_j) \equiv N(x'_j)$.

Programming Interface

Refer to API Reference: k-Nearest Neighbors Classification and Search.

Usage example

Training

```
knn::model<> run_training(const table& data,
                          const table& labels) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = train(knn_desc, data, labels);

   return result.get_model();
}
```

Inference

```
table run_inference(const knn::model<>& model,
                    const table& new_data) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = infer(knn_desc, model, new_data);

   print_table("labels", result.get_labels());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_knn_cls_brute_force_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_knn_cls_brute_force_dense_batch.cpp
- cpp_knn_cls_kd_tree_dense_batch.cpp
- cpp_knn_search_brute_force_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- bf_knn_classification_batch.py

Pairwise Distances

- Minkowski distance
- Chebyshev distance
- Cosine distance

Minkowski distance

The Minkowski distances are the set of distance metrics with different degree $(p > 0)$ and are widely used for distance computation in different algorithms. The most commonly used distance metric, Euclidean distance, is also a Minkowski distance with $p = 2.0$.

Operation	Computational methods
dense	dense

Mathematical formulation

Computing

Given a set *U* of *n* feature vectors $u_1 = (u_{11}, \ldots, u_{1k}), \ldots, u_n = (u_{n1}, \ldots, u_{nk})$ of dimension *k* and a set *V* of *m* feature vectors $v_1 = (v_{11}, \ldots, v_{1k}), \ldots, v_m = (v_{m1}, \ldots, v_{mk})$ of dimension *k*, the problem is to compute the Minkowski distance $||u_i, v_j||_p$ for any pair of input vectors:

$$||u_i, v_j||_p = \sum_{l=1}^{k} (|u_{il} - v_{jl}|^p)^{1/p},$$

where $1 \leq i \leq n, \quad 1 \leq j \leq m, \quad p > 0$.

Computation method: *dense*

The method defines Minkowski distance metric, which is used in other algorithms for the distance computation. There are no separate computation mode to compute distance manually.

Programming Interface

Refer to API Reference: Minkowski distance.

Chebyshev distance

The Chebyshev distance equals the limit of Minkowski distance metric with $p \to \infty$.

Operation	Computational methods
dense	dense

Mathematical formulation

Computing

Given a set *U* of *n* feature vectors $u_1 = (u_{11}, \ldots, u_{1k}), \ldots, u_n = (u_{n1}, \ldots, u_{nk})$ of dimension *k* and a set *V* of *m* feature vectors $v_1 = (v_{11}, \ldots, v_{1k}), \ldots, v_m = (v_{m1}, \ldots, v_{mk})$ of dimension *k*, the problem is to compute the Chebyshev distance $||u_i, v_j||_\infty$ for any pair of input vectors:

$$||u_i, v_j||_\infty = \max_l |u_{il} - v_{jl}|,$$

where $1 \leq i \leq n, \quad 1 \leq j \leq m, \quad 1 \leq l \leq k$.

Computation method: *dense*

The method defines Chebyshev distance metric, which is used in other algorithms for the distance computation. There are no separate computation mode to compute distance manually.

Programming Interface

Refer to API Reference: Chebyshev distance.

Cosine distance

The Cosine distance is a measure of distance between two non-zero vectors of an inner product space.

Operation	Computational methods
dense	dense

Mathematical formulation

Computing

Given a set *U* of *n* feature vectors $u_1 = (u_{11}, \ldots, u_{1k}), \ldots, u_n = (u_{n1}, \ldots, u_{nk})$ of dimension *k* and a set *V* of *m* feature vectors $v_1 = (v_{11}, \ldots, v_{1k}), \ldots, v_m = (v_{m1}, \ldots, v_{mk})$ of dimension *k*, the problem is to compute the Cosine distance $D_{cos}(u_i, v_j)$ for any pair of input vectors:

$$D_{cos}(u_i, v_j) = 1 - \frac{\sum_{l=1}^{k} u_{il} v_{jl}}{\sqrt{\sum_{l=1}^{k} u_{il}^2} \sqrt{\sum_{l=1}^{k} v_{jl}^2}},$$

where $1 \leq i \leq n, \quad 1 \leq j \leq m$.

Computation method: *dense*

The method defines Cosine distance metric, which is used in other algorithms for the distance computation. There is no separate computation mode to compute the distance manually.

Programming Interface

Refer to API Reference: Cosine distance.

Statistics

- Basic Statistics

Examples: Basic statistics

oneAPI DPC++

Batch Processing:

- dpc_basic_statistics_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_basic_statistics_dense_batch.cpp

Basic Statistics

Basic statistics algorithm computes the following set of quantitative dataset characteristics:

- minimums/maximums
- sums
- means
- sums of squares
- sums of squared differences from the means
- second order raw moments
- variances
- standard deviations
- variations

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Computing

Given a set *X* of *np*-dimensional feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$, the problem is to compute the following sample characteristics for each feature in the data set:

Statistic	Definition
Minimum	$min(j) = \min_i \{x_{ij}\}$
Maximum	$max(j) = \max_i \{x_{ij}\}$
Sum	$s(j) = \sum_i x_{ij}$

Statistic	Definition
Sum of squares	$s_2(j) = \sum_i x_{ij}^2$
Means	$m(j) = \frac{s(j)}{n}$
Second order raw moment	$a_2(j) = \frac{s_2(j)}{n}$
Sum of squared difference from the means	$\text{SDM}(j) = \sum_i (x_{ij} - m(j))^2$
Variance	$k_2(j) = \frac{\text{SDM}(j)}{n-1}$
Standard deviation	$\text{stdev}(j) = \sqrt{k_2(j)}$
Variation coefficient	$V(j) = \frac{\text{stdev}(j)}{m(j)}$

Computation method: ***dense***

The method computes the basic statistics for each feature in the data set.

Programming Interface

Refer to API Reference: Basic statistics.

Distributed mode

The algorithm supports distributed execution in SMPD mode (only on GPU).

Support Vector Machines

- Support Vector Machine Classifier and Regression (SVM)

Examples: SVM

oneAPI DPC++

Batch Processing:

- dpc_svm_two_class_thunder_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_svm_two_class_smo_dense_batch.cpp
- cpp_svm_two_class_thunder_dense_batch.cpp
- cpp_svm_reg_thunder_dense_batch.cpp
- cpp_svm_multi_class_thunder_dense_batch.cpp
- cpp_svm_nu_cls_thunder_dense_batch.cpp
- cpp_svm_nu_reg_thunder_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- svm_batch.py

Support Vector Machine Classifier and Regression (SVM)

Support Vector Machine (SVM) classification and regression are among popular algorithms. It belongs to a family of generalized linear classification problems.

Operation	Computational methods	Programming Interface			
Training	SMO	Thunder	train(...)	train_input	train_result
Inference	SMO	Thunder	infer(...)	infer_input	infer_result

Mathematical formulation

Training

Given *n* feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of size *p*, their non-negative observation weights $W = \{w_1, \ldots, w_n\}$, and *n* responses $Y = \{y_1, \ldots, y_n\}$,

Classification

- $y_i \in \{0, \ldots, M - 1\}$, where *M* is the number of classes

Regression

- $y_i \in \mathbb{R}$

Nu-classification

- $y_i \in \{0, \ldots, M - 1\}$, where *M* is the number of classes

Nu-regression

- $y_i \in \mathbb{R}$

the problem is to build a Support Vector Machine (SVM) classification, regression, nu-classification, or nu-regression model.

The SVM model is trained using the Sequential minimal optimization (SMO) method [Boser92] for reduced to the solution of the quadratic optimization problem

Classification

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$

with $0 \leq \alpha_i \leq C$, $i = 1, \ldots, n$, $y^T \alpha = 0$, where *e* is the vector of ones, *C* is the upper bound of the coordinates of the vector α, *Q* is a symmetric matrix of size $nimesn$ with $Q_{ij} = y_i y_j K(x_i, x_j)$, and $K(x, y)$ is a kernel function.

Regression

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - s^T \alpha$$

with $0 \leq \alpha_i \leq C$, $i = 1, \ldots, 2n$, $z^T \alpha = 0$, where *C* is the upper bound of the coordinates of the vector α, *Q* is a symmetric matrix of size $2n \times 2n$ with $Q_{ij} = y_i y_j K(x_i, x_j)$, and $K(x, y)$ is a kernel function. Vectors *s* and *z* for the regression problem are formulated according to the following rule:

$$\begin{cases} z_i = +1, s_i = \epsilon - y_i, & 0 < i \leq n \\ z_i = -1, s_i = \epsilon + y_i, & n < i \leq 2n \end{cases}$$

Where ϵ is the error tolerance parameter.

Nu-classification

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha$$

with $0 \leq \alpha_i \leq 1$, $i = 1, \ldots, n$, $e^T \alpha = n\nu$, $y^T \alpha = 0$, where *e* is the vector of ones, ν is an upper bound on the fraction of training errors and a lower bound of the fraction of the support vector, *Q* is a symmetric matrix of size $nimesn$ with $Q_{ij} = y_i y_j K(x_i, x_j)$, and $K(x, y)$ is a kernel function.

Nu-regression

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha + z^T \alpha$$

with $0 \leq \alpha_i \leq \frac{C}{n}$, $i = 1, \ldots, 2n$, $\sum_{i=1}^{n} \alpha_i - \sum_{i=n+1}^{2n} \alpha_i = 0$, $\sum_{i=1}^{2n} \alpha_i = C\nu$, where *C* is the upper bound of the coordinates of the vector α, ν is an upper bound on the fraction of training errors and a lower bound of the fraction of the support vector, *Q* is a symmetric matrix of size $2n \times 2n$ with $Q_{ij} = y_i y_j K(x_i, x_j)$, and $K(x, y)$ is a kernel function. Vector *z* for the regression problem are formulated according to the following rule:

$$\begin{cases} z_i = y_i, & 0 < i \leq n \\ z_i = y_{i-n}, & n < i \leq 2n \end{cases}$$

Working subset of α updated on each iteration of the algorithm is based on the Working Set Selection (WSS) 3 scheme [Fan05]. The scheme can be optimized using one of these techniques or both:

- **Cache**: the implementation can allocate a predefined amount of memory to store intermediate results of the kernel computation.
- **Shrinking**: the implementation can try to decrease the amount of kernel related computations (see [Joachims99]).

The solution of the problem defines the separating hyperplane and corresponding decision function $D(x) = \sum_k y_k \alpha_k K(x_k, x) + b$, where only those x_k that correspond to non-zero α_k appear in the sum, and *b* is a bias. Each non-zero α_k is called a dual coefficient and the corresponding x_k is called a support vector.

Training method: *smo*

In *smo* training method, all vectors from the training dataset are used for each iteration.

Training method: *thunder*

In *thunder* training method, the algorithm iteratively solves the convex optimization problem with the linear constraints by selecting the fixed set of active constrains (working set) and applying Sequential Minimal Optimization (SMO) solver to the selected subproblem. The description of this method is given in Algorithm [Wen2018].

Inference methods: *smo* and *thunder*

smo and *thunder* inference methods perform prediction in the same way:

Given the SVM classification or regression model and *r* feature vectors $x_1, \ldots, x_r$, the problem is to calculate the signed value of the decision function $D(x_i)$, $i = 1, \ldots, r$. The sign of the value defines the class of the feature vector, and the absolute value of the function is a multiple of the distance between the feature vector and the separating hyperplane.

Programming Interface

Refer to API Reference: Support Vector Machine Classifier and Regression.

Examples

oneAPI DPC++

Batch Processing:

- dpc_svm_two_class_thunder_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_svm_two_class_smo_dense_batch.cpp
- cpp_svm_two_class_thunder_dense_batch.cpp
- cpp_svm_reg_thunder_dense_batch.cpp
- cpp_svm_multi_class_thunder_dense_batch.cpp
- cpp_svm_nu_cls_thunder_dense_batch.cpp
- cpp_svm_nu_reg_thunder_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- svm_batch.py

Single Program Multiple Data

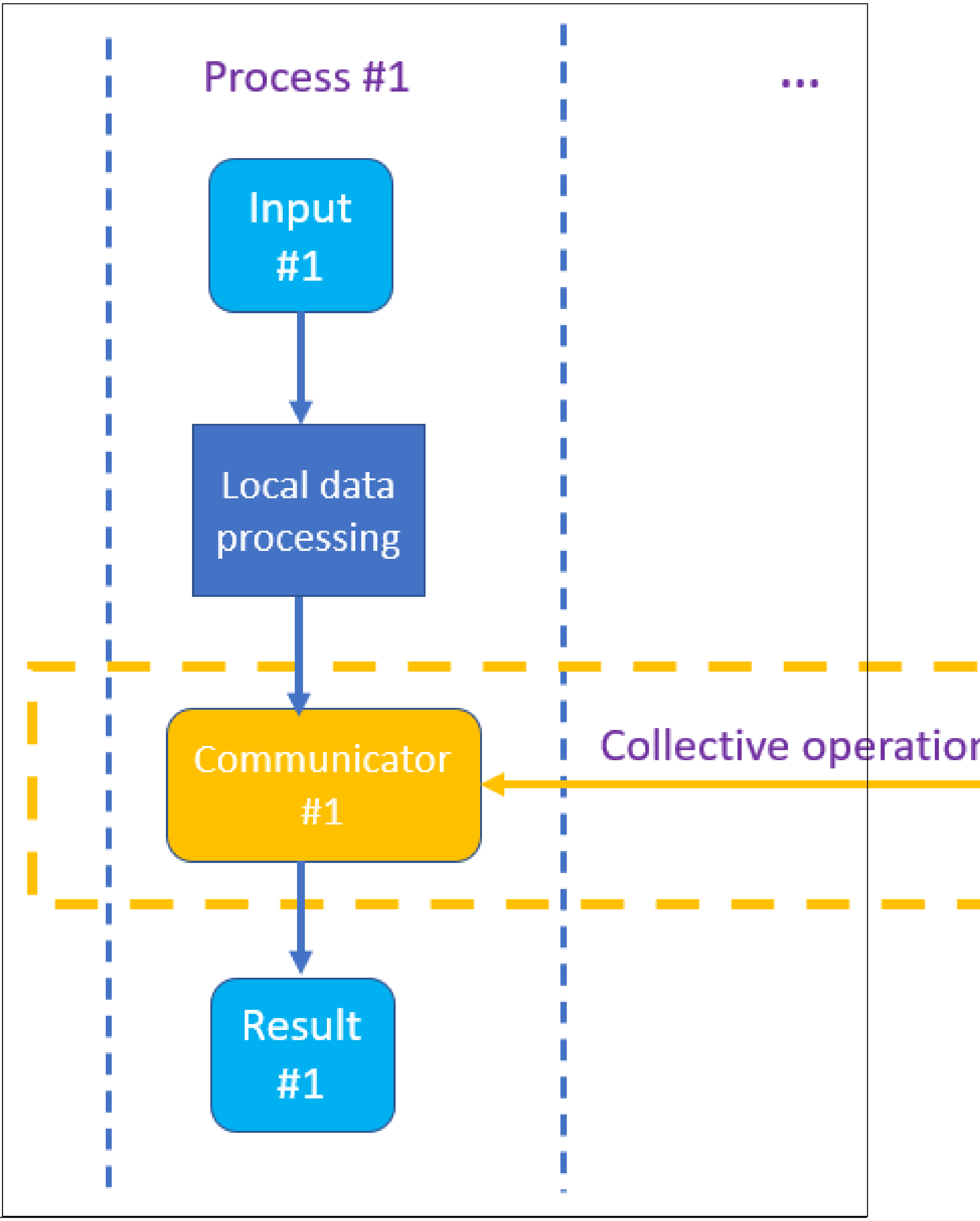
This section includes concepts and descriptions of objects that support distributed computations using SPMD model.

Distributed computation using SPMD model

In a typical usage scenario, a user provides a communicator object as a first parameter of a free function to indicate that the algorithm can process data simultaneously. All internal inter-process communications at sync points are hidden from the user.

General expectation is that input dataset is distributed among processes. Results are distributed in accordance with the input.

Example of SPMD Flow in oneDAL



Supported Collective Operations

The following collective operations are supported:

- `bcast` Broadcasts data from specified process.
- `allreduce` Reduces data among all processes.
- `allgatherv` Gathers data from all processes and shares the result among all processes.
- `sendrecv_replace` Sends and receives data using a single buffer.

Backend-specific restrictions

- oneCCL: Allgetherv does not support arbitrary displacements. The result is expected to be closely packed without gaps.
- oneMPI: Collective operations in oneMPI do not support asynchronous executions. They block the process till completion.

oneAPI Examples

- DPC++
  - basic_statistics_dense_batch.cpp
  - column_accessor_homogen.cpp
  - cor_dense_batch.cpp
  - cov_dense_batch.cpp
  - dbscan_brute_force_batch.cpp
  - df_cls_hist_batch.cpp
  - df_cls_traverse_model.cpp
  - df_reg_hist_batch.cpp
  - df_reg_traverse_model.cpp
  - kmeans_init_dense.cpp
  - kmeans_lloyd_dense_batch.cpp
  - knn_cls_brute_force_dense_batch.cpp
  - knn_reg_brute_force_dense_batch.cpp
  - knn_search_brute_force_dense_batch.cpp
  - linear_kernel_dense_batch.cpp
  - linear_regression_dense_batch.cpp
  - pca_cor_dense_batch.cpp
  - pca_precomputed_cor_dense_batch.cpp
  - pca_precomputed_cov_dense_batch.cpp
  - rbf_kernel_dense_batch.cpp
  - svm_two_class_thunder_dense_batch.cpp
- C++
  - basic_statistics_dense_batch.cpp
  - column_accessor_homogen.cpp
  - connected_components_batch.cpp
  - cor_dense_batch.cpp
  - cov_dense_batch.cpp
  - dbscan_brute_force_batch.cpp
  - df_cls_dense_batch.cpp
  - df_reg_dense_batch.cpp
  - directed_graph.cpp
  - graph_service_functions.cpp
  - jaccard_batch.cpp
  - jaccard_batch_app.cpp
  - kmeans_init_dense.cpp
  - kmeans_lloyd_dense_batch.cpp

- knn_cls_brute_force_dense_batch.cpp
- knn_cls_kd_tree_dense_batch.cpp
- knn_search_brute_force_dense_batch.cpp
- linear_kernel_dense_batch.cpp
- linear_regression_dense_batch.cpp
- louvain_batch.cpp
- pca_dense_batch.cpp
- pca_precomputed_dense_batch.cpp
- polynomial_kernel_dense_batch.cpp
- rbf_kernel_dense_batch.cpp
- shortest_paths_batch.cpp
- sigmoid_kernel_dense_batch.cpp
- subgraph_isomorphism_batch.cpp
- svm_multi_class_thunder_dense_batch.cpp
- svm_nu_cls_thunder_dense_batch.cpp
- svm_nu_reg_thunder_dense_batch.cpp
- svm_reg_thunder_dense_batch.cpp
- svm_two_class_smo_dense_batch.cpp
- svm_two_class_thunder_dense_batch.cpp
- triangle_counting_batch.cpp

oneAPI DPC++ examples

- basic_statistics_dense_batch.cpp
- column_accessor_homogen.cpp
- cor_dense_batch.cpp
- cov_dense_batch.cpp
- dbscan_brute_force_batch.cpp
- df_cls_hist_batch.cpp
- df_cls_traverse_model.cpp
- df_reg_hist_batch.cpp
- df_reg_traverse_model.cpp
- kmeans_init_dense.cpp
- kmeans_lloyd_dense_batch.cpp
- knn_cls_brute_force_dense_batch.cpp
- knn_reg_brute_force_dense_batch.cpp
- knn_search_brute_force_dense_batch.cpp
- linear_kernel_dense_batch.cpp
- linear_regression_dense_batch.cpp
- pca_cor_dense_batch.cpp
- pca_precomputed_cor_dense_batch.cpp
- pca_precomputed_cov_dense_batch.cpp
- rbf_kernel_dense_batch.cpp
- svm_two_class_thunder_dense_batch.cpp

basic_statistics_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/basic_statistics.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    const auto data_file_name = get_data_path("covcormoments_dense.csv");

    const auto data = dal::read<dal::table>(q, dal::csv::data_source{ data_file_name });

    const auto bs_desc = dal::basic_statistics::descriptor{};

    const auto result = dal::compute(q, bs_desc, data);

    std::cout << "Minimum:\n" << result.get_min() << std::endl;
    std::cout << "Maximum:\n" << result.get_max() << std::endl;
    std::cout << "Sum:\n" << result.get_sum() << std::endl;
    std::cout << "Sum of squares:\n" << result.get_sum_squares() << std::endl;
    std::cout << "Sum of squared difference from the means:\n"
              << result.get_sum_squares_centered() << std::endl;
    std::cout << "Mean:\n" << result.get_mean() << std::endl;
    std::cout << "Second order raw moment:\n" << result.get_second_order_raw_moment() <<
std::endl;
    std::cout << "Variance:\n" << result.get_variance() << std::endl;
    std::cout << "Standard deviation:\n" << result.get_standard_deviation() << std::endl;
    std::cout << "Variation:\n" << result.get_variation() << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

column_accessor_homogen.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
```

```
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>
#include <iostream>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/table/column_accessor.hpp"
#include "oneapi/dal/table/homogen.hpp"

#include "example_util/dpc_helpers.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    constexpr std::int64_t row_count = 6;
    constexpr std::int64_t column_count = 2;
    const float data_host[] = {
        0.f, 6.f, 1.f, 7.f, 2.f, 8.f, 3.f, 9.f, 4.f, 10.f, 5.f, 11.f,
    };

    auto data = sycl::malloc_shared<float>(row_count * column_count, q);
    q.memcpy(data, data_host, sizeof(float) * row_count * column_count).wait();

    auto table = dal::homogen_table{ q,
                                     data,
                                     row_count,
                                     column_count,
                                     dal::detail::make_default_delete<const float>(q) };
    dal::column_accessor<const float> acc{ table };

    for (std::int64_t col = 0; col < table.get_column_count(); col++) {
        std::cout << "column " << col << " values: ";

        const auto col_values = acc.pull(q, col);
        for (std::int64_t i = 0; i < col_values.get_count(); i++) {
            std::cout << col_values[i] << ", ";
        }
        std::cout << std::endl;
    }
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
```

```
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

cor_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/covariance.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    const auto input_file_name = get_data_path("covcormoments_dense.csv");

    const auto input = dal::read<dal::table>(q, dal::csv::data_source{ input_file_name });
    const auto cov_desc = dal::covariance::descriptor{}.set_result_options(
        dal::covariance::result_options::cor_matrix | dal::covariance::result_options::means);

    const auto result = dal::compute(q, cov_desc, input);

    std::cout << "Means:\n" << result.get_means() << std::endl;
    std::cout << "Cor:\n" << result.get_cor_matrix() << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

cov_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/covariance.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    const auto input_file_name = get_data_path("covcormoments_dense.csv");

    const auto input = dal::read<dal::table>(q, dal::csv::data_source{ input_file_name });
    auto cov_desc = dal::covariance::descriptor{}.set_result_options(
        dal::covariance::result_options::cov_matrix);

    auto result = dal::compute(q, cov_desc, input);

    std::cout << "Cov:\n" << result.get_cov_matrix() << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

dbscan_brute_force_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>
#include <iomanip>
#include <iostream>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/dbscan.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    const auto data_file_name = get_data_path("dbscan_dense.csv");

    const auto x_data = dal::read<dal::table>(q, dal::csv::data_source{ data_file_name });

    double epsilon = 0.04;
    std::int64_t min_observations = 45;

    auto dbscan_desc = dal::dbscan::descriptor<>(epsilon, min_observations);
    dbscan_desc.set_result_options(dal::dbscan::result_options::responses);

    const auto result_compute = dal::compute(q, dbscan_desc, x_data);

    std::cout << "Cluster count: " << result_compute.get_cluster_count() << std::endl;
    std::cout << "Responses:\n" << result_compute.get_responses() << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
```

```
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

df_cls_hist_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/decision_forest.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"
#include "oneapi/dal/exceptions.hpp"

namespace dal = oneapi::dal;
namespace df = dal::decision_forest;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("df_classification_train_data.csv");
    const auto train_response_file_name = get_data_path("df_classification_train_label.csv");
    const auto test_data_file_name = get_data_path("df_classification_test_data.csv");
    const auto test_response_file_name = get_data_path("df_classification_test_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });

    const auto df_desc =
        df::descriptor<float, df::method::hist, df::task::classification>{}
            .set_class_count(5)
            .set_tree_count(10)
            .set_features_per_node(x_train.get_column_count())
            .set_min_observations_in_leaf_node(8)
            .set_min_observations_in_split_node(16)
            .set_min_weight_fraction_in_leaf_node(0.0)
            .set_min_impurity_decrease_in_split_node(0.0)
            .set_error_metric_mode(df::error_metric_mode::out_of_bag_error)
            .set_variable_importance_mode(df::variable_importance_mode::mdi)
            .set_infer_mode(df::infer_mode::class_responses |
df::infer_mode::class_probabilities)
            .set_voting_mode(df::voting_mode::weighted);

    try {
        const auto result_train = dal::train(q, df_desc, x_train, y_train);

        std::cout << "Variable importance results:\n"
                  << result_train.get_var_importance() << std::endl;

        std::cout << "OOB error: " << result_train.get_oob_err() << std::endl;

        const auto result_infer = dal::infer(q, df_desc, result_train.get_model(), x_test);

        std::cout << "Prediction results:\n" << result_infer.get_responses() << std::endl;
        std::cout << "Probabilities results:\n" << result_infer.get_probabilities() << std::endl;

        std::cout << "Ground truth:\n" << y_test << std::endl;
    }
    catch (dal::unimplemented& e) {
        std::cout << "  " << e.what() << std::endl;
        return;
    }
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

df_cls_traverse_model.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/decision_forest.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"
#include "oneapi/dal/exceptions.hpp"

namespace dal = oneapi::dal;
namespace df = dal::decision_forest;

/* Decision forest parameters */
const std::int64_t class_count = 5; /* Number of classes */

/** Visitor class, prints out tree nodes of the model when it is called back by model traversal
method */
struct print_node_visitor {
    bool operator()(const df::leaf_node_info<df::task::classification>& info) {
        std::cout << std::string(info.get_level() * 2, ' ');
        std::cout << "Level " << info.get_level()
                  << ", leaf node. Response value = " << info.get_response()
                  << ", Impurity = " << info.get_impurity()
                  << ", Number of samples = " << info.get_sample_count() << ", Probabilities =
{ ";
        for (std::int64_t index_class = 0; index_class < class_count; ++index_class) {
            std::cout << info.get_probability(index_class) << ' ';
        }
        std::cout << "}" << std::endl;
        return true;
    }

    bool operator()(const df::split_node_info<df::task::classification>& info) {
        std::cout << std::string(info.get_level() * 2, ' ');
        std::cout << "Level " << info.get_level()
                  << ", split node. Feature index = " << info.get_feature_index()
                  << ", feature value = " << info.get_feature_value()
                  << ", Impurity = " << info.get_impurity()
                  << ", Number of samples = " << info.get_sample_count() << std::endl;
        return true;
    }
};

template <typename Task>
void print_model(const df::model<Task>& m) {
    std::cout << "Number of trees: " << m.get_tree_count() << std::endl;
    for (std::int64_t i = 0, n = m.get_tree_count(); i < n; ++i) {
        std::cout << "Tree #" << i << std::endl;
        m.traverse_depth_first(i, print_node_visitor{});
    }
}

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("df_classification_train_data.csv");
    const auto train_response_file_name = get_data_path("df_classification_train_label.csv");
    const auto test_data_file_name = get_data_path("df_classification_test_data.csv");
    const auto test_response_file_name = get_data_path("df_classification_test_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });

    const auto df_desc = df::descriptor<float, df::method::hist, df::task::classification>{}
                             .set_class_count(class_count)
                             .set_tree_count(2)
                             .set_features_per_node(1)
                             .set_min_observations_in_leaf_node(8)
                             .set_min_observations_in_split_node(16)
                             .set_min_weight_fraction_in_leaf_node(0.0)
                             .set_min_impurity_decrease_in_split_node(0.0)
                             .set_max_tree_depth(15);

    try {
        const auto result_train = dal::train(q, df_desc, x_train, y_train);
        print_model(result_train.get_model());
    }
    catch (dal::unimplemented& e) {
        std::cout << "  " << e.what() << std::endl;
        return;
    }
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

df_reg_hist_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/decision_forest.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"
#include "oneapi/dal/exceptions.hpp"

namespace dal = oneapi::dal;
namespace df = dal::decision_forest;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("df_regression_train_data.csv");
    const auto train_response_file_name = get_data_path("df_regression_train_label.csv");
    const auto test_data_file_name = get_data_path("df_regression_test_data.csv");
    const auto test_response_file_name = get_data_path("df_regression_test_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });

    const auto df_desc =
        df::descriptor<float, df::method::hist, df::task::regression>{}
            .set_tree_count(100)
            .set_features_per_node(0)
            .set_min_observations_in_leaf_node(1)
            .set_error_metric_mode(df::error_metric_mode::out_of_bag_error |
                                   df::error_metric_mode::out_of_bag_error_per_observation)
            .set_variable_importance_mode(df::variable_importance_mode::mdi);

    try {
        const auto result_train = dal::train(q, df_desc, x_train, y_train);

        std::cout << "Variable importance results:\n"
                  << result_train.get_var_importance() << std::endl;

        std::cout << "OOB error: " << result_train.get_oob_err() << std::endl;
        std::cout << "OOB error per observation:\n"
                  << result_train.get_oob_err_per_observation() << std::endl;

        const auto result_infer = dal::infer(q, df_desc, result_train.get_model(), x_test);

        std::cout << "Prediction results:\n" << result_infer.get_responses() << std::endl;

        std::cout << "Ground truth:\n" << y_test << std::endl;
    }
    catch (dal::unimplemented& e) {
        std::cout << "  " << e.what() << std::endl;
        return;
    }
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << std::endl
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

df_reg_traverse_model.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/decision_forest.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"
#include "oneapi/dal/exceptions.hpp"

namespace dal = oneapi::dal;
namespace df = dal::decision_forest;

/** Visitor class, prints out tree nodes of the model when it is called back by model traversal
method */
struct print_node_visitor {
    bool operator()(const df::leaf_node_info<df::task::regression>& info) {
        std::cout << std::string(info.get_level() * 2, ' ');
        std::cout << "Level " << info.get_level()
                  << ", leaf node. Response value = " << info.get_response()
                  << ", Impurity = " << info.get_impurity()
                  << ", Number of samples = " << info.get_sample_count() << std::endl;
        return true;
    }

    bool operator()(const df::split_node_info<df::task::regression>& info) {
        std::cout << std::string(info.get_level() * 2, ' ');
        std::cout << "Level " << info.get_level()
                  << ", split node. Feature index = " << info.get_feature_index()
                  << ", feature value = " << info.get_feature_value()
                  << ", Impurity = " << info.get_impurity()
                  << ", Number of samples = " << info.get_sample_count() << std::endl;
        return true;
    }
};

template <typename Task>
void print_model(const df::model<Task>& m) {
    std::cout << "Number of trees: " << m.get_tree_count() << std::endl;
    for (std::int64_t i = 0, n = m.get_tree_count(); i < n; ++i) {
        std::cout << "Tree #" << i << std::endl;
        m.traverse_depth_first(i, print_node_visitor{});
    }
}

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("df_regression_train_data.csv");
    const auto train_response_file_name = get_data_path("df_regression_train_label.csv");
    const auto test_data_file_name = get_data_path("df_regression_test_data.csv");
    const auto test_response_file_name = get_data_path("df_regression_test_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });

    const auto df_desc = df::descriptor<float, df::method::hist, df::task::regression>{}
                             .set_tree_count(2)
                             .set_features_per_node(0)
                             .set_min_observations_in_leaf_node(1);

    try {
        const auto result_train = dal::train(q, df_desc, x_train, y_train);
        print_model(result_train.get_model());
    }
    catch (dal::unimplemented& e) {
        std::cout << "  " << e.what() << std::endl;
        return;
    }
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

kmeans_init_dense.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>
#include <iomanip>
#include <iostream>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/kmeans.hpp"
#include "oneapi/dal/algo/kmeans_init.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

template <typename Method>
void run(sycl::queue& q, const dal::table& x_train, const std::string& method_name) {
    constexpr std::int64_t cluster_count = 20;
    constexpr std::int64_t max_iteration_count = 1000;
    constexpr double accuracy_threshold = 0.01;

    const auto kmeans_init_desc =
        dal::kmeans_init::descriptor<float, Method>().set_cluster_count(cluster_count);

    const auto result_init = dal::compute(q, kmeans_init_desc, x_train);

    const auto kmeans_desc = dal::kmeans::descriptor<>()
                                 .set_cluster_count(cluster_count)
                                 .set_max_iteration_count(max_iteration_count)
                                 .set_accuracy_threshold(accuracy_threshold);

    const auto result_train = dal::train(q, kmeans_desc, x_train, result_init.get_centroids());

    std::cout << "Method: " << method_name << std::endl;
    std::cout << "Max iteration count: " << max_iteration_count
```

```
              << ", Accuracy threshold: " << accuracy_threshold << std::endl;
    std::cout << "Iteration count: " << result_train.get_iteration_count()
              << ", Objective function value: " << result_train.get_objective_function_value()
              << '\n'
              << std::endl;
}

int main(int argc, char const* argv[]) {
    const auto train_data_file_name = get_data_path("kmeans_init_dense.csv");

    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << '\n'
                  << std::endl;
        auto q = sycl::queue{ d };

        const auto x_train =
            dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });

        run<dal::kmeans_init::method::dense>(q, x_train, "dense");
        run<dal::kmeans_init::method::random_dense>(q, x_train, "random_dense");
    }
    return 0;
}
```

kmeans_lloyd_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>
#include <iomanip>
#include <iostream>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/kmeans.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
```

```
void run(sycl::queue &q) {
    const auto train_data_file_name = get_data_path("kmeans_dense_train_data.csv");
    const auto initial_centroids_file_name = get_data_path("kmeans_dense_train_centroids.csv");
    const auto test_data_file_name = get_data_path("kmeans_dense_test_data.csv");
    const auto test_response_file_name = get_data_path("kmeans_dense_test_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto initial_centroids =
        dal::read<dal::table>(q, dal::csv::data_source{ initial_centroids_file_name });

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });

    const auto kmeans_desc = dal::kmeans::descriptor<>()
                                 .set_cluster_count(20)
                                 .set_max_iteration_count(5)
                                 .set_accuracy_threshold(0.001);

    const auto result_train = dal::train(q, kmeans_desc, x_train, initial_centroids);

    std::cout << "Iteration count: " << result_train.get_iteration_count() << std::endl;
    std::cout << "Objective function value: " << result_train.get_objective_function_value()
              << std::endl;
    std::cout << "Responses:\n" << result_train.get_responses() << std::endl;
    std::cout << "Centroids:\n" << result_train.get_model().get_centroids() << std::endl;

    const auto result_test = dal::infer(q, kmeans_desc, result_train.get_model(), x_test);

    std::cout << "Infer result:\n" << result_test.get_responses() << std::endl;

    std::cout << "Ground truth:\n" << y_test << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

knn_cls_brute_force_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
```

```
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/knn.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "oneapi/dal/exceptions.hpp"
#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("k_nearest_neighbors_train_data.csv");
    const auto train_response_file_name = get_data_path("k_nearest_neighbors_train_label.csv");
    const auto test_data_file_name = get_data_path("k_nearest_neighbors_test_data.csv");
    const auto test_response_file_name = get_data_path("k_nearest_neighbors_test_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    const auto knn_desc_uniform = dal::knn::descriptor(5, 1);
    const auto knn_desc_distance =
        dal::knn::descriptor(5, 1).set_voting_mode(dal::knn::voting_mode::distance);

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });

    const auto train_result_uniform = dal::train(q, knn_desc_uniform, x_train, y_train);
    const auto train_result_distance = dal::train(q, knn_desc_distance, x_train, y_train);

    const auto test_result_uniform =
        dal::infer(q, knn_desc_uniform, x_test, train_result_uniform.get_model());
    const auto test_result_distance =
        dal::infer(q, knn_desc_distance, x_test, train_result_distance.get_model());

    std::cout << "Test results (uniform voting):\n"
              << test_result_uniform.get_responses() << std::endl;
    std::cout << "Test results (distance voting):\n"
              << test_result_distance.get_responses() << std::endl;
    std::cout << "True responses:\n" << y_test << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

knn_reg_brute_force_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/knn.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "oneapi/dal/exceptions.hpp"
#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("knn_regression_train_data.csv");
    const auto train_response_file_name = get_data_path("knn_regression_train_responses.csv");
    const auto test_data_file_name = get_data_path("knn_regression_test_data.csv");
    const auto test_response_file_name = get_data_path("knn_regression_test_responses.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    using float_t = float;
    using method_t = dal::knn::method::by_default;
    using task_t = dal::knn::task::regression;
    using descriptor_t = dal::knn::descriptor<float_t, method_t, task_t>;

    const auto knn_desc_uniform = descriptor_t(5);
    const auto knn_desc_distance =
descriptor_t(5).set_voting_mode(dal::knn::voting_mode::distance);

    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(q,
dal::csv::data_source{ test_response_file_name });
```

```
    const auto train_result_uniform = dal::train(q, knn_desc_uniform, x_train, y_train);
    const auto train_result_distance = dal::train(q, knn_desc_distance, x_train, y_train);

    const auto test_result_uniform =
        dal::infer(q, knn_desc_uniform, x_test, train_result_uniform.get_model());
    const auto test_result_distance =
        dal::infer(q, knn_desc_distance, x_test, train_result_distance.get_model());

    std::cout << "Test results (uniform regression):\n"
              << test_result_uniform.get_responses() << std::endl;
    std::cout << "Test results (distance regression):\n"
              << test_result_distance.get_responses() << std::endl;
    std::cout << "True responses:\n" << y_test << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        // TODO: Should be deleted after regression algorithm introduction on CPU
        try {
            run(q);
        }
        catch (const dal::unimplemented& e) {
            std::cout << e.what() << std::endl;
        }
    }
    return 0;
}
```

knn_search_brute_force_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/knn.hpp"
#include "oneapi/dal/io/csv.hpp"
```

```
#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace knn = dal::knn;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("k_nearest_neighbors_train_data.csv");
    const auto query_data_file_name = get_data_path("k_nearest_neighbors_test_data.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto x_query = dal::read<dal::table>(q, dal::csv::data_source{ query_data_file_name });

    const std::size_t neighbors_count = 6;

    const auto knn_desc =
        knn::descriptor<float, knn::method::brute_force, knn::task::search>(neighbors_count)
            .set_result_options(knn::result_options::indices);

    const auto train_result = dal::train(q, knn_desc, x_train);
    const auto test_result = dal::infer(q, knn_desc, x_query, train_result.get_model());

    std::cout << "Indices result:\n" << test_result.get_indices() << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

linear_kernel_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
```

```
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/linear_kernel.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    std::cout << "Running on " << q.get_device().get_info<sycl::info::device::name>() << "\n"
              << std::endl;

    const auto data_file_name = get_data_path("kernel_function.csv");

    const auto x = dal::read<dal::table>(q, dal::csv::data_source{ data_file_name });
    const auto y = dal::read<dal::table>(q, dal::csv::data_source{ data_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);

    const auto result = dal::compute(q, kernel_desc, x, y);

    std::cout << "Values:\n" << result.get_values() << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

linear_regression_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
```

```
#endif

#include "oneapi/dal/algo/linear_regression.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "oneapi/dal/exceptions.hpp"
#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("linear_regression_train_data.csv");
    const auto train_response_file_name = get_data_path("linear_regression_train_responses.csv");
    const auto test_data_file_name = get_data_path("linear_regression_test_data.csv");
    const auto test_response_file_name = get_data_path("linear_regression_test_responses.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });
    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto lr_desc = dal::linear_regression::descriptor<>();

    const auto train_result = dal::train(q, lr_desc, x_train, y_train);
    const auto lr_model = train_result.get_model();

    const auto test_result_uniform = dal::infer(lr_desc, x_test, lr_model);

    std::cout << "Test results:\n" << test_result_uniform.get_responses() << std::endl;
    std::cout << "True responses:\n" << y_test << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

pca_cor_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iomanip>
#include <iostream>
#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/pca.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("pca_normalized.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });

    const auto pca_desc =
dal::pca::descriptor<>().set_component_count(5).set_deterministic(true);

    const auto result_train = dal::train(q, pca_desc, x_train);

    std::cout << "Eigenvectors:\n" << result_train.get_eigenvectors() << std::endl;

    std::cout << "Eigenvalues:\n" << result_train.get_eigenvalues() << std::endl;

    const auto result_infer = dal::infer(q, pca_desc, result_train.get_model(), x_train);

    std::cout << "Transformed data:\n" << result_infer.get_transformed_data() << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

pca_precomputed_cor_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iomanip>
#include <iostream>
#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/pca.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("precomputed_correlation.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    using float_t = float;
    using method_t = dal::pca::method::precomputed;
    using task_t = dal::pca::task::dim_reduction;
    using descriptor_t = dal::pca::descriptor<float_t, method_t, task_t>;
    const auto pca_desc = descriptor_t().set_component_count(5).set_deterministic(true);

    const auto result_train = dal::train(q, pca_desc, x_train);

    std::cout << "Eigenvectors:\n" << result_train.get_eigenvectors() << std::endl;

    std::cout << "Eigenvalues:\n" << result_train.get_eigenvalues() << std::endl;

    const auto result_infer = dal::infer(q, pca_desc, result_train.get_model(), x_train);

    std::cout << "Transformed data:\n" << result_infer.get_transformed_data() << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

pca_precomputed_cov_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2022 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iomanip>
#include <iostream>
#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/pca.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue& q) {
    const auto train_data_file_name = get_data_path("precomputed_covariance.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    using float_t = float;
    using method_t = dal::pca::method::precomputed;
    using task_t = dal::pca::task::dim_reduction;
    using descriptor_t = dal::pca::descriptor<float_t, method_t, task_t>;
    const auto pca_desc = descriptor_t().set_component_count(5).set_deterministic(true);

    const auto result_train = dal::train(q, pca_desc, x_train);

    std::cout << "Eigenvectors:\n" << result_train.get_eigenvectors() << std::endl;

    std::cout << "Eigenvalues:\n" << result_train.get_eigenvalues() << std::endl;

    const auto result_infer = dal::infer(q, pca_desc, result_train.get_model(), x_train);

    std::cout << "Transformed data:\n" << result_infer.get_transformed_data() << std::endl;
}

int main(int argc, char const* argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

rbf_kernel_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/rbf_kernel.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    const auto data_file_name = get_data_path("kernel_function.csv");

    const auto x = dal::read<dal::table>(q, dal::csv::data_source{ data_file_name });
    const auto y = dal::read<dal::table>(q, dal::csv::data_source{ data_file_name });

    const auto kernel_desc = dal::rbf_kernel::descriptor{}.set_sigma(1.0);
    const auto result = dal::compute(q, kernel_desc, x, y);

    std::cout << "Values:\n" << result.get_values() << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

svm_two_class_thunder_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <CL/sycl.hpp>

#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run(sycl::queue &q) {
    const auto train_data_file_name = get_data_path("svm_two_class_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_two_class_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_two_class_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_two_class_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(q, dal::csv::data_source{ train_data_file_name });
    const auto y_train =
        dal::read<dal::table>(q, dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);
    const auto svm_desc = dal::svm::descriptor{ kernel_desc }
                              .set_c(1.0)
                              .set_accuracy_threshold(0.001)
                              .set_max_iteration_count(100)
                              .set_cache_size(200.0)
                              .set_tau(1e-6);

    const auto result_train = dal::train(q, svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Support indices:\n" << result_train.get_support_indices() << std::endl;
    const auto x_test = dal::read<dal::table>(q, dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_test = dal::infer(q, svm_desc, result_train.get_model(), x_test);

    std::cout << "Decision function result:\n" << result_test.get_decision_function() <<
std::endl;
    std::cout << "Responses result:\n" << result_test.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;
}

int main(int argc, char const *argv[]) {
    for (auto d : list_devices()) {
        std::cout << "Running on " << d.get_platform().get_info<sycl::info::platform::name>()
                  << ", " << d.get_info<sycl::info::device::name>() << "\n"
                  << std::endl;
        auto q = sycl::queue{ d };
        run(q);
    }
    return 0;
}
```

oneAPI C++ Examples

- basic_statistics_dense_batch.cpp
- column_accessor_homogen.cpp
- connected_components_batch.cpp
- cor_dense_batch.cpp
- cov_dense_batch.cpp
- dbscan_brute_force_batch.cpp
- df_cls_dense_batch.cpp
- df_reg_dense_batch.cpp
- directed_graph.cpp
- graph_service_functions.cpp
- jaccard_batch.cpp
- jaccard_batch_app.cpp
- kmeans_init_dense.cpp
- kmeans_lloyd_dense_batch.cpp
- knn_cls_brute_force_dense_batch.cpp
- knn_cls_kd_tree_dense_batch.cpp
- knn_search_brute_force_dense_batch.cpp
- linear_kernel_dense_batch.cpp
- linear_regression_dense_batch.cpp
- louvain_batch.cpp
- pca_dense_batch.cpp
- pca_precomputed_dense_batch.cpp
- polynomial_kernel_dense_batch.cpp
- rbf_kernel_dense_batch.cpp
- shortest_paths_batch.cpp
- sigmoid_kernel_dense_batch.cpp
- subgraph_isomorphism_batch.cpp
- svm_multi_class_thunder_dense_batch.cpp
- svm_nu_cls_thunder_dense_batch.cpp
- svm_nu_reg_thunder_dense_batch.cpp
- svm_reg_thunder_dense_batch.cpp
- svm_two_class_smo_dense_batch.cpp

- svm_two_class_thunder_dense_batch.cpp
- triangle_counting_batch.cpp

basic_statistics_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/basic_statistics.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto data_file_name = get_data_path("covcormoments_dense.csv");

    const auto data = dal::read<dal::table>(dal::csv::data_source{ data_file_name });

    const auto bs_desc = dal::basic_statistics::descriptor{};

    const auto result = dal::compute(bs_desc, data);

    std::cout << "Minimum:\n" << result.get_min() << std::endl;
    std::cout << "Maximum:\n" << result.get_max() << std::endl;
    std::cout << "Sum:\n" << result.get_sum() << std::endl;
    std::cout << "Sum of squares:\n" << result.get_sum_squares() << std::endl;
    std::cout << "Sum of squared difference from the means:\n"
              << result.get_sum_squares_centered() << std::endl;
    std::cout << "Mean:\n" << result.get_mean() << std::endl;
    std::cout << "Second order raw moment:\n" << result.get_second_order_raw_moment() <<
std::endl;
    std::cout << "Variance:\n" << result.get_variance() << std::endl;
    std::cout << "Standard deviation:\n" << result.get_standard_deviation() << std::endl;
    std::cout << "Variation:\n" << result.get_variation() << std::endl;

    return 0;
}
```

column_accessor_homogen.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iostream>

#include "oneapi/dal/table/column_accessor.hpp"
#include "oneapi/dal/table/homogen.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    constexpr std::int64_t row_count = 6;
    constexpr std::int64_t column_count = 2;
    const float data[] = {
        0.f, 6.f, 1.f, 7.f, 2.f, 8.f, 3.f, 9.f, 4.f, 10.f, 5.f, 11.f,
    };

    auto table = dal::homogen_table::wrap(data, row_count, column_count);
    dal::column_accessor<const float> acc{ table };

    for (std::int64_t col = 0; col < table.get_column_count(); col++) {
        std::cout << "column " << col << " values: ";

        const auto col_values = acc.pull(col);
        for (std::int64_t i = 0; i < col_values.get_count(); i++) {
            std::cout << col_values[i] << ", ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

connected_components_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
```

```
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/connected_components.hpp"
#include "oneapi/dal/graph/undirected_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

int main(int argc, char** argv) {
    const auto filename = get_data_path("graph.csv");

    // read the graph
    using graph_t = dal::preview::undirected_adjacency_vector_graph<>;
    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename });

    // set algorithm parameters
    const auto cc_desc = dal::preview::connected_components::descriptor<>();

    // compute connected components
    const auto result_connected_components = dal::preview::vertex_partitioning(cc_desc, graph);

    // extract the result
    std::cout << "Components' labels:\n" << result_connected_components.get_labels() <<
std::endl;
    std::cout << "Number of connected components: "
              << result_connected_components.get_component_count() << std::endl;
    return 0;
}
```

cor_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/covariance.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"
```

```
namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto input_file_name = get_data_path("covcormoments_dense.csv");

    const auto input = dal::read<dal::table>(dal::csv::data_source{ input_file_name });
    const auto cov_desc = dal::covariance::descriptor{}.set_result_options(
        dal::covariance::result_options::cor_matrix | dal::covariance::result_options::means);

    const auto result = dal::compute(cov_desc, input);

    std::cout << "Means:\n" << result.get_means() << std::endl;
    std::cout << "Cor:\n" << result.get_cor_matrix() << std::endl;

    return 0;
}
```

cov_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/covariance.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto input_file_name = get_data_path("covcormoments_dense.csv");

    const auto input = dal::read<dal::table>(dal::csv::data_source{ input_file_name });
    auto cov_desc = dal::covariance::descriptor{}.set_result_options(
        dal::covariance::result_options::cov_matrix);

   auto result = dal::compute(cov_desc, input);

std::cout << "Cov:\n" << result.get_cov_matrix() << std::endl;

return 0;
}
```

dbscan_brute_force_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/dbscan.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto data_file_name = get_data_path("dbscan_dense.csv");

    const auto x_data = dal::read<dal::table>(dal::csv::data_source{ data_file_name });

    double epsilon = 0.04;
    std::int64_t min_observations = 45;
    auto dbscan_desc = dal::dbscan::descriptor<>(epsilon, min_observations);
    dbscan_desc.set_result_options(dal::dbscan::result_options::responses);

    const auto result_compute = dal::compute(dbscan_desc, x_data);

    std::cout << "Cluster count: " << result_compute.get_cluster_count() << std::endl;
    std::cout << "Responses:\n" << result_compute.get_responses() << std::endl;
    return 0;
}
```

df_cls_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/decision_forest.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;
namespace df = dal::decision_forest;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("df_classification_train_data.csv");
    const auto train_response_file_name = get_data_path("df_classification_train_label.csv");
    const auto test_data_file_name = get_data_path("df_classification_test_data.csv");
    const auto test_response_file_name = get_data_path("df_classification_test_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto df_desc =
        df::descriptor<>{}
            .set_class_count(5)
            .set_tree_count(10)
            .set_features_per_node(1)
            .set_min_observations_in_leaf_node(8)
            .set_min_observations_in_split_node(16)
            .set_min_weight_fraction_in_leaf_node(0.0)
            .set_min_impurity_decrease_in_split_node(0.0)
            .set_variable_importance_mode(df::variable_importance_mode::mdi)
            .set_error_metric_mode(df::error_metric_mode::out_of_bag_error)
            .set_infer_mode(df::infer_mode::class_responses |
df::infer_mode::class_probabilities)
            .set_voting_mode(df::voting_mode::weighted);

    const auto result_train = dal::train(df_desc, x_train, y_train);

    std::cout << "Variable importance results:\n" << result_train.get_var_importance() <<
std::endl;

    std::cout << "OOB error: " << result_train.get_oob_err() << std::endl;

    const auto result_infer = dal::infer(df_desc, result_train.get_model(), x_test);

    std::cout << "Prediction results:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Probabilities results:\n" << result_infer.get_probabilities() << std::endl;

    std::cout << "Ground truth:\n" << y_test << std::endl;

    return 0;
}
```

df_reg_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/decision_forest.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;
namespace df = dal::decision_forest;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("df_regression_train_data.csv");
    const auto train_response_file_name = get_data_path("df_regression_train_label.csv");
    const auto test_data_file_name = get_data_path("df_regression_test_data.csv");
    const auto test_response_file_name = get_data_path("df_regression_test_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto df_desc =
        df::descriptor<float, df::method::dense, df::task::regression>{}
            .set_tree_count(100)
            .set_features_per_node(0)
            .set_min_observations_in_leaf_node(1)
            .set_error_metric_mode(df::error_metric_mode::out_of_bag_error |
                                   df::error_metric_mode::out_of_bag_error_per_observation)
            .set_variable_importance_mode(df::variable_importance_mode::mda_raw);

    const auto result_train = dal::train(df_desc, x_train, y_train);

    std::cout << "Variable importance results:\n" << result_train.get_var_importance() <<
std::endl;

    std::cout << "OOB error: " << result_train.get_oob_err() << std::endl;
    std::cout << "OOB error per observation:\n"
              << result_train.get_oob_err_per_observation() << std::endl;

    const auto result_infer = dal::infer(df_desc, result_train.get_model(), x_test);

    std::cout << "Prediction results:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Ground truth:\n" << y_test << std::endl;

    return 0;
}
```

directed_graph.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iostream>

#include "example_util/utils.hpp"
#include "oneapi/dal/graph/service_functions.hpp"
#include "oneapi/dal/graph/directed_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"

using namespace std;

namespace dal = oneapi::dal;

int main(int argc, char** argv) {
    const auto filename = get_data_path("weighted_edge_list.csv");

    using vertex_type = int32_t;
    using weight_type = double;
    using graph_t = dal::preview::directed_adjacency_vector_graph<vertex_type, weight_type>;

    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename },
                                          dal::preview::read_mode::weighted_edge_list);

    std::cout << "Number of vertices: " << dal::preview::get_vertex_count(graph) << std::endl;
    std::cout << "Number of edges: " << dal::preview::get_edge_count(graph) << std::endl;

    dal::preview::vertex_outward_edge_size_type<graph_t> vertex_id = 0;
    std::cout << "Degree of " << vertex_id << ": "
              << dal::preview::get_vertex_outward_degree(graph, vertex_id) << std::endl;

    for (dal::preview::vertex_outward_edge_size_type<graph_t> j = 0;
         j < dal::preview::get_vertex_count(graph);
         ++j) {
        std::cout << "Neighbors of " << j << ": ";
        const auto neigh = dal::preview::get_vertex_outward_neighbors(graph, j);
        for (auto i = neigh.first; i != neigh.second; ++i) {
            std::cout << *i << "-" << dal::preview::get_edge_value(graph, j, *i) << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

graph_service_functions.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iostream>

#include "example_util/utils.hpp"
#include "oneapi/dal/graph/service_functions.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

int main(int argc, char **argv) {
    const auto filename = get_data_path("graph.csv");

    using graph_t = dal::preview::undirected_adjacency_vector_graph<>;
    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename });
    std::cout << "Number of vertices: " << dal::preview::get_vertex_count(graph) << std::endl;
    std::cout << "Number of edges: " << dal::preview::get_edge_count(graph) << std::endl;

    dal::preview::vertex_edge_size_type<graph_t> vertex_id = 0;
    std::cout << "Degree of " << vertex_id << ": "
              << dal::preview::get_vertex_degree(graph, vertex_id) << std::endl;

    for (dal::preview::vertex_edge_size_type<graph_t> j = 0;
         j < dal::preview::get_vertex_count(graph);
         ++j) {
        std::cout << "Neighbors of " << j << ": ";
        const auto neigh = dal::preview::get_vertex_neighbors(graph, j);
        for (auto i = neigh.first; i != neigh.second; ++i) {
            std::cout << *i << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

jaccard_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iostream>

#include "example_util/output_helpers_graph.hpp"
#include "example_util/utils.hpp"
#include "oneapi/dal/algo/jaccard.hpp"
#include "oneapi/dal/graph/undirected_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"
#include "oneapi/dal/table/common.hpp"

namespace dal = oneapi::dal;

int main(int argc, char **argv) {
    const auto filename = get_data_path("graph.csv");

    // read the graph
    using graph_t = dal::preview::undirected_adjacency_vector_graph<>;
    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename });

    // set blocks ranges
    const std::int64_t row_range_begin = 0;
    const std::int64_t row_range_end = 2;
    const std::int64_t column_range_begin = 0;
    const std::int64_t column_range_end = 3;

    // set algorithm parameters
    const auto jaccard_desc =
        dal::preview::jaccard::descriptor<>().set_block({ row_range_begin, row_range_end },
                                                        { column_range_begin,
column_range_end });

    // create caching builder for jaccard result
    dal::preview::jaccard::caching_builder builder;

    // compute Jaccard similarity coefficients
    const auto result_vertex_similarity =
        dal::preview::vertex_similarity(jaccard_desc, graph, builder);

    // extract the result
    const auto jaccard_coeffs = result_vertex_similarity.get_coeffs();
    const auto vertex_pairs = result_vertex_similarity.get_vertex_pairs();
    const std::int64_t nonzero_coeff_count = result_vertex_similarity.get_nonzero_coeff_count();

    std::cout << "The number of nonzero Jaccard coeffs in the block: " << nonzero_coeff_count
              << std::endl;

    print_vertex_similarity_result(result_vertex_similarity);
}
```

jaccard_batch_app.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iostream>

#include "tbb/global_control.h"
#include "tbb/parallel_for.h"

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/jaccard.hpp"
#include "oneapi/dal/graph/service_functions.hpp"
#include "oneapi/dal/graph/undirected_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"
#include "oneapi/dal/table/homogen.hpp"

namespace dal = oneapi::dal;

/// Computes Jaccard similarity coefficients for the graph. The upper triangular
/// matrix is processed only as it is symmetic for undirected graph.
///
/// @param [in]   g  The input graph
/// @param [in]   block_row_count    The size of block by rows
/// @param [in]   block_column_count The size of block by columns
template <class Graph>
void vertex_similarity_block_processing(const Graph &g,
                                        std::int32_t block_row_count,
                                        std::int32_t block_column_count);

int main(int argc, char **argv) {
    // load the graph
    const auto filename = get_data_path("graph.csv");

    using graph_t = dal::preview::undirected_adjacency_vector_graph<>;
    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename });

    // set the block sizes for Jaccard similarity block processing
    const std::int32_t block_row_count = 2;
    const std::int32_t block_column_count = 5;

    // set the number of threads
    const std::int32_t tbb_threads_number = 4;
    tbb::global_control c(tbb::global_control::max_allowed_parallelism, tbb_threads_number);

    // compute Jaccard similarity coefficients for the graph
    vertex_similarity_block_processing(graph, block_row_count, block_column_count);

    return 0;
}

template <class Graph>
void vertex_similarity_block_processing(const Graph &g,
                                        std::int32_t block_row_count,
                                        std::int32_t block_column_count) {
    // create caching builders for all threads
    std::vector<dal::preview::jaccard::caching_builder> processing_blocks(
        tbb::this_task_arena::max_concurrency());

    // compute the number of vertices in graph
    const std::int32_t vertex_count = dal::preview::get_vertex_count(g);

    // compute the number of rows
    std::int32_t row_count = vertex_count / block_row_count;
    if (vertex_count % block_row_count) {
        row_count++;
    }

    // parallel processing by rows
    tbb::parallel_for(
        tbb::blocked_range<std::int32_t>(0, row_count),
        [&](const tbb::blocked_range<std::int32_t> &r) {
            for (std::int32_t i = r.begin(); i != r.end(); ++i) {
                // compute the range of rows
                const std::int32_t row_range_begin = i * block_row_count;
                const std::int32_t row_range_end = (i + 1) * block_row_count;

                // start column ranges from diagonal
                const std::int32_t column_begin = 1 + row_range_begin;

                // compute the number of columns
                std::int32_t column_count = (vertex_count - column_begin) / block_column_count;
                if ((vertex_count - column_begin) % block_column_count) {
                    column_count++;
                }

                // parallel processing by columns
                tbb::parallel_for(
                    tbb::blocked_range<std::int32_t>(0, column_count),
                    [&](const tbb::blocked_range<std::int32_t> &inner_r) {
                        for (std::int32_t j = inner_r.begin(); j != inner_r.end(); ++j) {
                            // compute the range of columns
                            const std::int32_t column_range_begin =
                                column_begin + j * block_column_count;
                            const std::int32_t column_range_end =
                                column_begin + (j + 1) * block_column_count;

                            // set block ranges for the vertex similarity algorithm
                            const auto jaccard_desc =
                                dal::preview::jaccard::descriptor<>().set_block(
                                    { row_range_begin, std::min(row_range_end, vertex_count) },
                                    { column_range_begin,
                                      std::min(column_range_end, vertex_count) });

                            // compute Jaccard coefficients for the block
                            dal::preview::vertex_similarity(
                                jaccard_desc,
                                g,
                                processing_blocks[tbb::this_task_arena::current_thread_index()]);

                            // do application specific postprocessing of the result here
                        }
                    },
                    tbb::simple_partitioner{});
            }
        },
        tbb::simple_partitioner{});
}
```

kmeans_init_dense.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iomanip>
#include <iostream>

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/kmeans.hpp"
#include "oneapi/dal/algo/kmeans_init.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

template <typename Method>
void run(const dal::table& x_train, const std::string& method_name) {
    constexpr std::int64_t cluster_count = 20;
    constexpr std::int64_t max_iteration_count = 1000;
    constexpr double accuracy_threshold = 0.01;

    const auto kmeans_init_desc =
        dal::kmeans_init::descriptor<float, Method>().set_cluster_count(cluster_count);

    const auto result_init = dal::compute(kmeans_init_desc, x_train);

    const auto kmeans_desc = dal::kmeans::descriptor<>()
                                 .set_cluster_count(cluster_count)
                                 .set_max_iteration_count(max_iteration_count)
                                 .set_accuracy_threshold(accuracy_threshold);

    const auto result_train = dal::train(kmeans_desc, x_train, result_init.get_centroids());

    std::cout << "Method: " << method_name << std::endl;
    std::cout << "Max iteration count: " << max_iteration_count
              << ", Accuracy threshold: " << accuracy_threshold << std::endl;
    std::cout << "Iteration count: " << result_train.get_iteration_count()
              << ", Objective function value: " << result_train.get_objective_function_value()
              << '\n'
              << std::endl;
}

int main(int argc, char const* argv[]) {
    const auto train_data_file_name = get_data_path("kmeans_init_dense.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });

    run<dal::kmeans_init::method::dense>(x_train, "dense");
    run<dal::kmeans_init::method::random_dense>(x_train, "random_dense");
    run<dal::kmeans_init::method::plus_plus_dense>(x_train, "plus_plus_dense");
    run<dal::kmeans_init::method::parallel_plus_dense>(x_train, "parallel_plus_dense");

    return 0;
}
```

kmeans_lloyd_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/kmeans.hpp"
```

```
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("kmeans_dense_train_data.csv");
    const auto initial_centroids_file_name = get_data_path("kmeans_dense_train_centroids.csv");
    const auto test_data_file_name = get_data_path("kmeans_dense_test_data.csv");
    const auto test_response_file_name = get_data_path("kmeans_dense_test_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto initial_centroids =
        dal::read<dal::table>(dal::csv::data_source{ initial_centroids_file_name });

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto kmeans_desc = dal::kmeans::descriptor<>()
                                 .set_cluster_count(20)
                                 .set_max_iteration_count(5)
                                 .set_accuracy_threshold(0.001);

    const auto result_train = dal::train(kmeans_desc, x_train, initial_centroids);

    std::cout << "Iteration count: " << result_train.get_iteration_count() << std::endl;
    std::cout << "Objective function value: " << result_train.get_objective_function_value()
              << std::endl;
    std::cout << "Responses:\n" << result_train.get_responses() << std::endl;
    std::cout << "Centroids:\n" << result_train.get_model().get_centroids() << std::endl;

    const auto result_test = dal::infer(kmeans_desc, result_train.get_model(), x_test);

    std::cout << "Infer result:\n" << result_test.get_responses() << std::endl;

    std::cout << "Ground truth:\n" << y_test << std::endl;

    return 0;
}
```

knn_cls_brute_force_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/
```

```
#include <iomanip>
#include <iostream>

#include "oneapi/dal/algo/knn.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("k_nearest_neighbors_train_data.csv");
    const auto train_response_file_name = get_data_path("k_nearest_neighbors_train_label.csv");
    const auto test_data_file_name = get_data_path("k_nearest_neighbors_test_data.csv");
    const auto test_response_file_name = get_data_path("k_nearest_neighbors_test_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto knn_desc = dal::knn::descriptor(5, 1);

    const auto train_result = dal::train(knn_desc, x_train, y_train);

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto test_result = dal::infer(knn_desc, x_test, train_result.get_model());

    std::cout << "Test results:\n" << test_result.get_responses() << std::endl;
    std::cout << "True responses:\n" << y_true << std::endl;

    return 0;
}
```

knn_cls_kd_tree_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iomanip>
#include <iostream>

#include "oneapi/dal/algo/knn.hpp"
```

```
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("k_nearest_neighbors_train_data.csv");
    const auto train_response_file_name = get_data_path("k_nearest_neighbors_train_label.csv");
    const auto test_data_file_name = get_data_path("k_nearest_neighbors_test_data.csv");
    const auto test_response_file_name = get_data_path("k_nearest_neighbors_test_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto knn_desc =
        dal::knn::descriptor<float, dal::knn::method::kd_tree, dal::knn::task::classification>(5,
                                                                                           1);

    const auto train_result = dal::train(knn_desc, x_train, y_train);

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto test_result = dal::infer(knn_desc, x_test, train_result.get_model());

    std::cout << "Test results:\n" << test_result.get_responses() << std::endl;
    std::cout << "True responses:\n" << y_true << std::endl;

    return 0;
}
```

knn_search_brute_force_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/knn.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace knn = dal::knn;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("k_nearest_neighbors_train_data.csv");
    const auto query_data_file_name = get_data_path("k_nearest_neighbors_test_data.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto x_query = dal::read<dal::table>(dal::csv::data_source{ query_data_file_name });

    using cosine_desc_t = dal::cosine_distance::descriptor<float>;
    const auto cosine_desc = cosine_desc_t{};

    const std::size_t neighbors_count = 6;
    const auto knn_desc =
        knn::descriptor<float, knn::method::brute_force, knn::task::search, cosine_desc_t>(
            neighbors_count,
            cosine_desc);

    const auto train_result = dal::train(knn_desc, x_train);
    const auto test_result = dal::infer(knn_desc, x_query, train_result.get_model());

    std::cout << "Indices result:\n" << test_result.get_indices() << std::endl;
    std::cout << "Distance result:\n" << test_result.get_distances() << std::endl;
    return 0;
}
```

linear_kernel_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/linear_kernel.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto data_file_name = get_data_path("kernel_function.csv");

    const auto x = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto y = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);
    const auto result = dal::compute(kernel_desc, x, y);

    std::cout << "Values:\n" << result.get_values() << std::endl;

    return 0;
}
```

linear_regression_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/linear_regression.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "oneapi/dal/exceptions.hpp"
#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

void run() {
    const auto train_data_file_name = get_data_path("linear_regression_train_data.csv");
    const auto train_response_file_name = get_data_path("linear_regression_train_responses.csv");
    const auto test_data_file_name = get_data_path("linear_regression_test_data.csv");
    const auto test_response_file_name = get_data_path("linear_regression_test_responses.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto lr_desc = dal::linear_regression::descriptor<>();

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_test = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto train_result = dal::train(lr_desc, x_train, y_train);
    const auto lr_model = train_result.get_model();

    const auto test_result_uniform = dal::infer(lr_desc, x_test, lr_model);

    std::cout << "Test results:\n" << test_result_uniform.get_responses() << std::endl;
    std::cout << "True responses:\n" << y_test << std::endl;
}
```

```
int main(int argc, char const* argv[]) {
    run();
    return 0;
}
```

louvain_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <memory>

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/louvain.hpp"
#include "oneapi/dal/graph/undirected_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

int main(int argc, char** argv) {
    const auto filename = get_data_path("weighted_edge_list.csv");

    using vertex_type = int32_t;
    using weight_type = double;
    using graph_t = dal::preview::undirected_adjacency_vector_graph<vertex_type, weight_type>;
    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename },
                                          dal::preview::read_mode::weighted_edge_list);

    // set algorithm parameters
    const auto louvain_desc = dal::preview::louvain::descriptor<>()
                                  .set_resolution(1)
                                  .set_accuracy_threshold(0.0001)
                                  .set_max_iteration_count(3);
    // compute louvain
    const std::int64_t row_count = 7;
    const std::int64_t col_count = 1;
    const std::int64_t data[] = { 0, 1, 2, 3, 4, 5, 6 };
    const auto initial_labels = dal::homogen_table::wrap(data, row_count, col_count);

    const auto result = dal::preview::vertex_partitioning(louvain_desc, graph, initial_labels);

    std::cout << "Modularity: " << result.get_modularity() << std::endl;
    std::cout << "Number of communities: " << result.get_community_count() << std::endl;
    std::cout << "Labels of communities:" << std::endl << result.get_labels() << std::endl;

    return 0;
}
```

pca_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/pca.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

template <typename Method>
void run(const dal::table& x_train, const std::string& method_name) {
    const auto pca_desc =
        dal::pca::descriptor<float, Method>().set_component_count(5).set_deterministic(true);

    const auto result_train = dal::train(pca_desc, x_train);

    std::cout << method_name << "\n" << std::endl;

    std::cout << "Eigenvectors:\n" << result_train.get_eigenvectors() << std::endl;

    std::cout << "Eigenvalues:\n" << result_train.get_eigenvalues() << std::endl;

    const auto result_infer = dal::infer(pca_desc, result_train.get_model(), x_train);

    std::cout << "Transformed data:\n" << result_infer.get_transformed_data() << std::endl;
}

int main(int argc, char const* argv[]) {
    const auto train_data_file_name = get_data_path("pca_normalized.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });

    run<dal::pca::method::cov>(x_train, "Training method: Covariance");
    run<dal::pca::method::svd>(x_train, "Training method: SVD");
    return 0;
}
```

pca_precomputed_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2022 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/pca.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

template <typename Method>
void run(const dal::table& x_train, const std::string& method_name) {
    const auto pca_desc =
        dal::pca::descriptor<float, Method>().set_component_count(5).set_deterministic(true);

    const auto result_train = dal::train(pca_desc, x_train);

    std::cout << method_name << "\n" << std::endl;

    std::cout << "Eigenvectors:\n" << result_train.get_eigenvectors() << std::endl;

    std::cout << "Eigenvalues:\n" << result_train.get_eigenvalues() << std::endl;

    const auto result_infer = dal::infer(pca_desc, result_train.get_model(), x_train);

    std::cout << "Transformed data:\n" << result_infer.get_transformed_data() << std::endl;
}

int main(int argc, char const* argv[]) {
    const auto cov_data_file_name = get_data_path("precomputed_covariance.csv");
    const auto cor_data_file_name = get_data_path("precomputed_correlation.csv");

    const auto cov_train = dal::read<dal::table>(dal::csv::data_source{ cov_data_file_name });
    const auto cor_train = dal::read<dal::table>(dal::csv::data_source{ cor_data_file_name });

    run<dal::pca::method::precomputed>(cov_train, "PCA precomputed method with covariance
matrix");
    run<dal::pca::method::precomputed>(cor_train, "PCA precomputed method with correlation
matrix");

    return 0;
}
```

polynomial_kernel_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/polynomial_kernel.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto data_file_name = get_data_path("kernel_function.csv");

    const auto x = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto y = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto kernel_desc =
        dal::polynomial_kernel::descriptor{}.set_scale(1.0).set_shift(0.0).set_degree(2);

    const auto result = dal::compute(kernel_desc, x, y);

    std::cout << "Values:\n" << result.get_values() << std::endl;

    return 0;
}
```

rbf_kernel_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
```

```
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/rbf_kernel.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto data_file_name = get_data_path("kernel_function.csv");

    const auto x = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto y = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto kernel_desc = dal::rbf_kernel::descriptor{}.set_sigma(1.0);

    const auto result = dal::compute(kernel_desc, x, y);

    std::cout << "Values:\n" << result.get_values() << std::endl;

    return 0;
}
```

shortest_paths_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <memory>

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/shortest_paths.hpp"
#include "oneapi/dal/graph/directed_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;

int main(int argc, char** argv) {
    const auto filename = get_data_path("weighted_edge_list.csv");
    using vertex_type = int32_t;
    using weight_type = double;
    using graph_t = dal::preview::directed_adjacency_vector_graph<vertex_type, weight_type>;

    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename },
                                          dal::preview::read_mode::weighted_edge_list);

    // set algorithm parameters
    const auto shortest_paths_desc = dal::preview::shortest_paths::descriptor<
        float,
        dal::preview::shortest_paths::method::delta_stepping,
        dal::preview::shortest_paths::task::one_to_all>(
        0,
        0.85,
        dal::preview::shortest_paths::optional_results::distances |
            dal::preview::shortest_paths::optional_results::predecessors);
    // compute shortest paths
    const auto result_shortest_paths = dal::preview::traverse(shortest_paths_desc, graph);

    // extract the result
    std::cout << "Distances: " << std::endl;
    std::cout << result_shortest_paths.get_distances() << std::endl;
    std::cout << "Predecessors: " << std::endl;
    std::cout << result_shortest_paths.get_predecessors() << std::endl;

    return 0;
}
```

sigmoid_kernel_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/sigmoid_kernel.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto data_file_name = get_data_path("kernel_function.csv");
    const auto x = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto y = dal::read<dal::table>(dal::csv::data_source{ data_file_name });
    const auto kernel_desc = dal::sigmoid_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);

    const auto result = dal::compute(kernel_desc, x, y);

    std::cout << "Values:\n" << result.get_values() << std::endl;

    return 0;
}
```

subgraph_isomorphism_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <iostream>

#include "example_util/utils.hpp"
#include "oneapi/dal/algo/subgraph_isomorphism.hpp"
#include "oneapi/dal/exceptions.hpp"
#include "oneapi/dal/graph/undirected_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"
#include "oneapi/dal/table/common.hpp"

namespace dal = oneapi::dal;

int main(int argc, char **argv) {
    auto target_filename = get_data_path("si_target_graph.csv");
    auto pattern_filename = get_data_path("si_pattern_graph.csv");

    using graph_t = dal::preview::undirected_adjacency_vector_graph<>;

    const auto target_graph = dal::read<graph_t>(dal::csv::data_source{ target_filename });
    const auto pattern_graph = dal::read<graph_t>(dal::csv::data_source{ pattern_filename });

    // set algorithm parameters
    const auto subgraph_isomorphism_desc =
        dal::preview::subgraph_isomorphism::descriptor<>()
            .set_kind(dal::preview::subgraph_isomorphism::kind::non_induced)
            .set_semantic_match(false)
            .set_max_match_count(10);

    const auto result =
        dal::preview::graph_matching(subgraph_isomorphism_desc, target_graph, pattern_graph);

    // extract the result
    std::cout << "Number of matchings: " << result.get_match_count() << std::endl;
    std::cout << "Matchings:" << std::endl << result.get_vertex_match() << std::endl;

    return 0;
}
```

svm_multi_class_thunder_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("svm_multi_class_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_multi_class_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_multi_class_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_multi_class_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);
    const auto svm_desc = dal::svm::descriptor{ kernel_desc }.set_class_count(5).set_c(1.0);
    const auto result_train = dal::train(svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Coeffs indices:\n" << result_train.get_coeffs() << std::endl;

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_test = dal::infer(svm_desc, result_train.get_model(), x_test);

    std::cout << "Decision function result:\n" << result_test.get_decision_function() <<
std::endl;
    std::cout << "Responses result:\n" << result_test.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;

    return 0;
}
```

svm_nu_cls_thunder_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace svm = dal::svm;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("svm_two_class_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_two_class_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_two_class_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_two_class_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);

    const auto svm_desc =
        svm::descriptor<float, svm::method::thunder, svm::task::nu_classification>{ kernel_desc }
            .set_nu(0.5)
            .set_accuracy_threshold(0.001)
            .set_max_iteration_count(100)
            .set_cache_size(200.0)
            .set_tau(1e-6);

    const auto result_train = dal::train(svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Support indices:\n" << result_train.get_support_indices() << std::endl;
    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_infer = dal::infer(svm_desc, result_train.get_model(), x_test);

    std::cout << "Decision function result:\n" << result_infer.get_decision_function() <<
std::endl;
    std::cout << "Responses result:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;

    return 0;
}
```

svm_nu_reg_thunder_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace svm = dal::svm;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("svm_reg_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_reg_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_reg_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_reg_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);

    const auto svm_desc =
        svm::descriptor<float, svm::method::thunder, svm::task::nu_regression>{ kernel_desc }
            .set_nu(0.5)
            .set_c(100.0)
            .set_accuracy_threshold(0.001)
            .set_cache_size(200.0)
            .set_tau(1e-6);

    const auto result_train = dal::train(svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Support indices:\n" << result_train.get_support_indices() << std::endl;

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_infer = dal::infer(svm_desc, result_train.get_model(), x_test);

    std::cout << "Responses result:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;

    return 0;
}
```

svm_reg_thunder_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2021 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace svm = dal::svm;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("svm_reg_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_reg_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_reg_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_reg_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);
    const auto svm_desc =
        svm::descriptor<float, svm::method::thunder, svm::task::regression>{ kernel_desc }
            .set_c(100.0)
            .set_epsilon(0.3)
            .set_accuracy_threshold(0.001)
            .set_cache_size(200.0)
            .set_tau(1e-6);

    const auto result_train = dal::train(svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Support indices:\n" << result_train.get_support_indices() << std::endl;

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_infer = dal::infer(svm_desc, result_train.get_model(), x_test);

    std::cout << "Responses result:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;

    return 0;
}
```

svm_two_class_smo_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace svm = dal::svm;

int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("svm_two_class_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_two_class_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_two_class_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_two_class_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);

    const auto svm_desc =
        svm::descriptor<float, svm::method::smo, svm::task::classification>{ kernel_desc }
            .set_c(1.0)
            .set_accuracy_threshold(0.001)
            .set_max_iteration_count(1000)
            .set_cache_size(200.0)
            .set_shrinking(true)
            .set_tau(1e-6);

    const auto result_train = dal::train(svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Support indices:\n" << result_train.get_support_indices() << std::endl;

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_infer = dal::infer(svm_desc, result_train.get_model(), x_test);

    std::cout << "Decision function result:\n" << result_infer.get_decision_function() <<
std::endl;
    std::cout << "Responses result:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;

    return 0;
}
```

svm_two_class_thunder_dense_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include "oneapi/dal/algo/svm.hpp"
#include "oneapi/dal/io/csv.hpp"

#include "example_util/utils.hpp"

namespace dal = oneapi::dal;
namespace svm = dal::svm;
```

```
int main(int argc, char const *argv[]) {
    const auto train_data_file_name = get_data_path("svm_two_class_train_dense_data.csv");
    const auto train_response_file_name = get_data_path("svm_two_class_train_dense_label.csv");
    const auto test_data_file_name = get_data_path("svm_two_class_test_dense_data.csv");
    const auto test_response_file_name = get_data_path("svm_two_class_test_dense_label.csv");

    const auto x_train = dal::read<dal::table>(dal::csv::data_source{ train_data_file_name });
    const auto y_train =
dal::read<dal::table>(dal::csv::data_source{ train_response_file_name });

    const auto kernel_desc = dal::linear_kernel::descriptor{}.set_scale(1.0).set_shift(0.0);

    const auto svm_desc = svm::descriptor{ kernel_desc }
                              .set_c(1.0)
                              .set_accuracy_threshold(0.001)
                              .set_max_iteration_count(100)
                              .set_cache_size(200.0)
                              .set_tau(1e-6);

    const auto result_train = dal::train(svm_desc, x_train, y_train);

    std::cout << "Biases:\n" << result_train.get_biases() << std::endl;
    std::cout << "Support indices:\n" << result_train.get_support_indices() << std::endl;

    const auto x_test = dal::read<dal::table>(dal::csv::data_source{ test_data_file_name });
    const auto y_true = dal::read<dal::table>(dal::csv::data_source{ test_response_file_name });

    const auto result_infer = dal::infer(svm_desc, result_train.get_model(), x_test);

    std::cout << "Decision function result:\n" << result_infer.get_decision_function() <<
std::endl;
    std::cout << "Responses result:\n" << result_infer.get_responses() << std::endl;
    std::cout << "Responses true:\n" << y_true << std::endl;

    return 0;
}
```

triangle_counting_batch.cpp

```
/*******************************************************************************
* Copyright 2020 Intel Corporation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*******************************************************************************/

#include <memory>

#include "example_util/utils.hpp"
```

```
#include "oneapi/dal/algo/triangle_counting.hpp"
#include "oneapi/dal/graph/undirected_adjacency_vector_graph.hpp"
#include "oneapi/dal/io/csv.hpp"

namespace dal = oneapi::dal;
using namespace dal::preview::triangle_counting;

int main(int argc, char** argv) {
    const auto filename = get_data_path("graph.csv");

    // read the graph
    using graph_t = dal::preview::undirected_adjacency_vector_graph<>;
    const auto graph = dal::read<graph_t>(dal::csv::data_source{ filename });
    // set algorithm parameters
    const auto tc_desc = descriptor<float, method::ordered_count, task::local_and_global>();

    // compute local and global triangles
    const auto result_vertex_ranking = dal::preview::vertex_ranking(tc_desc, graph);

    // extract the result
    std::cout << "Global triangles: " << result_vertex_ranking.get_global_rank() << std::endl;
    std::cout << "Local triangles: " << std::endl;

    auto local_triangles_table = result_vertex_ranking.get_ranks();
    const auto& local_triangles = static_cast<const dal::homogen_table&>(local_triangles_table);
    const auto local_triangles_data = local_triangles.get_data<std::int64_t>();
    for (auto i = 0; i < local_triangles_table.get_row_count(); i++) {
        std::cout << i << ":\t" << local_triangles_data[i] << std::endl;
    }

    return 0;
}
```

Appendix

- Decision Tree
- k-d Tree

Decision Tree

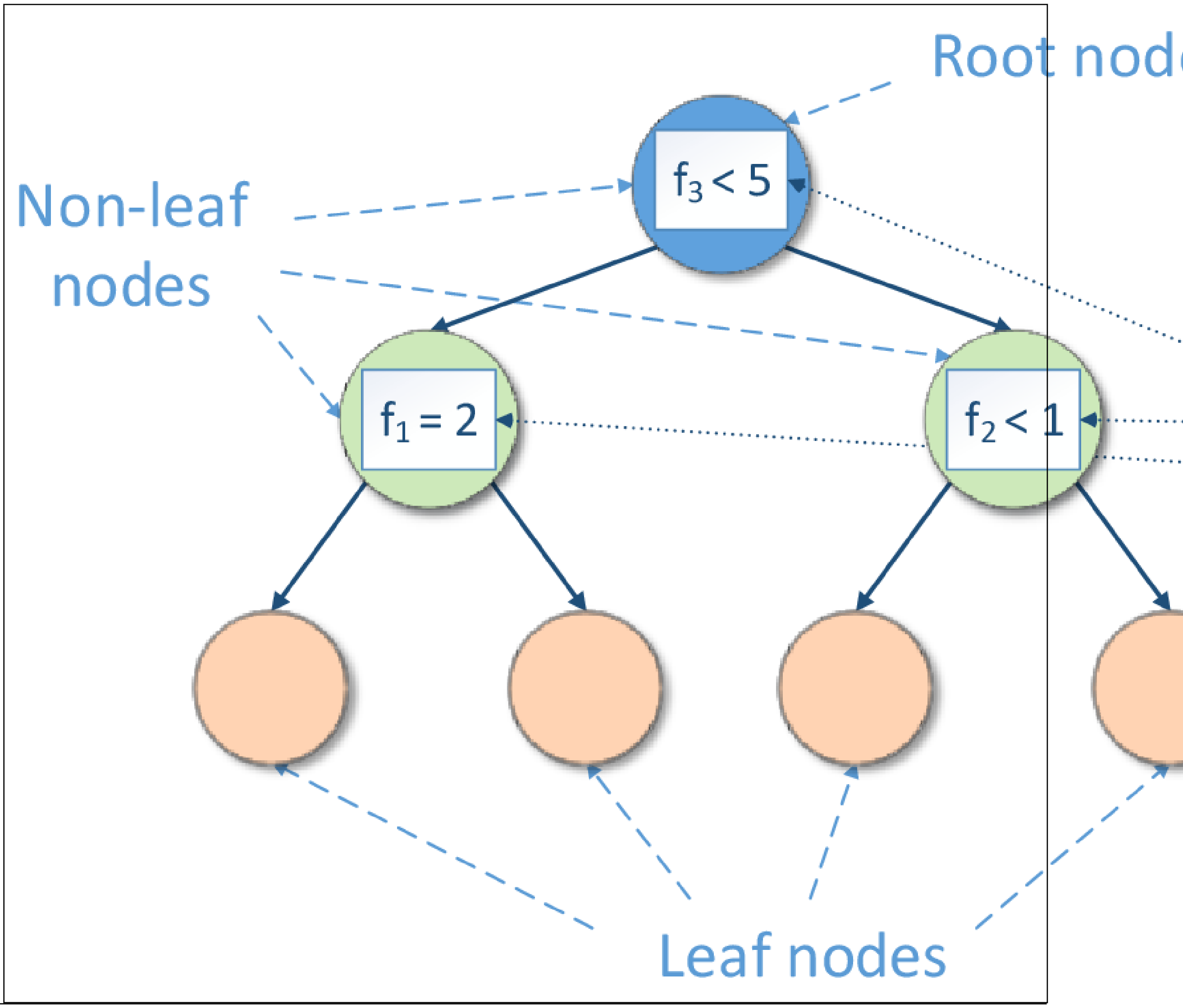
Decision trees partition the feature space into a set of hypercubes, and then fit a simple model in each hypercube. The simple model can be a prediction model, which ignores all predictors and predicts the majority (most frequent) class (or the mean of a dependent variable for regression), also known as 0-R or constant classifier.

Decision tree induction forms a tree-like graph structure as shown in the figure below, where:

- Each internal (non-leaf) node denotes a test on one of the features
- Each branch descending from a non-leaf node corresponds to an outcome of the test

- Each external node (leaf) denotes the mentioned simple model

Decision Tree Structure



A test is a rule for partitioning the feature space. A test depends on feature values. Each outcome of a test represents an appropriate hypercube associated with both the test and one of the descending branches.

If a test is a Boolean expression (for example, $f < c$ or $f = c$, where *f* is a feature and *c* is a constant fitted during decision tree induction), the inducted decision tree is a binary tree, so its non-leaf nodes have exactly two branches, 'true' and 'false', each corresponding to the result of the Boolean expression.

Prediction is performed by starting at the root node of the tree, testing features by the test specified in this node, then moving down the tree branch corresponding to the outcome of the test for the given sample. This process is then repeated for the subtree rooted at the node, discovered at the selected branch. The final result is the prediction of the simple model at the leaf node.

Decision trees are often used in ensemble algorithms, such as boosting, bagging, or decision forest.

k-d Tree

k-d tree is a space-partitioning binary tree [Bentley80], where

- Each non-leaf node induces the hyperplane that splits the feature space into two parts. To define the splitting hyperplane explicitly, a non-leaf node stores the identifier of the feature (that defines axis in the feature space) and a cut-point
- Each leaf node of the tree has an associated subset (*a bucket*) of elements of the training data set. Feature vectors from a bucket belong to the region of the space defined by tree nodes on the path from the root node to the respective leaf.

Related terms

A cut-point	A feature value that corresponds to a non-leaf node of a *k-d* tree and defines the splitting hyperplane orthogonal to the axis specified by the given feature.

DAAL Interfaces

This chapter documents algorithms implemented in DAAL interfaces. See oneAPI Interfaces to find documentation on oneAPI interfaces. Refer to oneAPI vs. DAAL Interfaces to learn the difference between them.

- CPU and GPU Support
  - Computation modes
  - Methods
  - Parameters
- Library Usage
  - Algorithms
  - Computation Modes
  - Training and Prediction
- Data Management
- Analysis
  - K-Means Clustering
  - Density-Based Spatial Clustering of Applications with Noise
  - Correlation and Variance-Covariance Matrices
  - Principal Component Analysis
  - Principal Components Analysis Transform
  - Singular Value Decomposition
  - Association Rules
  - Kernel Functions
  - Expectation-Maximization
  - Cholesky Decomposition
  - QR Decomposition
  - Outlier Detection
  - Distance Matrix
  - Distributions
  - Engines
  - Moments of Low Order
  - Quantile
  - Quality Metrics
  - Sorting
  - Normalization
  - Optimization Solvers
- Training and Prediction
  - Decision Forest
  - Decision Trees
  - Gradient Boosted Trees
  - Stump
  - Linear and Ridge Regressions

    - LASSO and Elastic Net Regressions
    - k-Nearest Neighbors (kNN) Classifier
    - Implicit Alternating Least Squares
    - Logistic Regression
    - Naïve Bayes Classifier
    - Support Vector Machine Classifier
    - Multi-class Classifier
    - Boosting
    - Training Alternative
- Services
    - Extracting Version Information
    - Handling Errors
    - Managing Memory
    - Managing the Computational Environment
    - Providing a Callback for the Host Application

Examples

You can find examples on Github*:

- C++ (CPU)
- Java* (not supported on GPU)
- Python*

CPU and GPU Support

Not all computation modes, methods, and parameters are supported on both CPU and GPU. Differences in CPU and GPU support are listed below.

Computation modes

For the following algorithms, only listed computation modes are supported on GPU:

GPU Support: Computaion Modes

Algorithm	Supported on GPU
Density-Based Spatial Clustering of Applications with Noise	batch
Linear Regression	batch, online
Logistic Regression	batch, online

Methods

For the following algorithms, only listed methods are supported on GPU:

GPU Support: Methods

Algortihm	Supported on GPU
K-Means Clustering	`defaultDense`
Initialization	`defaultDense`, `randomDense`
Linear Regression	`defaultDense`
Moments of Low Order	`defaultDense`

Algortihm	Supported on GPU
Stochastic Gradient Descent Algorithm	`miniBatch`
Covariance	`defaultDense`
Principal Component Analysis	`defaultDense`
k-Nearest Neighbors (kNN) Classifier	Brute Force
Support Vector Machine Classifier	`thunder`
Decision Forest	`hist`

Parameters

GPU Support: Algorithm Parameters

Algortihm	Notes
Support Vector Machine Classifier	`doShrinking` is only supported for `defaultDense` method.
Density-Based Spatial Clustering of Applications with Noise	• On GPU, the `memorySavingMode` flag can only be set to `true`. • On GPU, the `weights` parameter is not supported.
Kernel Functions	On GPU, the only supported computation mode (`ComputationMode`) is `matrixMatrix`.
Objective Function	• On GPU, only Logistic Loss and Cross-entropy Loss are supported, Mean Squared Error Algorithm is not supported. • On GPU, `resultsToCompute` only computes `value`, `gradient`, and `hessian`.
Logistic Regression	`penaltyL1` is not supported on GPU

Library Usage

- Algorithms
  - Algorithm Input
  - Algorithm Output
  - Algorithm Parameters
- Computation Modes
  - Batch processing
  - Online processing
  - Distributed processing
- Training and Prediction
  - Classification Usage Model
  - Regression Usage Model
  - Recommendation Systems Usage Model

Algorithms

All Algorithms classes are derived from the base class AlgorithmIface. It provides interfaces for computations covering a variety of usage scenarios. Basic methods that you typically call are compute() and finalizeCompute(). In a very generic form algorithms accept one or several numeric tables or models as an input and return one or several numeric tables and models as an output. Algorithms may also require algorithm-specific parameters that you can modify by accessing the parameter field of the algorithm. Because most of algorithm parameters are preset with default values, you can often omit initialization of the parameter.

Algorithm Input

An algorithm can accept one or several numeric tables or models as an input. In computation modes that permit multiple calls to the compute() method, ensure that the structure of the input data, that is, the number of features, their order, and type, is the same for all the calls. The following methods are available to provide input to an algorithm:

Algorithm Input

`input.set( Input ID, InputData)`	Use to set a pointer to the input argument with the Input ID identifier. This method overwrites the previous input pointer stored in the algorithm.
`input.add( Input ID, InputData)`	Use in the distributed computation mode to add the pointers with the Input ID identifier. Unlike the input.set() method, input.add() does not overwrite the previously set input pointers, but stores all the input pointers until the compute() method is called.
`input.get( Input ID)`	Use to get a reference to the pointer to the input data with the Input ID identifier.

For the input that each specific algorithm accepts, refer to the description of this algorithm.

Algorithm Output

Output of an algorithm can be one or several models or numeric tables. To retrieve the results of the algorithm computation, call the getResult() method. To access specific results, use the get(Result ID) method with the appropriate Result ID identifier. In the distributed processing mode, to get access to partial results of the algorithm computation, call the getPartialResult() method on each computation node. For a full list of algorithm computation results available, refer to the description of an appropriate algorithm.

By default, all algorithms allocate required memory to store partial and final results. Follow these steps to provide user allocated memory for partial or final results to the algorithm:

1. Create an object of an appropriate class for the results. For the classes supported, refer to the description of a specific algorithm.
2. Provide a pointer to that object to the algorithm by calling the setPartialResult() or setResult() method as appropriate.
3. Call the compute() method. After the call, the object created contains partial or final results.

Algorithm Parameters

Most of algorithms in oneDAL have a set of algorithm-specific parameters. Because most of the parameters are optional and preset with default values, you can often omit parameter modification. Provide required parameters to the algorithm using the constructor during algorithm initialization. If you need to change the parameters, you can do it by accessing the public field parameter of the algorithm. Some algorithms have an initialization procedure that sets or precomputes specific parameters needed to compute the algorithm. You can use the InitializationProcedureIface interface class to implement your own initialization procedure when the default implementation does not meet your specific needs.

Each algorithm also has generic parameters, such as the floating-point type, computation method, and computation step for the distributed processing mode.

- In C++, these parameters are defined as template parameters, and in most cases they are preset with default values. You can change the template parameters while declaring the algorithm.
- In Java, the generic parameters have no default values, and you need to define them in the constructor during algorithm initialization.

For a list of algorithm parameters, refer to the description of an appropriate algorithm.

Computation Modes

The library algorithms support the following computation modes:

- Batch processing
- Online processing
- Distributed processing

You can select the computation mode during initialization of the Algorithm.

For a list of computation parameters of a specific algorithm in each computation mode, possible input types, and output results, refer to the description of an appropriate algorithm.

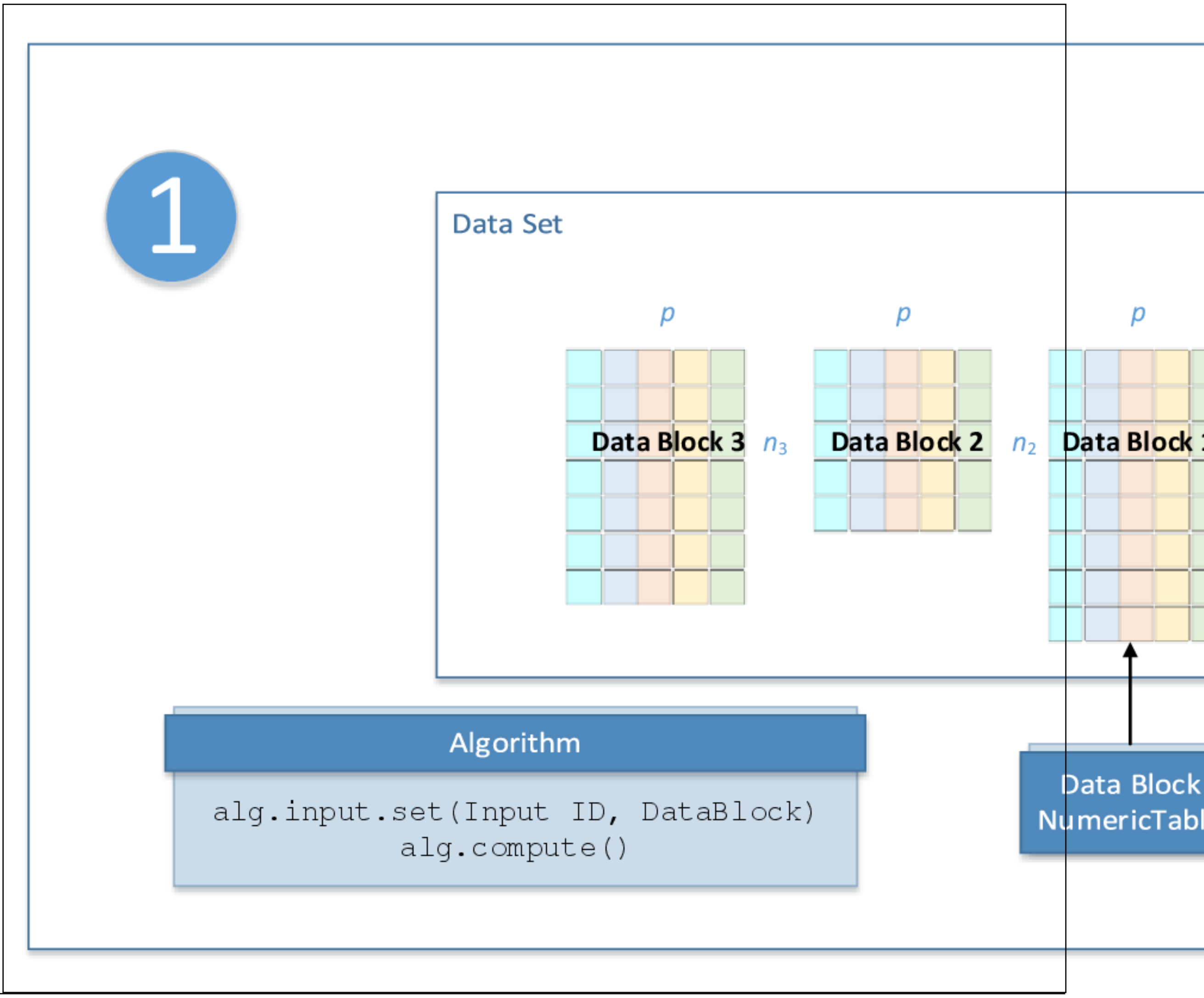
Batch processing

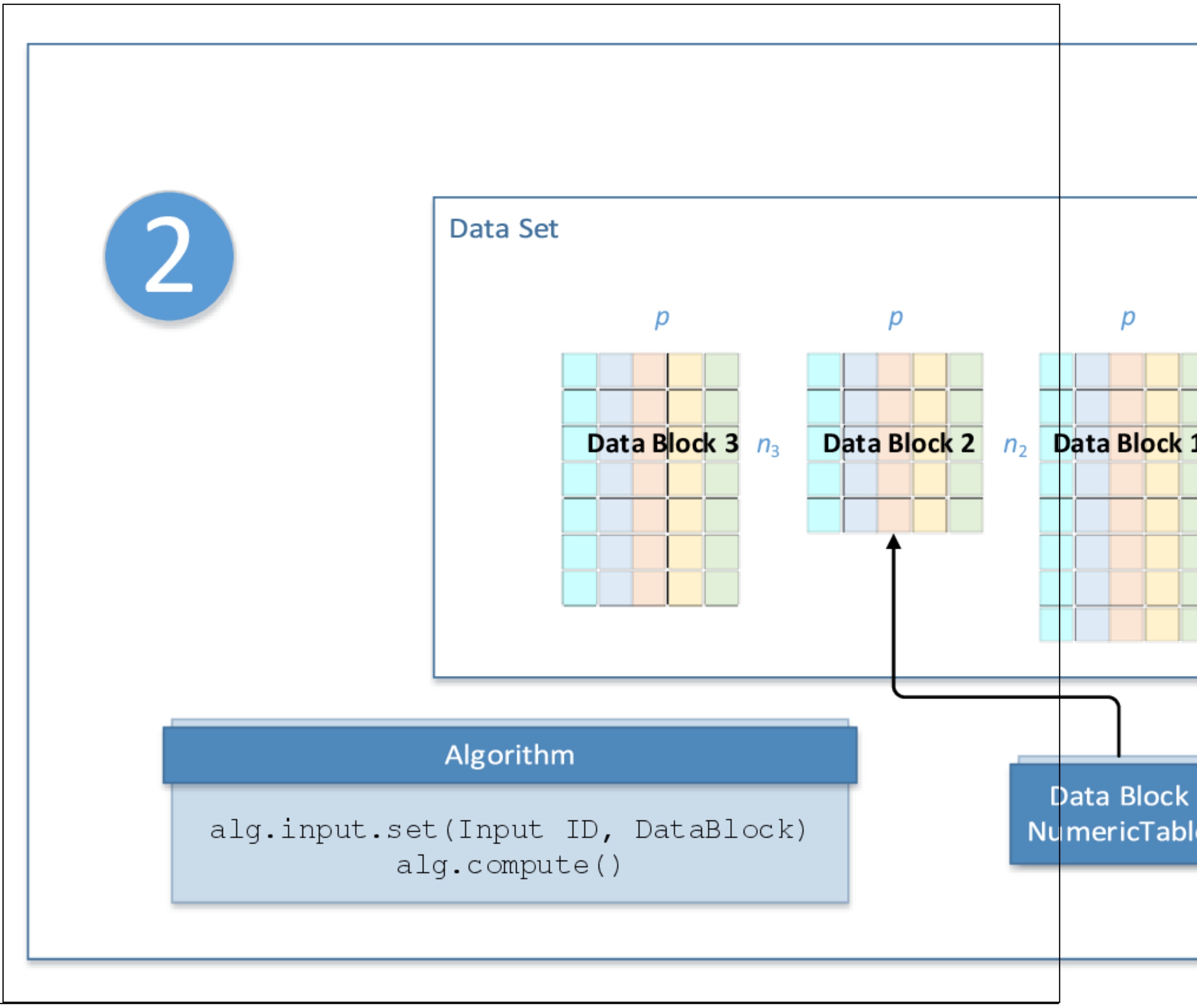
All oneDAL algorithms support at least the batch processing computation mode. In the batch processing mode, the only compute method of a particular algorithm class is used.

Online processing

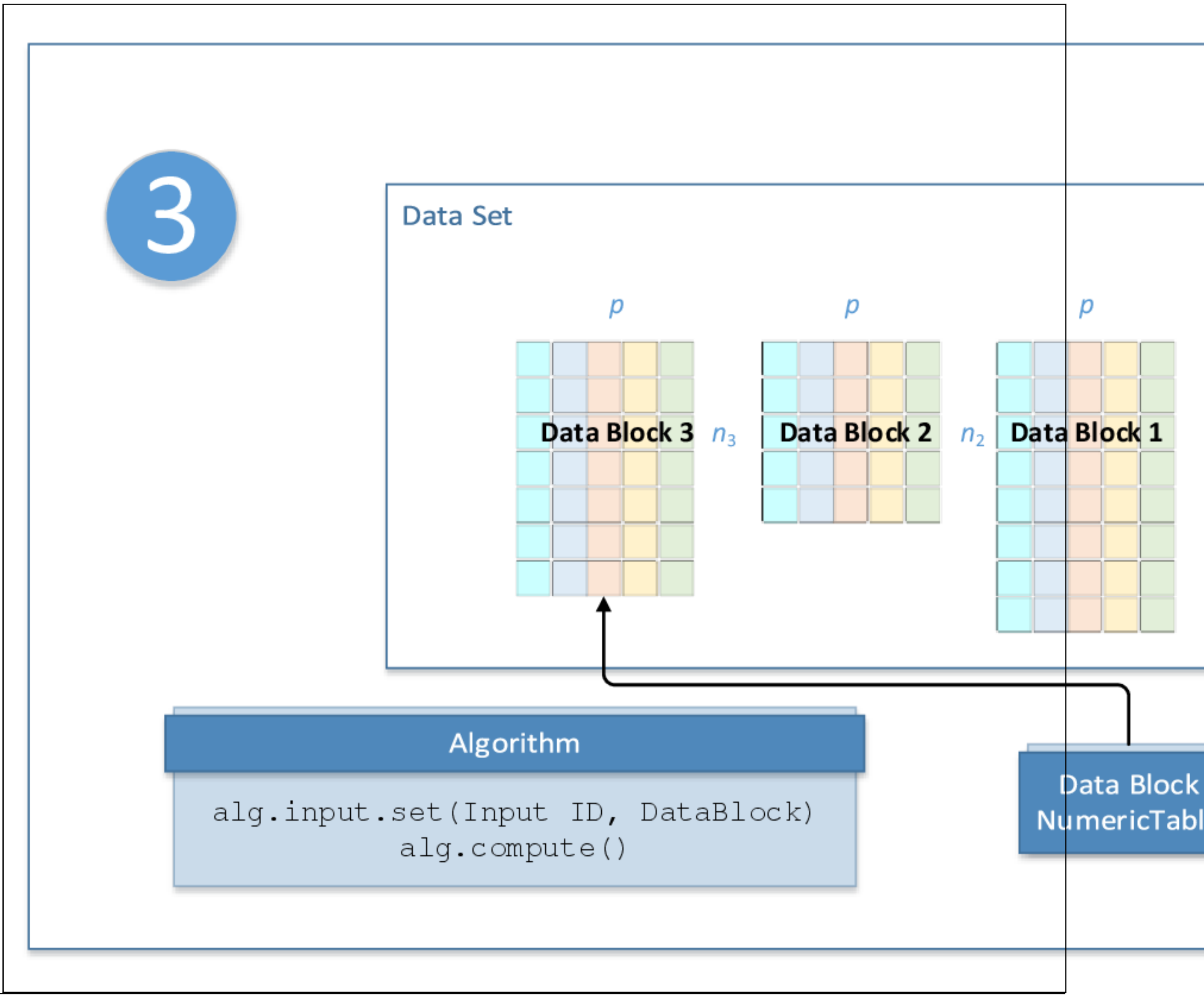
Some oneDAL algorithms enable processing of data sets in blocks. In the online processing mode, the `compute()`, and `finalizeCompute()` methods of a particular algorithm class are used. This computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \ldots \mathrm{nblocks}$. Call the `compute()` method each time a new input becomes available. When the last block of data arrives, call the `finalizeCompute()` method to produce final results. If the input data arrives in an asynchronous mode, you can use the `getStatus()` method for a given data source to check whether a new block of data is available for loading.

The following diagram illustrates the computation schema for online processing:

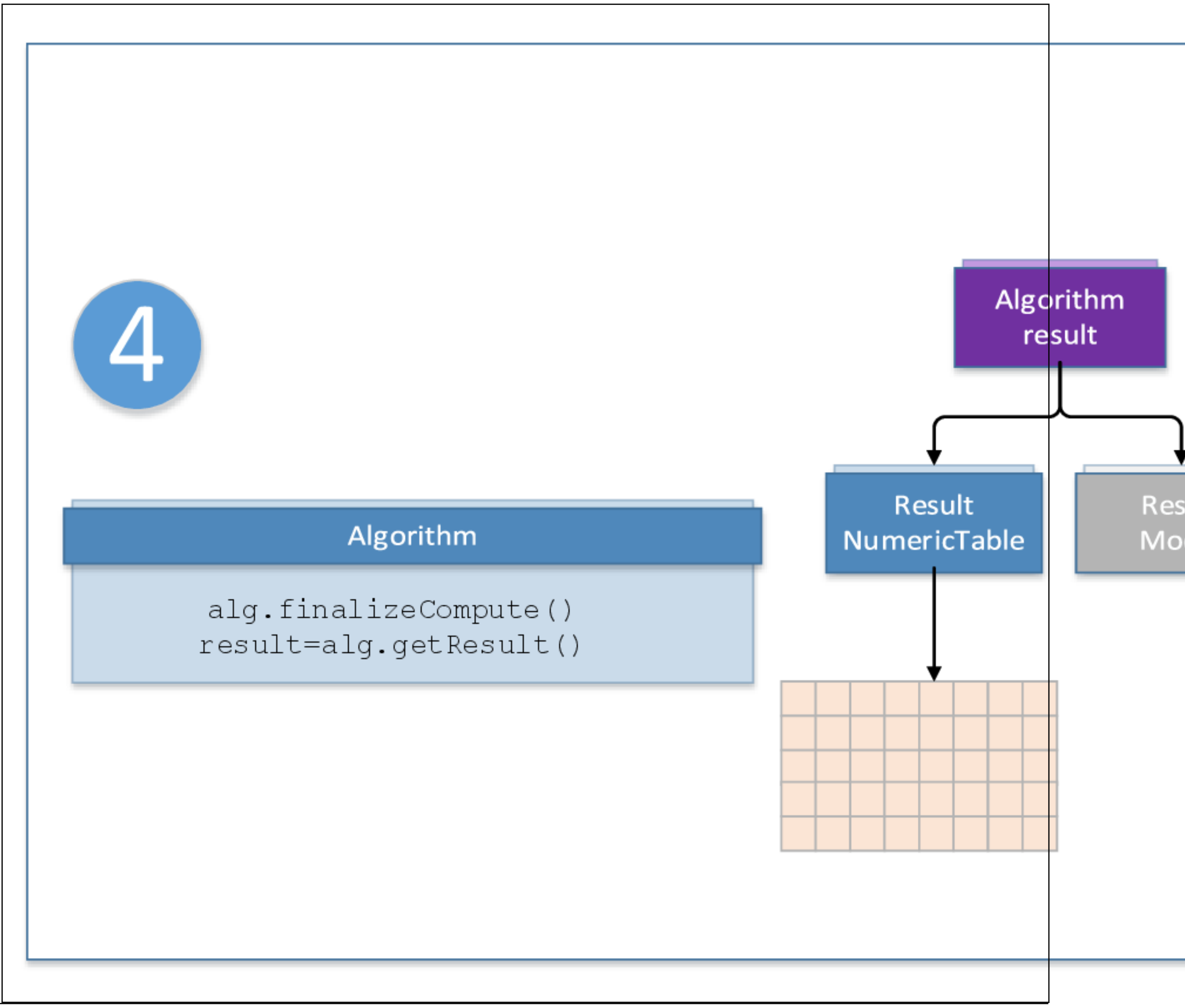

1
Data Set
p
p
p
Data Block 3
n_3
Data Block 2
n_2
Data Block
Algorithm
alg.input.set(Input ID, DataBlock)
alg.compute()
Data Block
NumericTabl



NOTE While different data blocks may have different numbers of observations n_i, they must have the same number of feature vectors *p*.



```
alg.input.set(Input ID, DataBlock)
alg.compute()
```

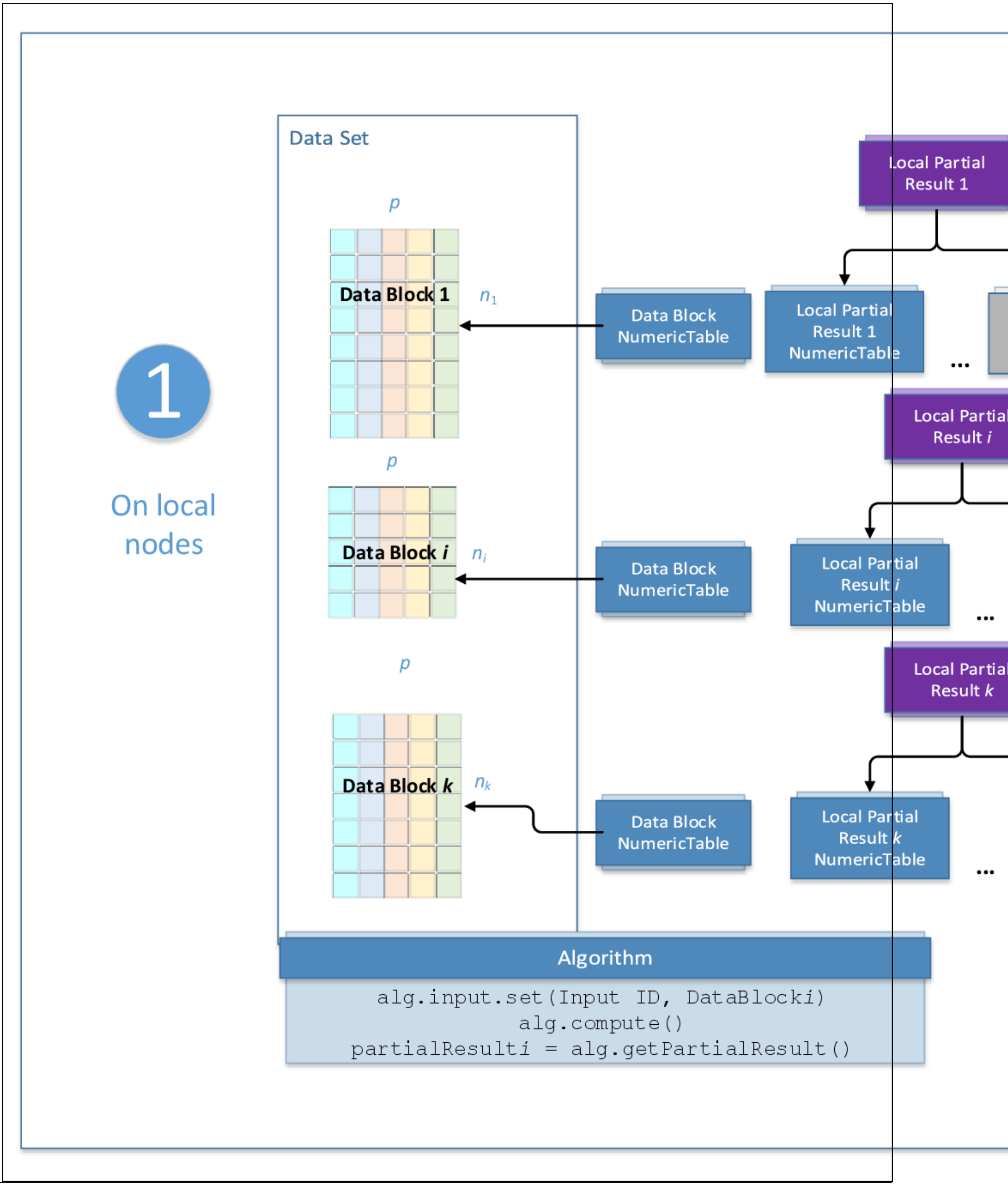


Distributed processing

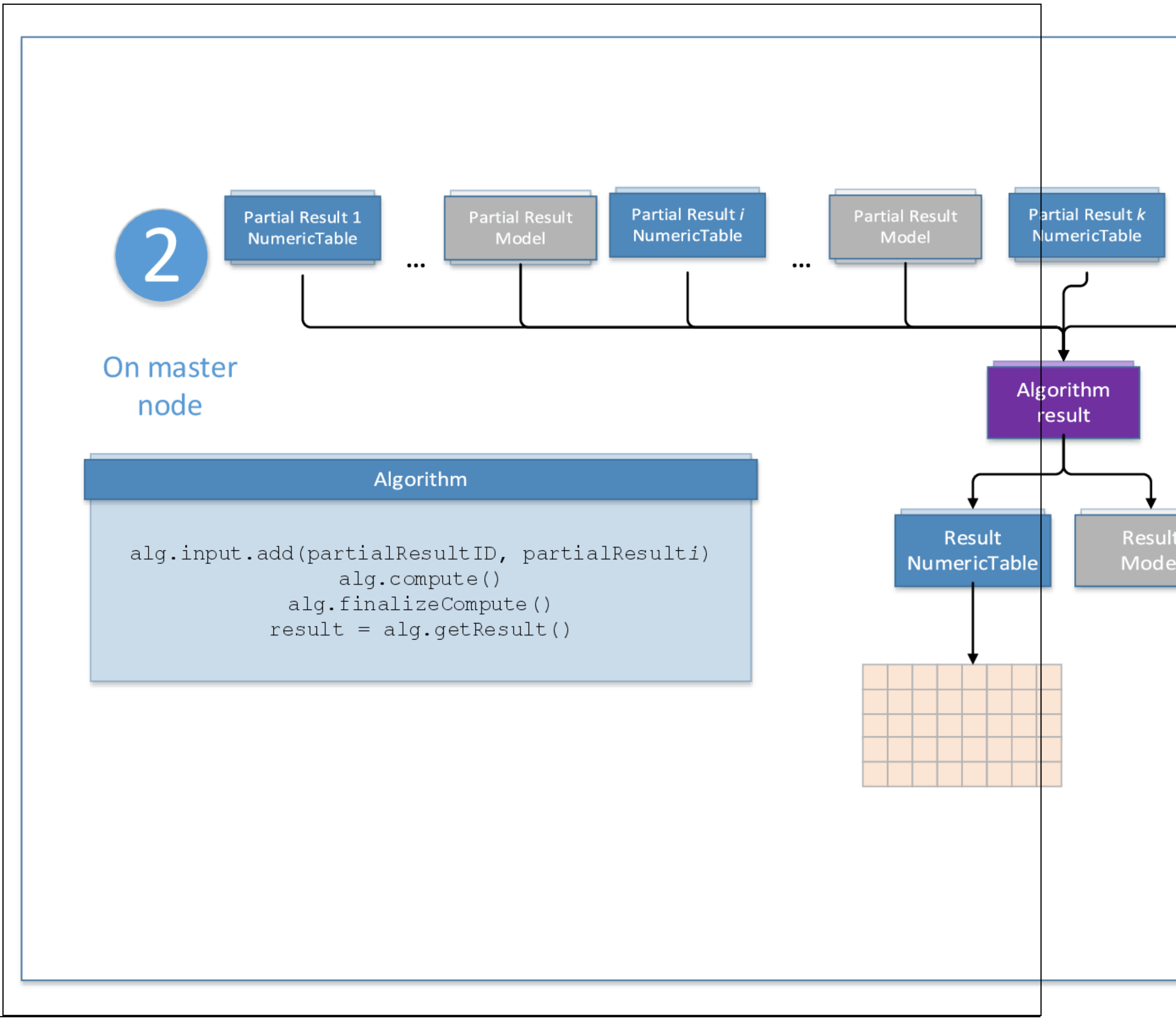
Some oneDAL algorithms enable processing of data sets distributed across several devices. In distributed processing mode, the `compute()` and the `finalizeCompute()` methods of a particular algorithm class are used. This computation mode assumes that the data set is split in nblocks blocks across computation nodes.

Computation is done in several steps. You need to define the computation step for an algorithm by providing the computeStep value to the constructor during initialization of the algorithm. Use the `compute()` method on each computation node to compute partial results. Use the `input.add()` method on the master node to add pointers to partial results processed on each computation node. When the last partial result arrives, call the `compute()` method followed by `finalizeCompute()` to produce final results. If the input data arrives in an asynchronous mode, you can use the `getStatus()` method for a given data source to check whether a new block of data is available for loading.

The computation schema is algorithm-specific. The following diagram illustrates a typical computation schema for distribute processing:



```
alg.input.set(Input ID, DataBlocki)
alg.compute()
partialResulti = alg.getPartialResult()
```



```
alg.input.add(partialResultID, partialResulti)
alg.compute()
alg.finalizeCompute()
result = alg.getResult()
```

For the algorithm-specific computation schema, refer to the Distributed Processing section in the description of an appropriate algorithm.

Distributed algorithms in oneDAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios. They include but are not limited to MPI* based cluster environments, Hadoop* or Spark* based cluster environments, low-level data exchange protocols, and more.

Usage Model: Training and Prediction

Typical workflows:

- Classification Usage Model
- Regression Usage Model
- Recommendation Systems Usage Model

Classification Usage Model

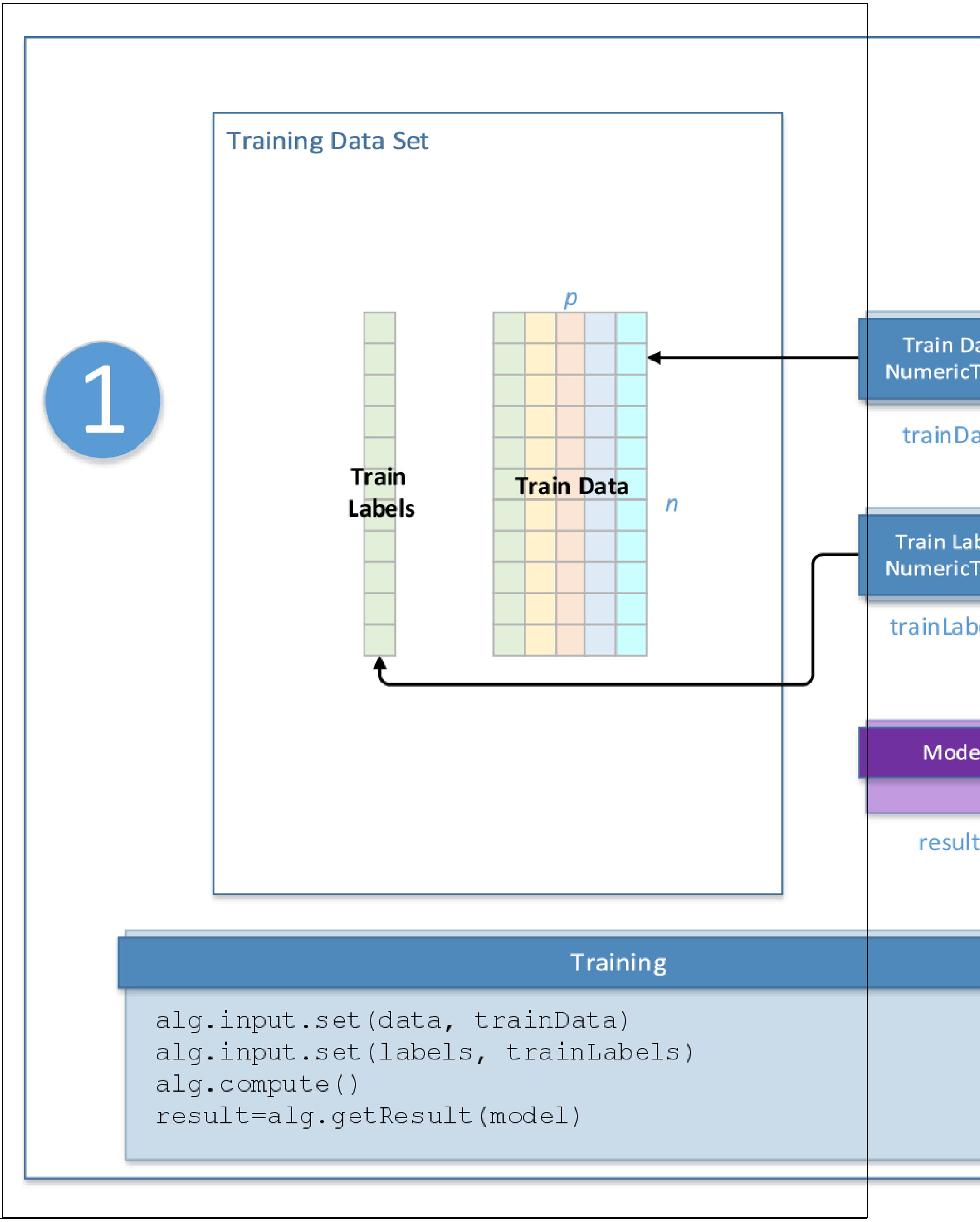
A typical workflow for classification methods includes training and prediction, as explained below.

Algorithm-Specific Parameters

The parameters used by classification algorithms at each stage depend on a specific algorithm. For a list of these parameters, refer to the description of an appropriate classification algorithm.

Training Stage

Classification Usage Model: Training Stage



```
alg.input.set(data, trainData)
alg.input.set(labels, trainLabels)
alg.compute()
result=alg.getResult(model)
```

At the training stage, classification algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Classification Algorithms

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the training data set. This table can be an object of any class derived from `NumericTable`.
`weights`	Weights of the observations in the training data set. Argument is optional, but it is required by the selected algorithms.
`labels`	Pointer to the $nimes1$ numeric table with class labels. This table can be an object of any class derived from NumericTable except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

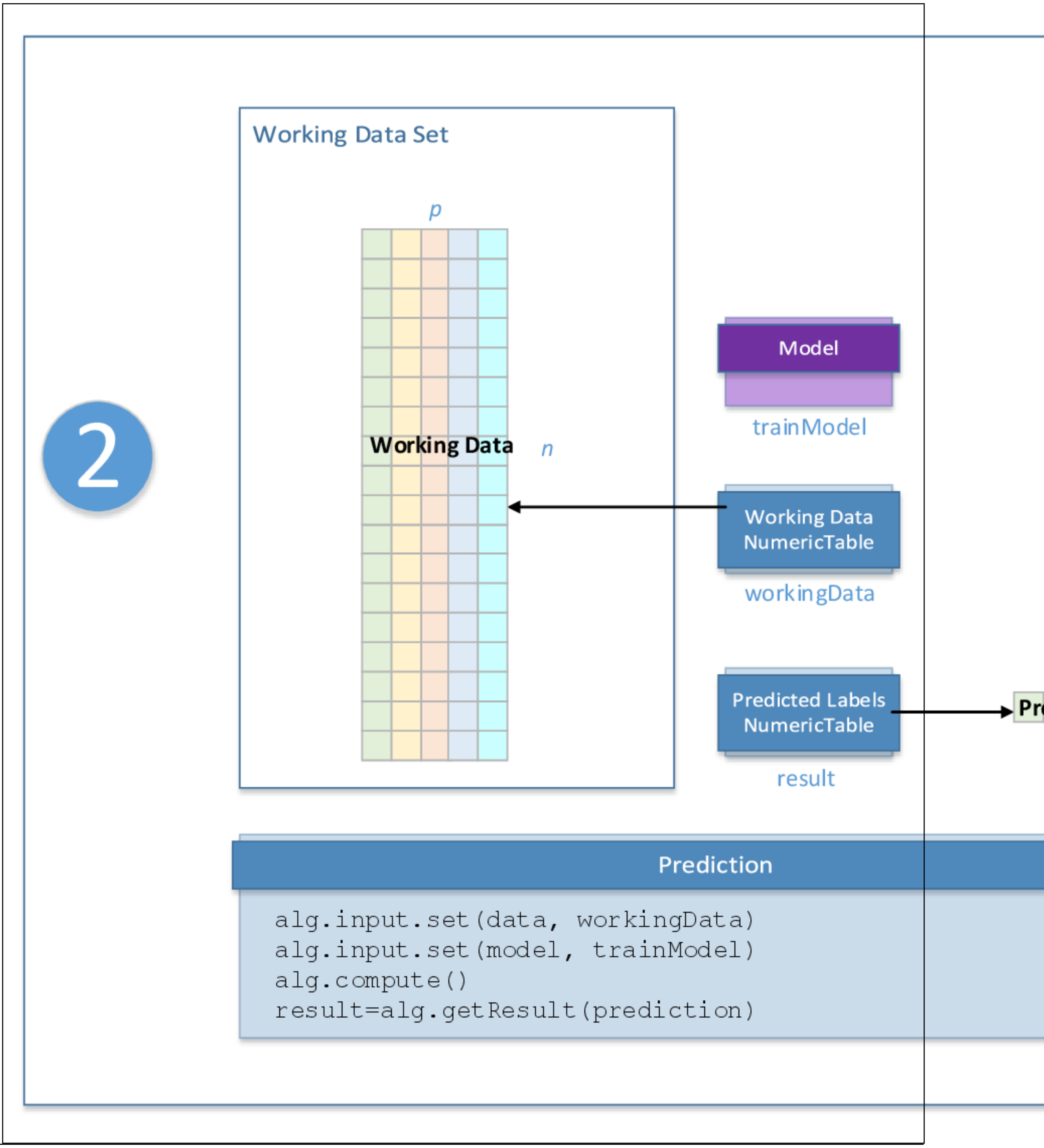
At the training stage, classification algorithms calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Classification Algorithms

Result ID	Result
`model`	Pointer to the classification model being trained. The result can only be an object of the `Model` class.

Prediction Stage

Classification Usage Model: Prediction Stage



```
alg.input.set(data, workingData)
alg.input.set(model, trainModel)
alg.compute()
result=alg.getResult(prediction)
```

At the prediction stage, classification algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Prediction Input for Classification Algorithms

Input ID	Input
data	Pointer to the $nimesp$ numeric table with the working data set. This table can be an object of any class derived from NumericTable.
model	Pointer to the trained classification model. This input can only be an object of the Model class.

At the prediction stage, classification algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Prediction Output for Classification Algorithms

Result ID	Result
prediction	Pointer to the $nimes1$ numeric table with classification results (class labels or confidence levels). **NOTE** By default, this table is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except PackedSymmetricMatrix and PackedTriangularMatrix.
probabilities	A numeric table of size $n \times \mathrm{nClasses}$, containing probabilities of classes computed when the computeClassProbabilities option is enabled. This result table is available for selected algorithms, see corresponding algorithm documentation for details.
logProbabilities	A numeric table of size $n \times \mathrm{nClasses}$, containing logarithms of classes' probabilities computed when the computeClassLogProbabilities option is enabled. This result table is available for selected algorithms, see corresponding algorithm documentation for details. **NOTE** By default, this table is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, CSRNumericTable.

Regression Usage Model

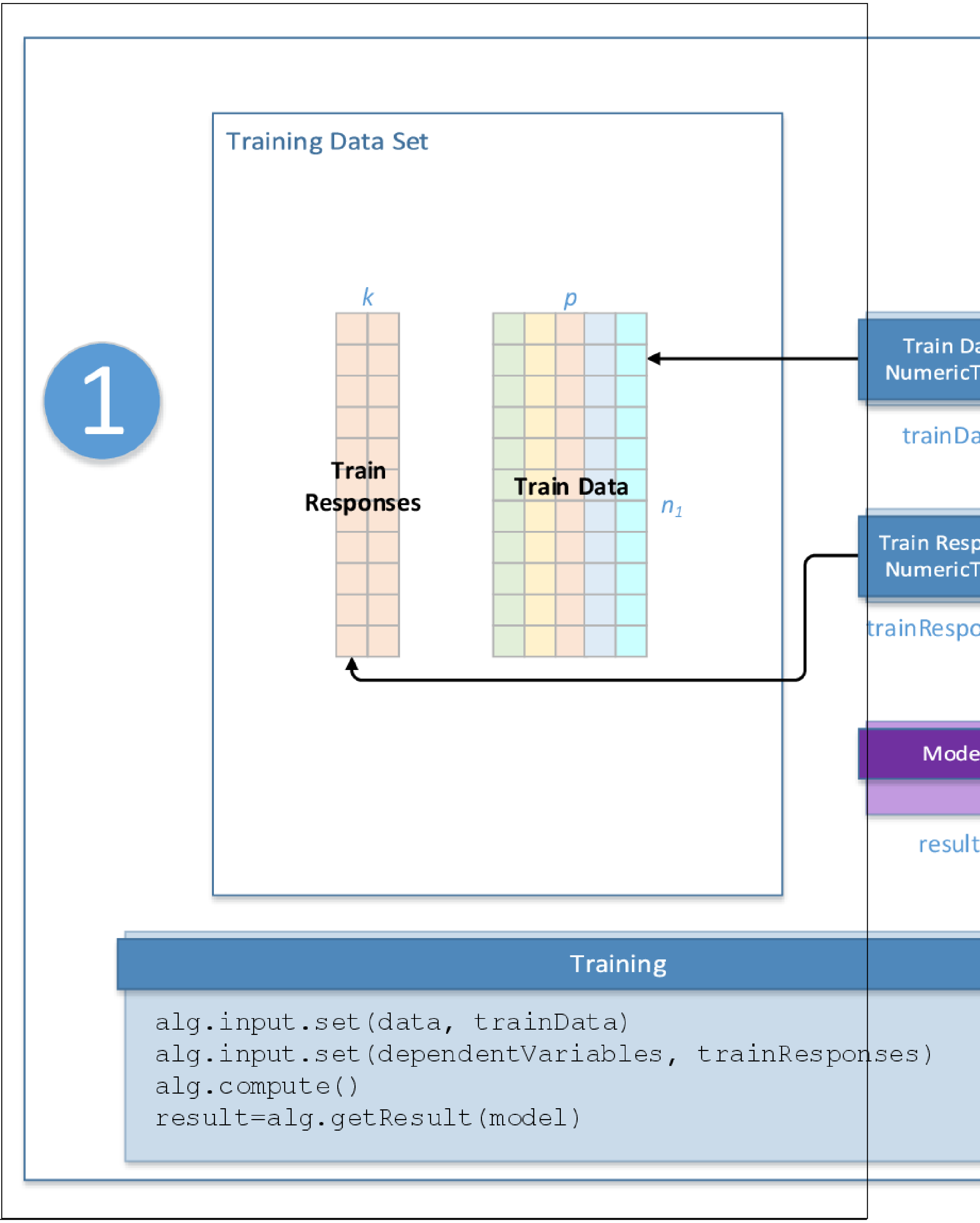
A typical workflow for regression methods includes training and prediction, as explained below.

Algorithm-Specific Parameters

The parameters used by regression algorithms at each stage depend on a specific algorithm. For a list of these parameters, refer to the description of an appropriate regression algorithm.

Training Stage

Regression Usage Model: Training Stage



```
alg.input.set(data, trainData)
alg.input.set(dependentVariables, trainResponses)
alg.compute()
result=alg.getResult(model)
```

At the training stage, regression algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Regression Algorithms

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the training data set. This table can be an object of any class derived from `NumericTable`.
`weights`	Weights of the observations in the training data set. Optional argument.
`dependentVariables`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables). This table can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

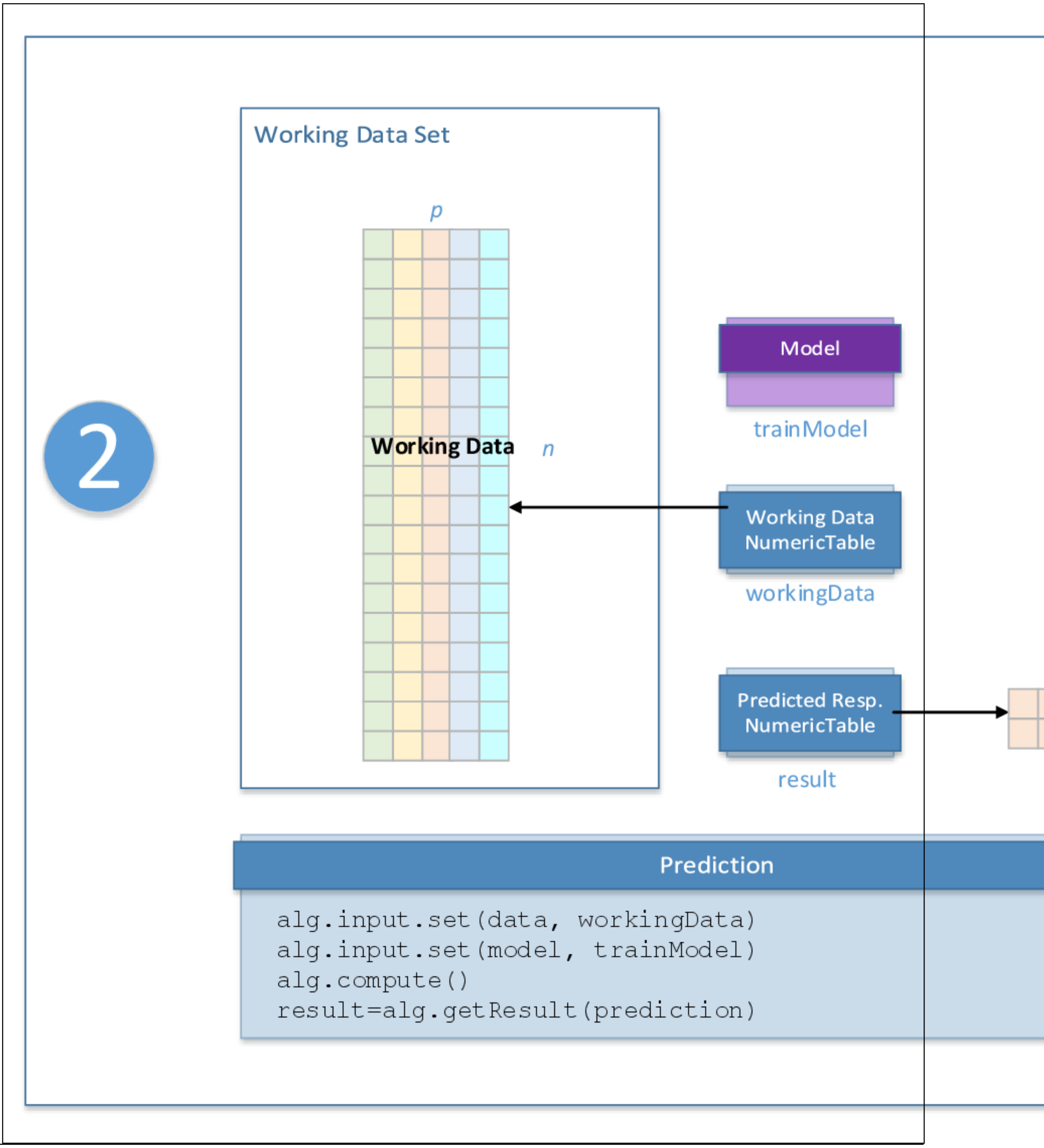
At the training stage, regression algorithms calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Regression Algorithms

Result ID	Result
`model`	Pointer to the regression model being trained. The result can only be an object of the `Model` class.

Prediction Stage

Regression Usage Model: Prediction Stage



```
alg.input.set(data, workingData)
alg.input.set(model, trainModel)
alg.compute()
result=alg.getResult(prediction)
```

At the prediction stage, regression algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Prediction Input for Regression Algorithms

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the working data set. This table can be an object of any class derived from `NumericTable`.
`model`	Pointer to the trained regression model. This input can only be an object of the `Model` class.

At the prediction stage, regression algorithms calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Prediction Output for Regression Algorithms

Result ID	Result
`prediction`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables). By default, this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

Recommendation Systems Usage Model

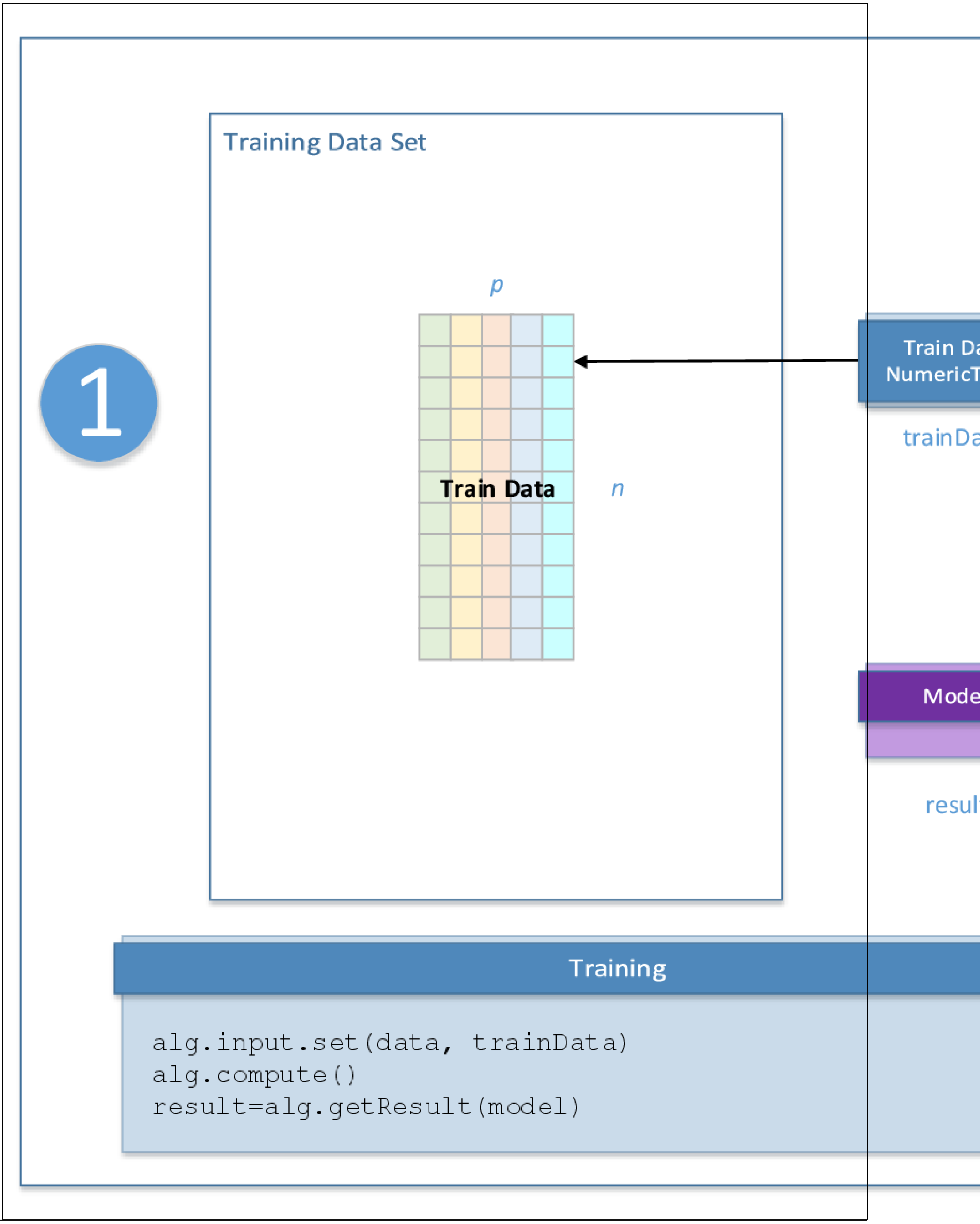
A typical workflow for methods of recommendation systems includes training and prediction, as explained below.

Algorithm-Specific Parameters

The parameters used by recommender algorithms at each stage depend on a specific algorithm. For a list of these parameters, refer to the description of an appropriate recommender algorithm.

Training Stage

Recommendation Systems Usage Model: Training Stage



```
alg.input.set(data, trainData)
alg.compute()
result=alg.getResult(model)
```

At the training stage, recommender algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Recommender Algorithms

Input ID	Input
`data`	Pointer to the $m \times n$ numeric table with the mining data. **NOTE** This table can be an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `PackedSymmetricMatrix`.

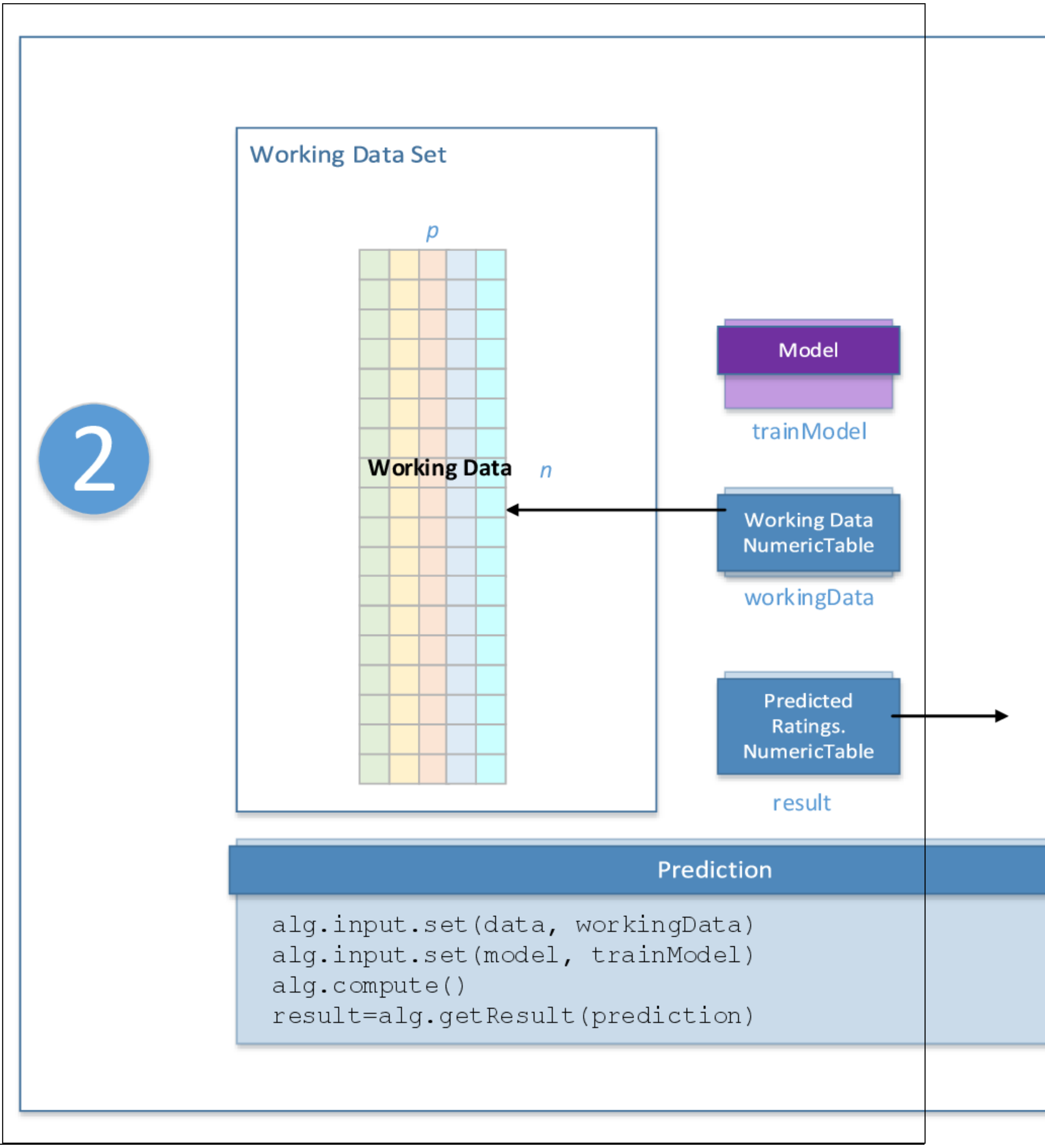
At the training stage, recommender algorithms calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Recommender Algorithms

Result ID	Result
`model`	Model with initialized item factors. **NOTE** The result can only be an object of the `Model` class.

Prediction Stage

Recommendation Systems Usage Model: Prediction Stage



```
alg.input.set(data, workingData)
alg.input.set(model, trainModel)
alg.compute()
result=alg.getResult(prediction)
```

At the prediction stage, recommender algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Prediction Input for Recommender Algorithms

Input ID	Input
`model`	Model with initialized item factors. **NOTE** This input can only be an object of the `Model` class.

At the prediction stage, recommender algorithms calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Prediction Output for Recommender Algorithms

Result ID	Result
`prediction`	Pointer to the $m \times n$ numeric table with predicted ratings. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Data Management

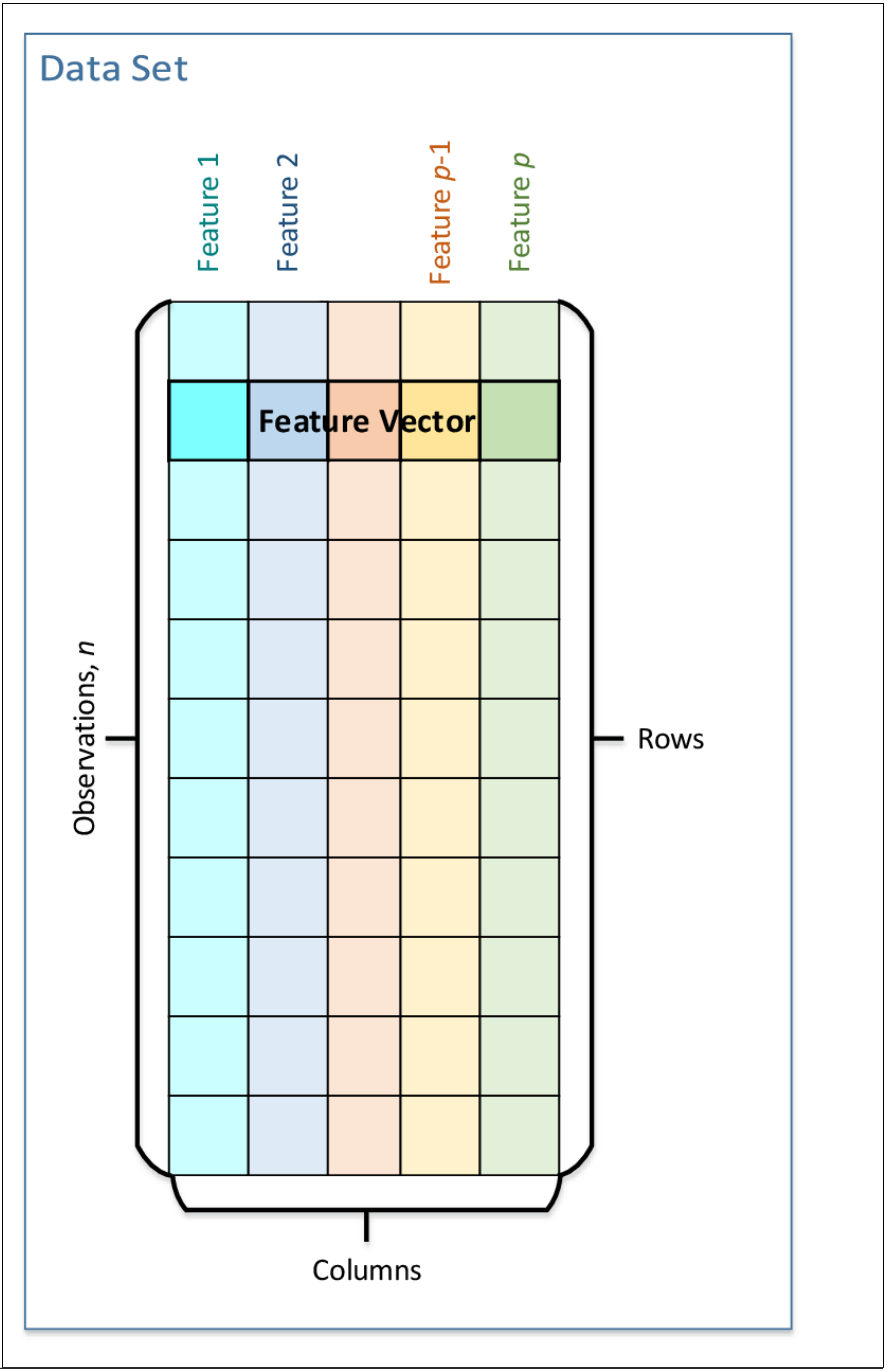
Effective data management is among key constituents of the performance of a data analytics application. For Intel® oneAPI Data Analytics Library, effective data management requires effectively performing the following operations:

1. Raw data acquisition, filtering, and normalization with data source interfaces.
2. Data conversion to a numeric representation for numeric tables.
3. Data streaming from a numeric table to an algorithm.

Depending on the usage model, you may also want to apply compression and decompression to the data you operate on. You can either use compression and decompression embedded into data source interfaces or apply data serialization and deserialization interfaces.

oneDAL provides a set of customizable interfaces to operate on your out-of-memory and in-memory data in different usage scenarios, which include batch processing, online processing, and distributed processing, as well as more complex scenarios, such as a combination of online and distributed processing.

One of key concepts of Data Management in oneDAL is a data set. A *data set* is a collection of data of a defined structure that characterizes an analyzed and modeled object. Specifically, the object is characterized by a set of attributes (Features), which form a Feature Vector of dimension p. Multiple feature vectors form a set of Observations of size n. oneDAL defines a tabular view of a data set where table rows represent observations and columns represent features.



Data Set
Feature 1
Feature 2
Feature p-1
Feature p
Feature Vector
Observations, n
Rows
Columns

An observation corresponds to a particular measurement of an observed object, and therefore when measurements are done, at distinct moments in time, the set of observations characterizes how the object evolves in time.

It is not a rare situation when only a subset of features can be measured at a given moment. In this case, the non-measured features in the feature vector become blank, or missing. Special statistical techniques enable recovery (emulation) of missing values.

You normally start working with oneDAL by selecting an appropriate data source, which provides an interface for your raw data set. oneDAL data sources support categorical, ordinal, and continuous features. It means that data sources can automatically transform non-numeric categorical and ordinary data into a numeric representation. When the structure of your raw data is more complex or when the default transformation mechanism does not fit your needs, you may customize the data source by implementing a custom derivative class.

Because a data source is typically associated with out-of-memory data, such as files, databases, and so on, streaming out-of-memory data into memory and back is among major functions of a data source. However you can also use a data source to implement an in-memory non-numeric data transformation into a numeric form.

A numeric table is a key interface to operate on numeric in-memory data. oneDAL supports several important cases of a numeric data layout: homogeneous tables, arrays of structures, and structures of arrays, as well as Compressed Sparse Row (CSR) encoding for sparse data.

oneDAL algorithms operate with in-memory numeric data accessed through Numeric table interfaces.

Numeric Tables

- Generic Interfaces
- Essential Interfaces for Algorithms
- Types of Numeric Tables

Effective data management is one of the key components for achieving good performance in data analytics applications. oneDAL defines the `NumericTable` class that is responsible for storage of and access to the datasets represented in numeric format on the computational node:

- `NumericTable` does not track data available on other nodes. The logic that controls synchronization of data between nodes should be implemented on the application level.
- `NumericTable` does not accumulate information about data coming in streaming way. All necessary computations are done on the level of the oneDAL algorithm and/or application software.

The library supports the following data layouts:

- Heterogeneous, Array Of Structures (AOS)
- Heterogeneous, Structure Of Arrays (SOA)
- Homogeneous, dense
- Homogeneous matrix, dense
- Homogeneous symmetric matrix, packed
- Homogeneous triangular matrix, packed
- Homogeneous, sparse CSR

The optimal data layout for homogeneous and heterogeneous numeric tables highly depends on a particular algorithm. You can find algorithm-specific guidance in the Performance Considerations section for the appropriate algorithm.

Generic Interfaces

Numeric tables provide interfaces for data management, such as memory allocation and deallocation, and respective memory access methods, dictionary management, and table size management.

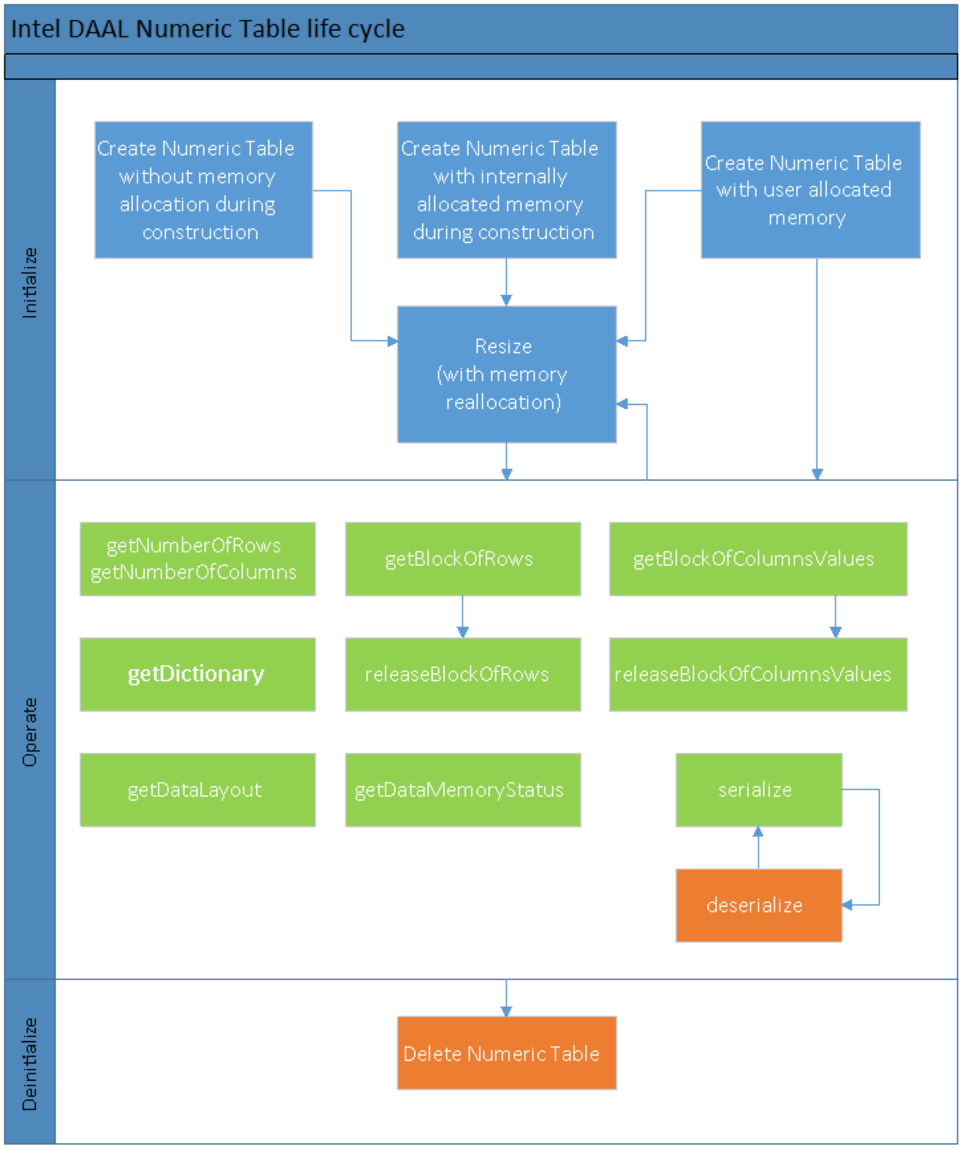
The life cycle of a numeric table consists of the following major steps:

1. Initialize
2. Operate

3. Deinitialize

The following diagram shows possible flows and transitions between the states of the numeric table for each step. The description of these steps applies to different types of numeric tables supported in the library, such as CSR, with appropriate changes in the method names and respective arguments.

Numeric Table Lifecycle


Intel DAAL Numeric Table life cycle
Initialize
Create Numeric Table without memory allocation during construction
Create Numeric Table with internally allocated memory during construction
Create Numeric Table with user allocated memory
Resize (with memory reallocation)
Operate
getNumberOfRows getNumberOfColumns
getBlockOfRows
getBlockOfColumnsValues
getDictionary
releaseBlockOfRows
releaseBlockOfColumnsValues
getDataLayout
getDataMemoryStatus
serialize
deserialize
Deinitialize
Delete Numeric Table

Initialize

A data dictionary is associated with numeric table over its whole life cycle. If the dictionary is not explicitly provided by the user during initialization, it is automatically constructed using the parameters provided to the constructor of the numeric table.

If you need to modify the numeric table dictionary by changing, for example, the number of attributes (that equals to the number of columns in the table), create another instance of the numeric table to work with the data. Modification of the dictionary via respective methods for the existing and initialized numeric table does not imply re-allocation of the internal data structures of the numeric table and can result in unpredicted behavior of the application.

oneDAL provides several constructors for numeric tables to cover a variety of table initialization scenarios. The constructors require the numbers of rows and columns for the table or a dictionary. If you do not have the dictionary or sizes of the numeric table at the time of construction, you can use the constructor with default values and sizes. The following scenarios are available for use of constructors:

- If the table size is unknown at the time of object construction, you can construct an empty table and change the size and allocate the memory later. You can also use the constructor to specify the sizes, but provide a pointer to the memory later:

```
HomogenNumericTable<float> table(nColumns, nRows, NumericTable::doNotAllocate);
float data[nColumns * nRows];
table.setArray(data, nRows);
```

- If the table size is known but the data is not yet in memory, oneDAL can allocate the memory automatically at the time of object construction and even initialize the memory, that is, allocate the matrix with zero elements:

```
HomogenNumericTable<float> table(nColumns, nRows, NumericTable::doAllocate, 0.0);
```

- If the data is already available in memory by the time of object construction, you can provide a pointer to this data through the appropriate constructor:

```
float data[nColumns * nRows];
HomogenNumericTable<float> table(data, nColumns, nRows);
```

To allocate or reallocate the memory after construction of the numeric table, use service methods:

- `resize()`

  This method modifies the number of rows in the table according to the provided parameter and operates according to the description below:

  - If a memory buffer for the numeric table is not allocated, this method allocates memory of the respective size for the table.
  - If a memory buffer for the numeric table is allocated by the library and the number of rows passed to the function requires a larger memory buffer, the method deallocates it and allocates a new buffer of the respective size.
  - If a memory buffer for the numeric table is provided by the user and the number of rows passed to the function requires a larger memory buffer, the method internally allocates a new buffer of the respective size. The memory buffer provided by the user is not deallocated by the library in this case.
  - Otherwise, the method modifies the respective number of rows in the internal data structures.

Operate

After initialization or re-initialization of a numeric table, you can use the following methods for the numeric table to access the data:

- `getBlockOfRows()` and `releaseBlockOfRows()`

  The `getBlockOfRows()` method provides access to a data block stored in the numeric table. The `rwflag` argument specifies read or write access. Provide the object of the BlockDescriptor type to the method to interface the requested block of rows. This object, the block descriptor, represents the data in the contiguous raw-major layout with the number of rows specified in the method and number of columns specified in the numeric table.

In oneDAL you can represent the data in the block descriptor with the data type different from the data type of the numeric table. For example: you can represent a homogeneous data with the float data type, while the block descriptor represents the requested data in double. You can specify the required data type during the construction of the block descriptor object. Make sure to call the `releaseBlockOfRows()` method after a call to `getBlockOfRows()`. The data types of the numeric table and block descriptor, as well as the rwflag argument of the `getBlockOfRows()` method, define the behavior of `releaseBlockOfRows()`:

- If `rwflag` is set to `writeOnly` or `readWrite`, `releaseBlockOfRows()` writes the data from the block descriptor back to the numeric table.
- If the numeric table and block descriptor use different data types or memory layouts, `releaseBlockOfRows()` deallocates the allocated buffers regardless of the value of `rwflag`.

```
HomogenNumericTable<double> table(data, nColumns, nRows);
BlockDescriptor<float> block;
table.getBlockOfRows(firstReadRow, nReadRows, readOnly, block);
float *array = block.getBlockPtr();
for (size_t row = 0; row < nReadRows; row++)
{
  for (size_t col = 0; col < nColumns; col++)
  {
    std::cout << array[row * nColumns + col] << "   ";
  }
  std::cout << std::endl;
}
table.releaseBlockOfRows(block);
```

- `getBlockOfColumnValues()` and `releaseBlockOfColumnValues()`

  These methods provide access to values in the specific column of a numeric table, similarly to `getBlockOfRows()` and `releaseBlockOfRows()`.
- `getNumberOfRows()` and `getNumberOfColumns()`

  Call these methods to determine the number of rows and columns, respectively, associated with a given numeric table.
- `getDictionary()` and `resetDictionary()`, as well as `getFeatureType()` and `getNumberOfCategories()`.

  These methods provide access to the data dictionary associated with a given numeric table. See Data Dictionaries for more details.
- `getDataMemoryStatus()`

  Call this method to determine whether the memory is allocated by the `allocateDataMemory()` method, a user provided a pointer to the allocated data, or no data is currently associated with the numeric table. Additionally, the `getArray()` method is complimentary to `setArray()` and provides access to the data associated with a given table of a given layout.
- `serialize()` and `deserialize()`

  The `serialize()` method enables you to serialize the numeric table. Call the deserialization method `deserialize()` after each call to `serialize()`, but before a call to other data access methods.

Deinitialize

After you complete your work with a data resource, the appropriate memory is deallocated implicitly in the destructor of the numeric table.

NOTE

- If the library internally allocates or reallocates the memory buffers for the data inside the numeric table, do not use the pointer returned by the getArray() method of the numeric table after its destruction.
- The default data type for a homogeneous numeric table is float.
- **Python*:** When creating a numpy array from a numeric table, make sure that a reference to the numeric table exists as long as a reference to the derived numpy array is being used.

Examples

C++:

- datasource/datastructures_merged.cpp
- datasource/datastructures_homogen.cpp

Java*:

- datasource/DataStructuresMerged.java
- datasource/DataStructuresHomogen.java

Essential Interfaces for Algorithms

In addition to Generic Interfaces, more methods enable interfacing numeric tables with algorithms.

The getDataLayout method provides information about the data layout:

Data Layout	Description
soa	Structure-Of-Arrays (SOA). Values of individual data features are stored in contiguous memory blocks.
aos	Array-Of-Structures (AOS). Feature vectors are stored in contiguous memory block.
csr_Array	Condensed-Sparse-Row (CSR).
lowerPackedSymetricMatrix	Lower packed symmetric matrix
lowerPackedTriangularMatrix	Lower packed triangular matrix
upperPackedSymetricMatrix	Upper packed symmetric matrix
upperPackedTriangularMatrix	Upper packed triangular matrix
unknown	No information about data layout or unsupported layout.

Rather than access the entire in-memory data set, it is often more efficient to process it by blocks. The key methods that oneDAL algorithms use for per-block data access are `getBlockOfRows()` and `getBlockOfColumnValues()`. The `getBlockOfRows()` method accesses a block of feature vectors, while the `getBlockOfColumnValues()` method accesses a block of values for a given feature. A particular algorithm uses `getBlockOfRows()`, `getBlockOfColumnValues()`, or both methods to access the data. The efficiency of data access highly depends on the data layout and on whether the data type of the feature is natively supported by the algorithm without type conversions. Refer to the Performance Considerations section in the description of a particular algorithm for a discussion of the optimal data layout and natively supported data types.

When the data layout fits the per-block data access pattern and the algorithm requests the data type that corresponds to the actual data type, the `getBlockOfRows()` and `getBlockOfColumnValues()` methods avoid data copying and type conversion. However, when the layout does not fit the data access pattern or when type conversion is required, both methods automatically re-pack and convert data as required.

When dealing with custom or unsupported data layouts, you must implement NumericTableIface, DenseNumericTableIface interfaces, and optionally CSRNumericTableIface or PackedNumericTableIface interfaces.

Some algorithms, such as Moments of Low Order, compute basic statistics (minimums, maximums, and so on). The other algorithms, such as Correlation and Variance-Covariance Matrices or Principal Component Analysis, require some basic statistics on input. To avoid duplicated computation of basic statistics, oneDAL provides methods to store and retrieve basic statistics associated with a given numeric table: `basicStatistics.set()` and `basicStatistics.get()`. The following basic statistics are computed for each numeric table:

- minimum - minimum
- maximum - maximum
- sum - sum
- sumSquares - sum of squares

NOTE The default data type of basic statistics is float.

Special Interfaces for the HomogenNumericTable and Matrix Classes

- Use the assign method to initialize elements of a dense homogeneous numeric table with a certain value, that is, to set all elements of the matrix to zero.
- Use the operator [] method to access rows of a homogeneous dense numeric table.

Special Interfaces for the PackedTriangularMatrix and PackedSymmetricMatrix Classes

- While you can use generic `getArray()` and `setArray()` methods to access the data in a packed format, in algorithms that have specific implementations for a packed data layout, you can use more specific `getPackedValues()` and `releasePackedValues()` methods.

Special Interfaces for the CSRNumericTable Class

- To access three CSR arrays (values , columns, and rowIndex), use `getArrays()` and `setArrays()` methods instead of generic `getArray()` and `setArray()` methods. For details of the arrays, see CSR data layout.
- Similarly, in algorithms that have specific implementations for the CSR data layout, you can use more specific `getBlockOfCSRValues()` and `releaseBlockOfCSRValues()` methods.

Special Interfaces for the MergedNumericTable Class

- To add a new array to the object of the MergedNumericTable class, use the `addNumericTable()` method.

Types of Numeric Tables

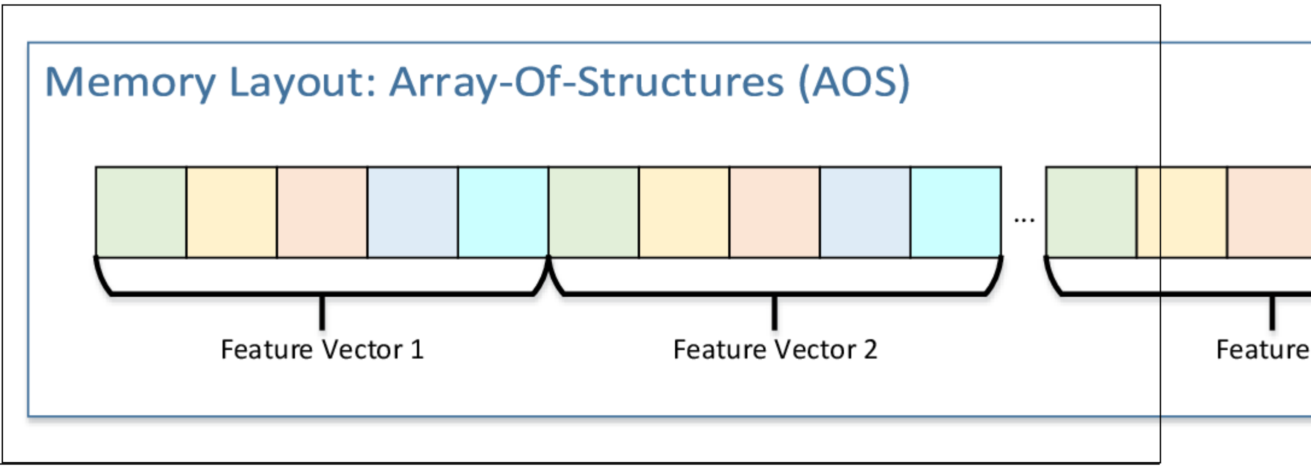
Heterogeneous Numeric Tables

Heterogeneous numeric tables enable you to deal with data structures that are of different data types by nature. oneDAL provides two ways to represent non-homogeneous numeric tables: AOS and SOA.

AOS Numeric Table

AOS Numeric Table provides access to observations (feature vectors) that are laid out in a contiguous memory block:

Array-Of-Structures (AOS) Memory Layout



Examples

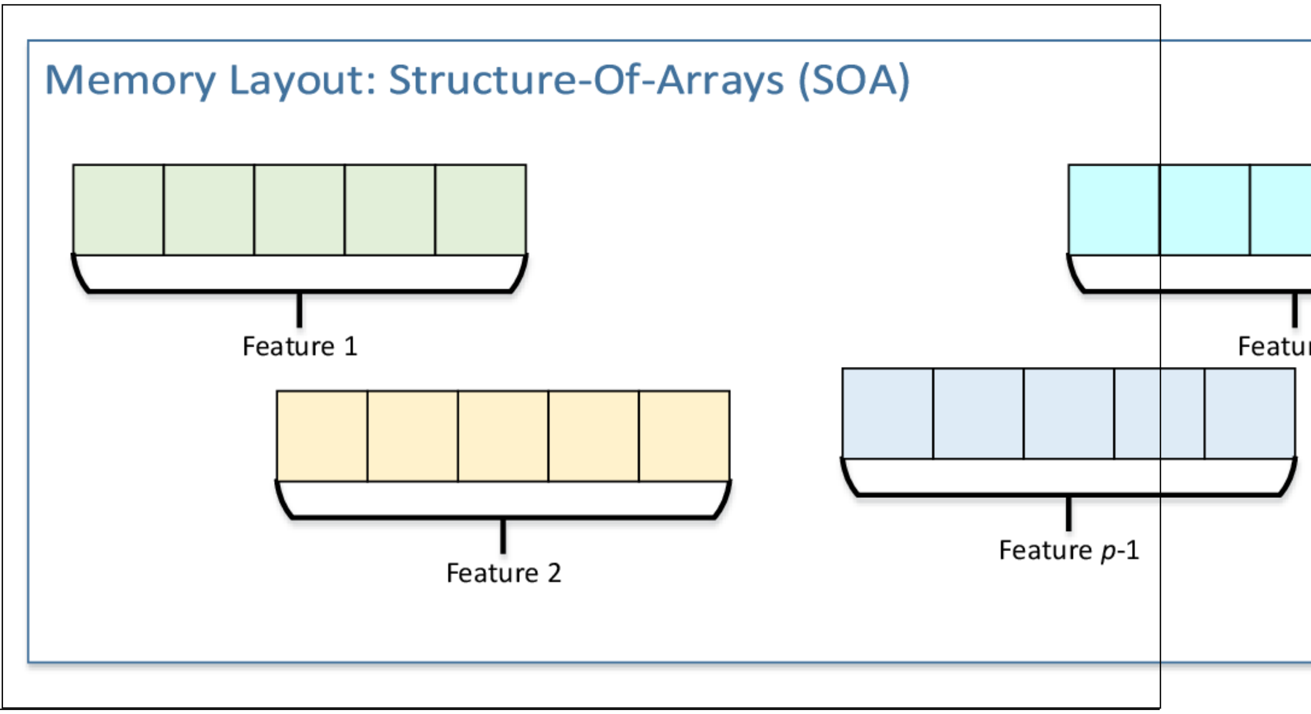
C++: datasource/datastructures_aos.cpp

Java*: datasource/DataStructuresAOS.java

SOA Numeric Table

SOA Numeric Table provides access to data sets where observations for each feature are laid out contiguously in memory:

Structure-Of-Arrays (SOA) Memory Layout



Examples

C++: datasource/datastructures_soa.cpp

Java*: datasource/DataStructuresSOA.java

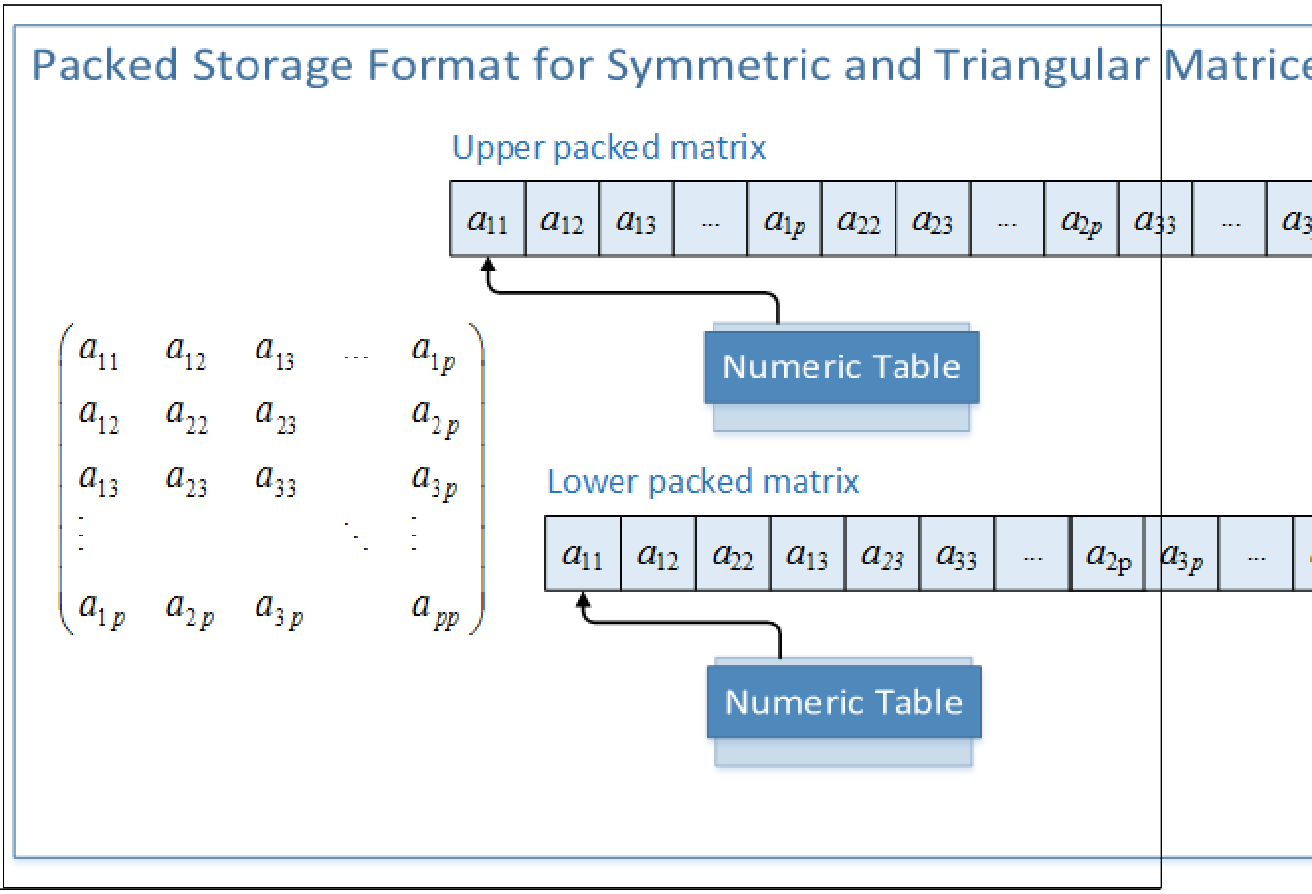
Homogeneous Numeric Tables

Use homogeneous numeric tables, that is, objects of the `HomogenNumericTable` class, and matrices, that is, objects of the `Matrix`, `PackedTriangularMatrix`, and `PackedSymmetricMatrix` classes, when all the features are of the same basic data type. Values of the features are laid out in memory as one contiguous block in the row-major order, that is, *Observation 1*, *Observation 2*, and so on. In oneDAL, `Matrix` is a homogeneous numeric table most suitable for matrix algebra operations.

For triangular and symmetric matrices with reduced memory footprint, special classes are available: `PackedTriangularMatrix` and `PackedSymmetricMatrix`. Use the DataLayout enumeration to choose between representations of triangular and symmetric matrices:

- Lower packed: `lowerPackedSymetricMatrix` or `lowerPackedTriangularMatrix`
- Upper packed: `upperPackedTriangularMatrix` or `upperPackedSymetricMatrix`

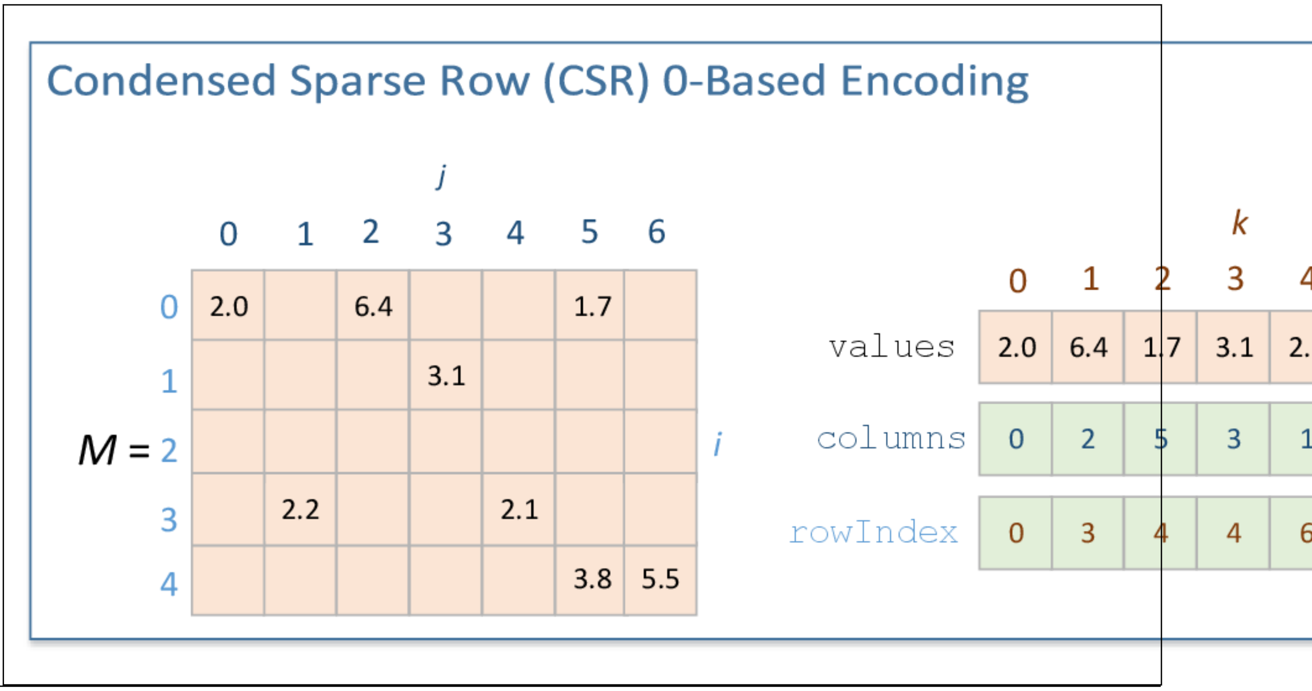
Packed Storage Format for Symmetric and Triangular Matrices


Packed Storage Format for Symmetric and Triangular Matrice
Upper packed matrix
a_{11} a_{12} a_{13} ... a_{1p} a_{22} a_{23} ... a_{2p} a_{33} ... a_3
Numeric Table
a_{11} a_{12} a_{13} ... a_{1p}
a_{12} a_{22} a_{23} a_{2p}
a_{13} a_{23} a_{33} a_{3p}
a_{1p} a_{2p} a_{3p} a_{pp}
Lower packed matrix
a_{11} a_{12} a_{22} a_{13} a_{23} a_{33} ... a_{2p} a_{3p} ...
Numeric Table

CSR Numeric Table

oneDAL offers the `CSRNumericTable` class for a special version of a homogeneous numeric table that encodes sparse data, that is, the data with a significant number of zero elements. The library uses the Condensed Sparse Row (CSR) format for encoding:

Condensed Sparse Row (CSR) 0-Based Encoding

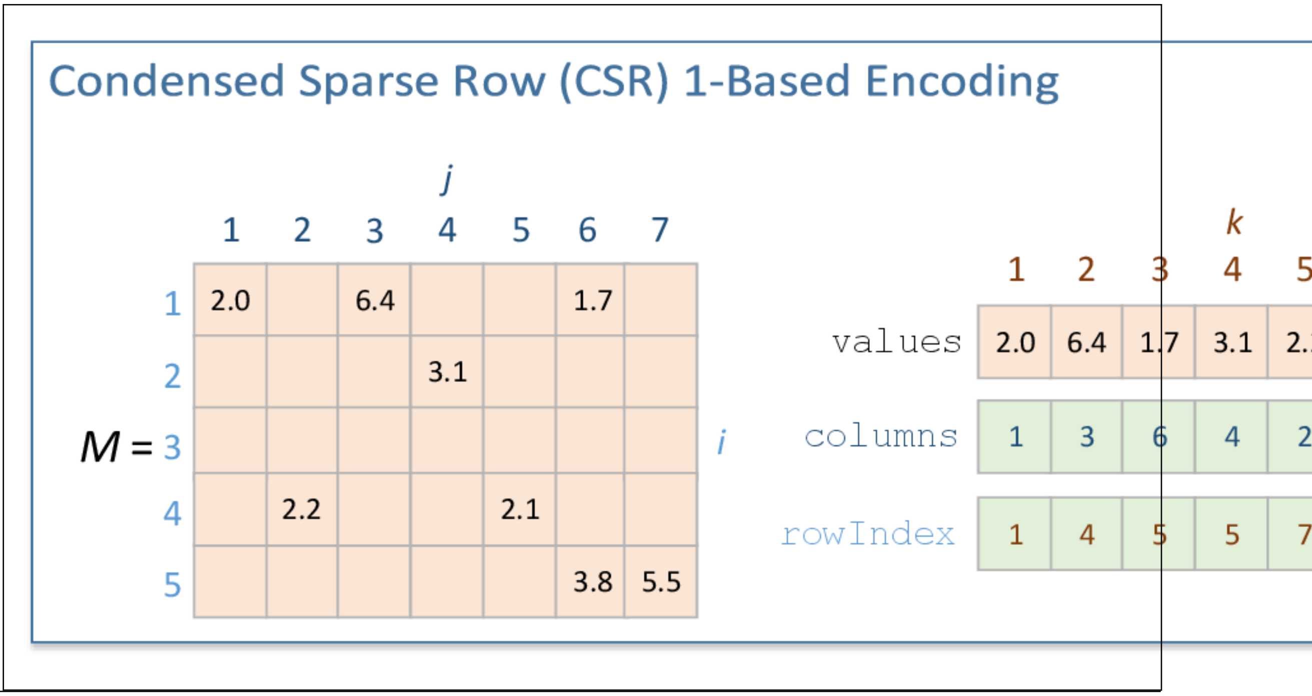


Condensed Sparse Row (CSR) 0-Based Encoding

	0	1	2	3	4	5	6
0	2.0		6.4			1.7	
1				3.1			
2							
3		2.2			2.1		
4						3.8	5.5

	0	1	2	3	4
values	2.0	6.4	1.7	3.1	2.
columns	0	2	5	3	1
rowIndex	0	3	4	4	6

Condensed Sparse Row (CSR) 1-Based Encoding



Condensed Sparse Row (CSR) 1-Based Encoding

	1	2	3	4	5	6	7
1	2.0		6.4			1.7	
2				3.1			
3							
4		2.2			2.1		
5						3.8	5.5

	1	2	3	4	5
values	2.0	6.4	1.7	3.1	2.
columns	1	3	6	4	2
rowIndex	1	4	5	5	7

Three arrays describe the sparse matrix M as follows:

- The array values contains non-zero elements of the matrix row-by-row.
- The j-th element of the array columns encodes the column index in the matrix M for the j-th element of the array values.
- The i-th element of the array rowIndex encodes the index in the array values corresponding to the first non-zero element in rows indexed i or greater. The last element in the array rowIndex encodes the number of non-zero elements in the matrix M.

The library supports 1-based CSR encoding only. In C++ you can specify it by providingoneBased value through the indexing parameter of type `CSRIndexing` in the constructor of `CSRNumericTable`.

Examples

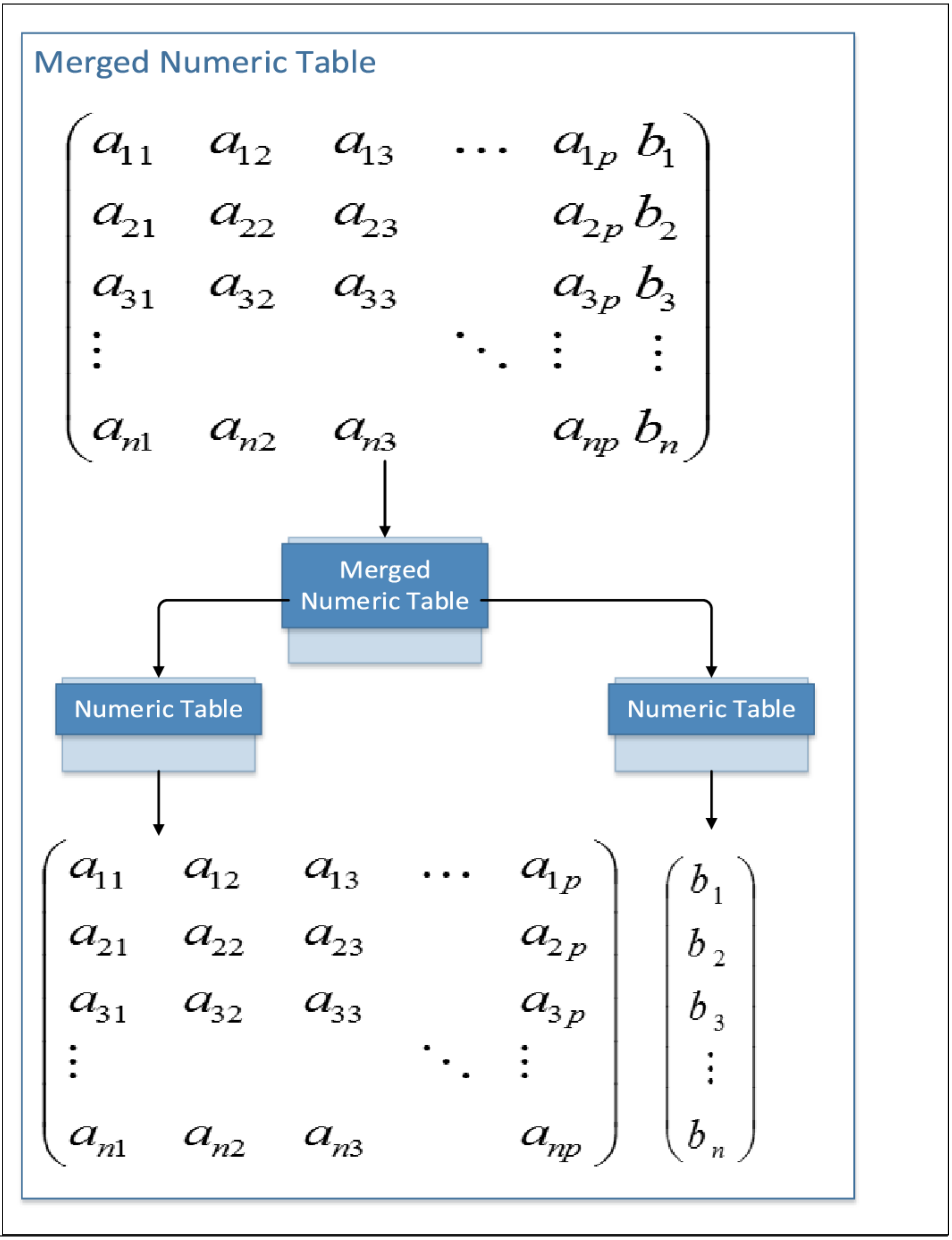
C++: datasource/datastructures_csr.cpp

Java*: datasource/DataStructuresCSR.java

Merged Numeric Table

oneDAL offers the `MergedNumericTable` class for tables that provides access to data sets comprising several logical components, such as a set of feature vectors and corresponding labels. This type of tables enables you to read those data components from one data source. This special type of numeric tables can hold several numeric tables of any type but `CSRNumericTable`. In a merged numeric table, arrays are joined by columns and therefore can have different numbers of columns. In the case of different numbers of rows in input matrices, the number of rows in a merged table equals $min(r_1, r_2, ..., r_m)$, where r_i is the number of rows in the i-th matrix, $i = 1, 2, 3, ..., m$.

Merged Numeric Table


Merged Numeric Table
Merged Numeric Table
Numeric Table
Numeric Table
a_{11} a_{12} a_{13} ... a_{1p} b_1
a_{21} a_{22} a_{23} a_{2p} b_2
a_{31} a_{32} a_{33} a_{3p} b_3
a_{n1} a_{n2} a_{n3} a_{np} b_n

Examples

C++: datasource/datastructures_merged.cpp

Java*: datasource/DataStructuresMerged.java

Data Sources

Data sources define interfaces for access and management of data in raw format and out-of-memory data. A data source is closely coupled with the data dictionary that describes the structure of the data associated with the data source. To create the associated data dictionary, you can do one of the following:

- While constructing a data source object, specify whether it should automatically create and initialize the associated data dictionary.
- Call the createDictionaryFromContext() method.

The getDictionary() method returns the dictionary associated with the data source.

Data sources stream and transform raw out-of-memory data into numeric in-memory data accessible through numeric table interfaces. A data source is associated with the corresponding numeric table. To allocate the associated numeric table, you can do one of the following:

- While constructing a data source object, specify whether it should automatically allocate the numeric table.
- Call the allocateNumericTable() method.

The getNumericTable() method returns the numeric table associated with the data source.

To retrieve the number of columns (features) in a raw data set, use the getNumberOfColumns() method. To retrieve the number of rows (observations) available in a raw data set, use the getNumberOfAvailableRows() method. The getStatus() method returns the current status of the data source:

- readyForLoad - the data is available for the load operation.
- waitingForData - the data source is waiting for new data to arrive later; designated for data sources that deal with asynchronous data streaming, that is, the data arriving in blocks at different points in time.
- endOfData- all the data is already loaded.

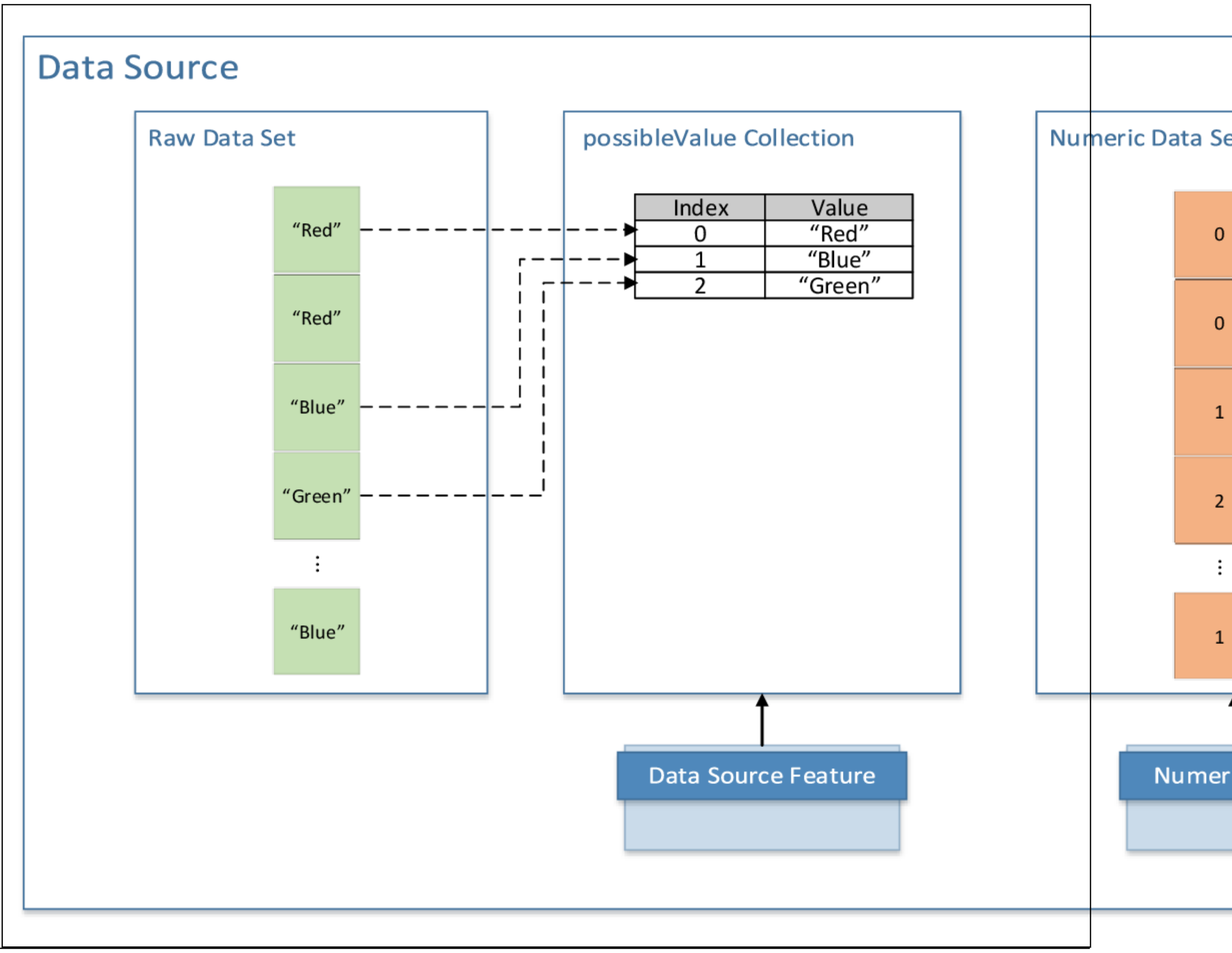
Because the entire out-of-memory data set may fail to fit into memory, as well as for performance reasons, oneDAL implements data loading in blocks. Use the loadDataBlock() method to load the next block of data into the numeric table. This method enables you to load a data block into an internally allocated numeric table or into the provided numeric table. In both cases, you can specify the number of rows or not. The method also recalculates basic statistics associated with this numeric table.

oneDAL maintains the list of possible values associated with categorical features to convert them into a numeric form. In this list, a new index is assigned to each new value found in the raw data set. You can get the list of possible values from the possibleValues collection associated with the corresponding feature in the data source. In the case you have several data sets with same data structure and you want to use continuous indexing, do the following:

1. Retrieve the data dictionary from the last data source using the getDictionary() method.
2. Assign this dictionary to the next data source using the setDictionary() method.

3. Repeat these steps for each next data source.

Reading from a Data Source



oneDAL implements classes for some popular types of data sources. Each of these classes takes a feature manager class as the class template parameter. The feature manager parses, filters, and normalizes the data and converts it into a numeric format. The following are the data sources and the corresponding feature manager classes:

- Text file (FileDataSource class), to be used with the CSVFeatureManager class
- ODBC (ODBCDataSource class), to be used with the MySQLFeatureManager class
- In-memory text (StringDataSource class), to be used with the CSVFeatureManager class
- KDB relational database (KDBDataSource class), to be used with the KDBFeatureManager class

CSVFeatureManager provides additional capabilities for features modification. Use addModifier() to enable specific modification when loading data to a numeric table:

- Add the ColumnFilter object if you need to have a predefined subset of features loaded
- Add the OneHotEncoder object if you need a categorical feature to be encoded using the one-hot scheme

Feature managers provide additional capabilities for the modification of the input data during its loading. Use the Feature modifier entity to define desired modification. Feature modifiers enables you to implement a wide range of feature extraction or transformation techniques, for instance, feature binarization, one-hot-encoding, or polynomial features generation. To enable specific modification, use the addModifier() method that accepts two parameters:

- featureIds - a subset of feature identifiers for which you want to apply modification.

- featureModifier - an implementation of the Feature modifier, an object that implements the FeatureModifierIface interface and specifies the way how features of the input data set should be modified and written to the output numeric table.

Typical feature modifiers usage scenario is the following:

1. Create the data source object and specify a feature manager and its parameters.
2. Define a subset of features for modification and proper feature modifier.
3. Add modifier to the feature manager of the data source object.
4. Call loadDataBlock(), it causes data set loading and applying specified modification procedure to the features of the data set.

The code block bellow demonstrates feature modifiers usage scenario in case of FileDataSource and CSVFeatureManager.

```
// Crate DataSource object (for example FileDataSource)
FileDataSource<CSVFeatureManager> ds("file.csv", options);

// Specify features subset and modifier
auto featureIds = features::list("f1", "f2");
auto featureModifier = modifiers::csv::continuous();

// Add modifier to feature manager
ds.getFeatureManager().addModifier(featureIds, modifier);

// Cause data loading
ds.loadDataBlock();
```

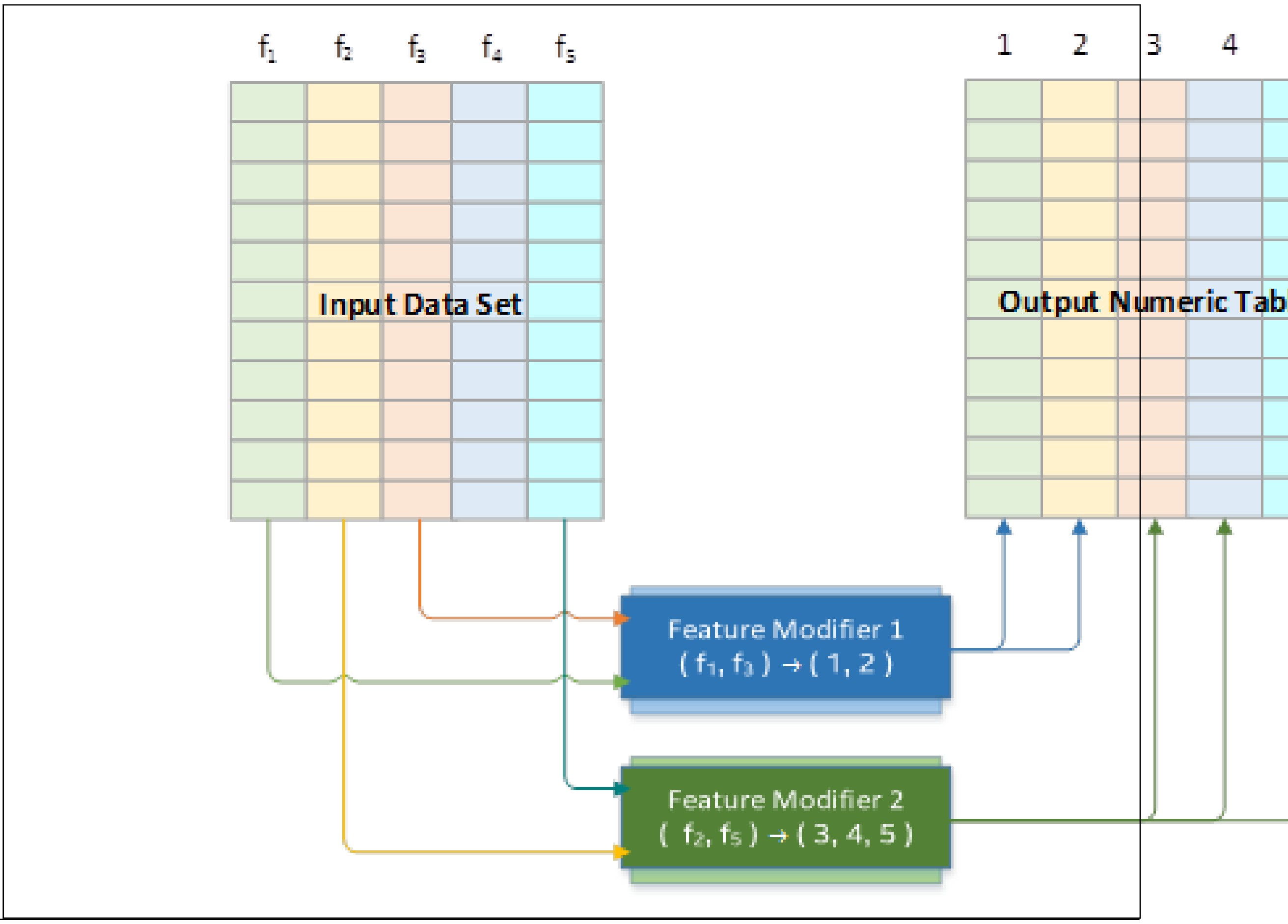
A feature subset may be defined with the functions list(…) , range(…), all(), or allReverse() located in the namespace data_management::features. For example, you can use numerical or string identifiers to refer to the particular feature in the data set. A string identifier may correspond to a feature name (for instance, name in CSV header or in SQL table column name) and numerical one to the index of a feature. The following code block shows several ways to define a feature subset. f1 , f2, and f4 are the names of the respective columns in CSV file or SQL table, and the numbers 0, 2 - 4 are the indices of columns starting from the left one.

```
features::list("f1", "f2")   // String identifiers
features::list(0, 3);        // Numerical identifiers
features::list("f1", 2);     // Mixed identifiers
features::range(0, 4);       // Range of features, the same as list(0,…,4)
features::range("f1", "f4"); // Range with string identifiers
features::all();             // Refer to all features in the data set
features::allReverse()       // Like features::all() but in reverse order


// With STL vector
std::vector<features::IdFactory> fv;
fv.push_back("f2"); fv.push_back(3);
features::list(fv);

// With C++ 11 initializer list
features::list({ "f2", 3, "f1" });
```

We will use the term *input features* to refer to the columns of raw out-of-memory data and the term *output features* for the columns of numeric in-memory data. A feature modifier transforms specified input features subset to the output features. The number of output features is determined by the modifier. A feature modifier is expected to read the values corresponding to specified input features from the i-th row and write modified values to the i-th row of the output numeric table. In general case, feature modifier is able to process arbitrary number of input features to arbitrary number of output features. Let’s assume that we added m modifiers along with the features subsets $F_1, \ldots, F_m$ and the *j*-th modifier has the C_j output columns, where $F_j = (f^j_{i_1}, \ldots, f^j_{i_{n_j}})$ are specified input features of interest, $f^j_i \in \{f_1, \ldots, f_p\}$,

$f_1, \ldots, f_p$ are all possible features, *p* is the number of features in the input data. The output numeric table will contain $C_1 + C_2 + \ldots + C_m$ columns. The *j*-th feature modifier writes result to the columns starting with the index C_k, in particular the first feature modifier writes to the first C_1 columns, and the last to the last C_m columns of the output table. The following picture demonstrates the case of two modifiers. *Feature Modifier 1* reads the features f_1, f_3 from an input data set, performs data transformation and writes the result to the columns 1, 2 in the output numeric table. *Feature Modifier 2* behaves similarly, but processes features f_2, f_5 and has 3 output features.

Feature Modifiers



The oneDAL has several predefined feature modifiers available for CSV and SQL feature managers.

- continuous - parses input values as real numbers, the number of output features is equal to the number of input features.
- categorical - parses input values as categorical features (described above), the number of output features is equal to the number of input features.
- automatic - automatically selects appropriate parsing scheme (continuous or categorical)
- oneHotEncoder - apply one-hot-encoding scheme for input features, the number of output features is equal to the sum of unique values for features in the input data set.

NOTE The current version of the library does not provide predefined feature modifiers for handling ordinal features.

You can implement you own feature modifier by inheriting from FeatureModifierBase and overriding its methods. An example interface of user-defined feature modifier is shown in the code block bellow:

```
class MyFeatureModifier : public modifiers::csv::FeatureModifierBase
{
public:
   virtual void initialize(modifiers::csv::Config &config);
   virtual void apply(modifiers::csv::Context &context);
   virtual void finalize(modifiers::csv::Config &config);
};
```

Use the addModifier(…) method to add the user-defined modifier to the feature manager:

```
ds.getFeatureManager().addModifier(
   features::list(0, 3), modifiers::custom<MyFeatureModifier>()
);
```

Feature modifier’s lifetime consists of three stages:

1. Initialization. Feature manager performs modifier initialization by calling the initialize method. The Config class provides methods to change configuration of the modifier. For example use the Config::setNumberOfOutputFeatures(…) to adjust numbers of output features produced by the modifier. By default, the number of output feature is equal to the number of input features.
2. Applying loop. Feature manager calls the apply method for every row in the input data set, information about the current row is provided via context object. To implement this method, you need to get the input data from the context, carry out desired transformation and write result back to the context output buffer. You can get the output buffer by calling the Context::getOutputBuffer() method, the buffer’s size must be equal to the number of output features you specified at the initialization stage.
3. Finalization. Finalization happens when feature manager calls the finalize method with the same config object passed at the initialization stage. For example, you may use this method to release intermediate buffers when the data transformation is done.

Note that exact set of methods available for Config and Context depends on the data source type. Please refer to Developer Reference to get detailed information about supported methods.

Samples

- mysql/sources/datasource_mysql.cpp
- kdb/sources/datasource_kdb.cpp

Examples

- datasource/simple_csv_feature_modifiers.cpp
- datasource/custom_csv_feature_modifiers.cpp

Data Dictionaries

A data dictionary is the metadata that describes features of a data set. The NumericTableFeature and DataSourceFeature structures describe a particular feature within a dictionary of the associated numeric table and data source respectively. These structures specify:

- Whether the feature is continuous, categorical, or ordinal
- Underlying data types (double, integer, and so on) used to represent feature values

The DataSourceFeature structure also specifies:

- Possible values for a categorical feature
- The feature name

The DataSourceDictionary class is a data dictionary that describes raw data associated with the corresponding data source. The NumericTableDictionary class is a data dictionary that describes in-memory numeric data associated with the corresponding numeric table. Both classes provide generic methods for dictionary manipulation, such as accessing a particular data feature, setting and retrieving the number of features, and adding a new feature. Respective DataSource and NumericTable classes have generic dictionary manipulation methods, such as getDictionary() and setDictionary().

To create a dictionary from the data source context, you can do one of the following:

- Set the doDictionaryFromContext flag in the DataSource constructor.
- Call to the createDictionaryFromContext() method.

Examples

C++:

- datasource/datastructures_aos.cpp
- datasource/datastructures_soa.cpp
- datasource/datastructures_homogen.cpp

Java*:

- datasource/DataStructuresAOS.java
- datasource/DataStructuresSOA.java
- datasource/DataStructuresHomogen.java

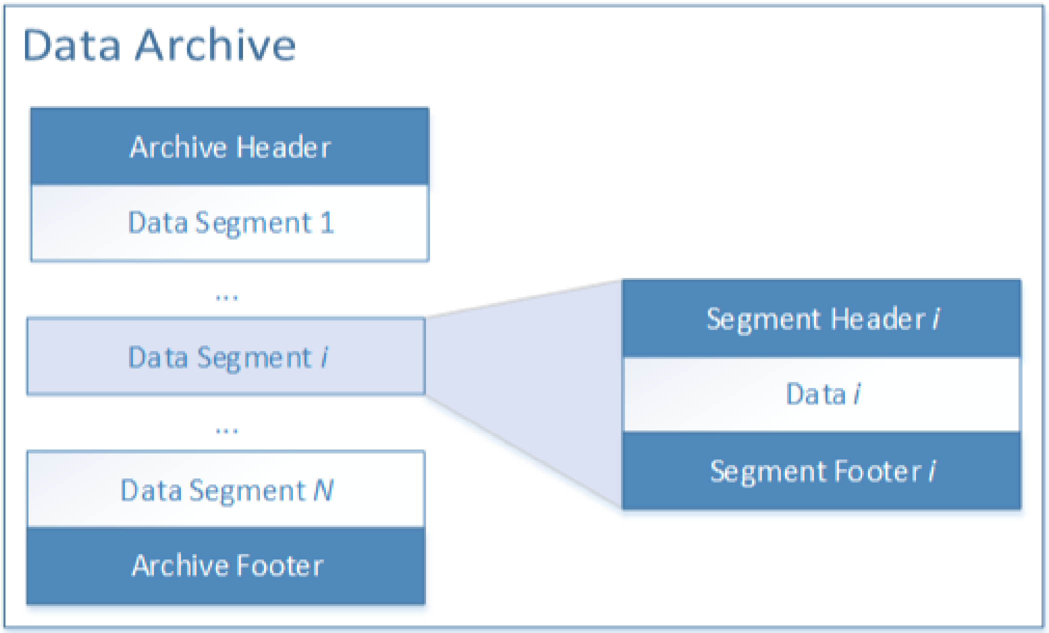
Data Serialization and Deserialization

oneDAL provides interfaces for serialization and deserialization of data objects, which are an essential technique for data exchange between devices and for implementing data recovery mechanisms on a device failure.

The InputDataArchive class provides interfaces for creation of a serialized object archive. The OutputDataArchive class provides interfaces for deserialization of an object from the archive. To reduce network traffic, memory, or persistent storage footprint, you can compress data objects during serialization and decompress them back during deserialization. To this end, provide Compressor and Decompressor objects as arguments for InputDataArchive and OutputDataArchive constructors respectively. For details of compression and decompression, see Data Compression.

A general structure of an archive is as follows:

Data Archive Structure



Headers and footers contain information required to reconstruct the archived object.

All serializable objects, such as numeric tables, a data dictionary, and models, have serialization and deserialization methods. These methods take input archive and output archive, respectively, as method parameters.

Examples

C++: serialization/serialization.cpp

Java: serialization/SerializationExample.java

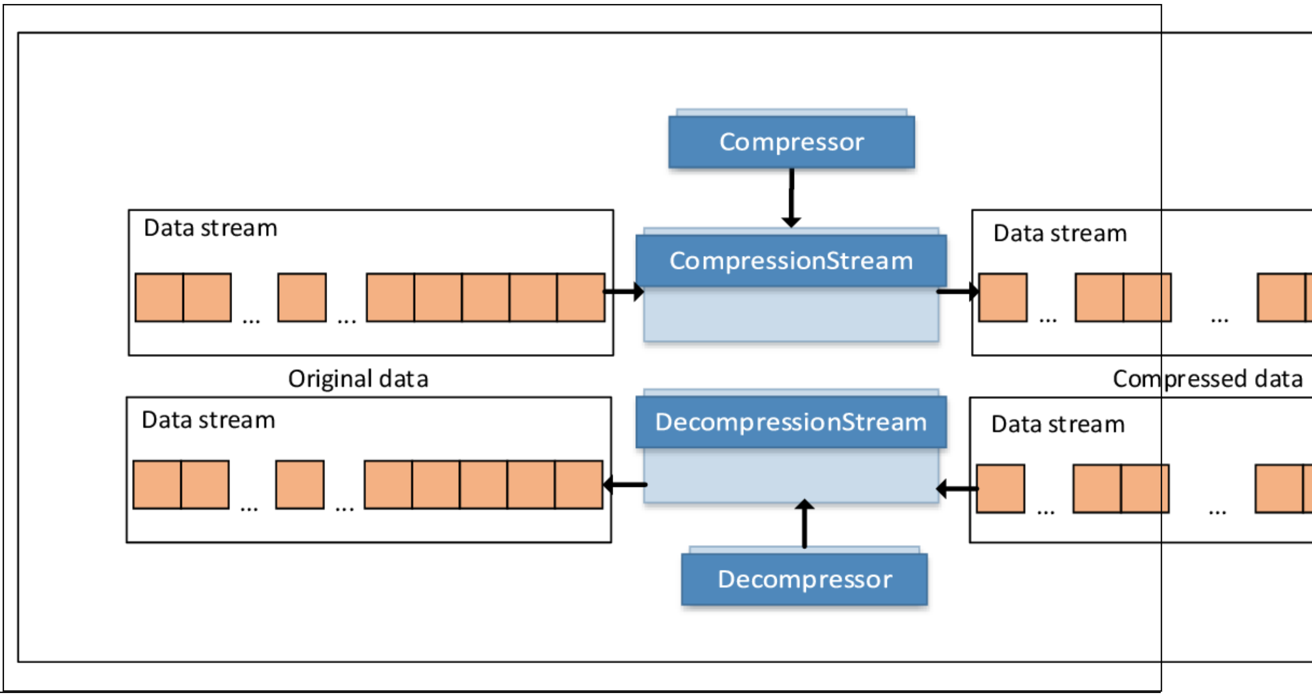
Data Compression

When large amounts of data are sent across devices or need to be stored in memory or in a persistent storage, data compression enables you to reduce network traffic, memory, and persistent storage footprint. oneDAL implements several most popular generic compression and decompression methods, which include ZLIB, LZO, RLE, and BZIP2.

General API for Data Compression and Decompression

The CompressionStream and DecompressionStream classes provide general methods for data compression and decompression. The following diagram illustrates the compression and decompression flow at a high level:

Data Compression and Decompression Flow



To define compression or decompression methods and related parameters, provide Compressor or Decompressor objects as arguments to CompressionStream or DecompressionStream constructors respectively. For more details on Compressor and Decompressor, refer to Compression and Decompression Interfaces.

Use operator << of CompressionStream or DecompressionStream to provide input data for compression or decompression stream. By default, all compression and decompression stream methods allocate the memory required to store results of compression and decompression. For details of controlling memory allocation, refer to Compression and Decompression Interfaces.

The following methods are available to retrieve compressed data stored in CompressionStream:

- Copy compressed data blocks into a contiguous array using the copyCompressedArray() method.

  You can define the data blocks to copy by specifying the number of bytes to copy. The method copies the data from the beginning of the stream and removes the copied data from CompressionStream, so next time you call the copyCompressedArray() method, it copies the next block of data. To copy all the data, before a call to copyCompressedArray(), call the getCompressedBlocksSize() method to get the total size of compressed data in the stream.
- Call the getCompressedBlocksCollection() method.

  Unlike the copyCompressedArray() method, getCompressedBlocksCollection() does not copy compressed blocks but provides a reference to the collection of compressed data blocks. The collection is available until you call the getCompressedBlocksCollection() method next time.

The following methods are available to retrieve decompressed data stored in DecompressionStream:

- Copy decompressed data blocks into a contiguous array using the copyDecompressedArray() method.

  You can define the data blocks to copy by specifying the number of bytes to copy. The method copies the data from the beginning of the stream and removes the copied data from DecompressionStream, so next time you call the copyDecompressedArray() method, it copies the next block of data. To copy all the data, before a call to copyDecompressedArray(), call the getDecompressedBlocksSize() method to get the total size of decompressed data in the stream.
- Call the getDecompressedBlocksCollection() method.

  Unlike the copyDecompressedArray() method, getDecompressedBlocksCollection() does not copy decompressed blocks but provides a reference to the collection of decompressed data blocks. The collection is available until you call the getDecompressedBlocksCollection() method next time.

Compression and Decompression Interfaces

CompressionStream and DecompressionStream classes cover most typical usage scenarios. Therefore, you need to work directly with Compressor and Decompressor objects only in the cases as follows:

- CompressionStream and DecompressionStream classes do not cover your specific usage model.
- You want to control memory allocation and deallocation for results of compression and decompression.
- You need to modify compression and decompression default parameters.

The Compressor and Decompressor classes provide interfaces to supported compression and decompression methods (ZLIB, LZO, RLE, and BZIP2).

Compression and decompression objects are initialized with a set of default parameters. You can modify parameters of a specific compression method by accessing the parameter field of the Compressor or Decompressor object.

To perform compression or decompression using the Compressor or Decompressor classes, respectively, provide input data using the setInputDataBlock() method and call the run() method. This approach requires that you allocate and control the memory to store the results of compression or decompression. In general, it is impossible to accurately estimate the required size of the output data block, and the memory you provide may be insufficient to store results of compression or decompression. However, you can check whether you need to allocate additional memory to continue the run() operation. To do this, use the isOutputDataBlockFull() method. You can also use the getUsedOutputDataBlockSize() method to obtain the size of compressed or decompressed data actually written to the output data block.

You can use your own compression and decompression methods in CompressionStream and DecompressionStream. In this case, you need to override Compressor and Decompressor objects.

Examples

C++:

- compression/compressor.cpp
- compression/compression_batch.cpp
- compression/compression_online.cpp

Java*:

- compression/CompressorExample.java
- compression/CompressionBatch.java
- compression/CompressionOnline.java

Data Model

The Data Model component of the Intel® oneAPI Data Analytics Library (oneDAL) provides classes for model representation. The model mimics the actual data and represents it in a compact way so that you can use the library when the actual data is missing, incomplete, noisy or unavailable.

There are two categories of models in the library: Regression models and Classification models. Regression models are used to predict the values of dependent variables (responses) by observing independent variables. Classification models are used to predict to which sub-population (class) a given observation belongs.

A set of parameters characterizes each model. oneDAL model classes provide interfaces to access these parameters. It also provides the corresponding classes to train models, that is, to estimate model parameters using training data sets. As soon as a model is trained, it can be used for prediction and cross-validation. For this purpose, the library provides the corresponding prediction classes.

Analysis

- K-Means Clustering
- Density-Based Spatial Clustering of Applications with Noise
- Correlation and Variance-Covariance Matrices
- Principal Component Analysis
- Principal Components Analysis Transform
- Singular Value Decomposition
- Association Rules
- Kernel Functions
- Expectation-Maximization
- Cholesky Decomposition
- QR Decomposition
- Outlier Detection
- Distance Matrix
- Distributions
- Engines
- Moments of Low Order
- Quantile
- Quality Metrics
- Sorting
- Normalization

Optimization Solvers

- Optimization Solvers
    - Objective Function
        - Computation
        - Sum of Functions
        - Mean Squared Error Algorithm
        - Objective Function with Precomputed Characteristics Algorithm
        - Logistic Loss

    - Cross-entropy Loss
- Iterative Solver
    - Computation
    - Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm
    - Stochastic Gradient Descent Algorithm
    - Adaptive Subgradient Method
    - Coordinate Descent Algorithm
    - Stochastic Average Gradient Accelerated Method

K-Means Clustering

NOTE K-Means ans K-Means initialization are also available with oneAPI interfaces:

- K-Means
- K-Means initialization

K-Means is among the most popular and simplest clustering methods. It is intended to partition a data set into a small number of clusters such that feature vectors within a cluster have greater similarity with one another than with feature vectors from other clusters. Each cluster is characterized by a representative point, called a centroid, and a cluster radius.

In other words, the clustering methods enable reducing the problem of analysis of the entire data set to the analysis of clusters.

There are numerous ways to define the measure of similarity and centroids. For K-Means, the centroid is defined as the mean of feature vectors within the cluster.

Details

Given the set $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors and a positive integer *k*, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of *kp*-dimensional vectors that minimize the objective function (overall error)

$$ERRORprocessingmath$$

where $d^2(x_i, C)$ is the distance from x_i to the closest center in *C*, such as the Euclidean distance. The vectors $c_1, \ldots, c_k$ are called centroids. To start computations, the algorithm requires initial values of centroids.

Centroid Initialization

Centroids initialization can be done using these methods:

- Choice of first *k* feature vectors from the data set *X*.
- Random choice of *k* feature vectors from the data set using the following simple random sampling draw-by-draw algorithm. The algorithm does the following:

    1. Chooses one of the feature vectors x_i from *X* with equal probability.
    2. Excludes x_i from *X* and adds it to the current set of centers.
    3. Resumes from step 1 until the set of centers reaches the desired size *k*.
- K-Means++ algorithm [Arthur2007], which selects centers with the probability proportional to their contribution to the overall error $ERRORprocessingmath$ according to the following scheme:

    1. Chooses one of the feature vectors x_i from *X* with equal probability.
    2. Excludes x_i from *X* and adds it to the current set of centers *C*.

3. For each feature vector x_i in *X* calculates its minimal distance $d(x_i, C)$ from the current set of centers *C*.
4. Chooses one of the feature vectors x_i from *X* with the probability $ERRORprocessingmath$.
5. Resumes from step 2 until the set of centers *C* reaches the desired size *k*.

- Parallel K-Means++ algorithm [Bahmani2012] that does the following:

  1. Chooses one of the feature vectors x_i from *X* with equal probability.
  2. Excludes x_i from *X* and adds it to the current set of centers *C*.
  3. Repeats *nRounds* times:

     a. For each feature vector x_i from *X* calculates its minimal distance $d(x_i, C)$ from the current set of centers *C*.

     b. Chooses $L = oversamplingFactor \cdot k$ feature vectors x_i from *X* with the probability $ERRORprocessingmath$.

     c. Excludes x_i vectors chosen in the previous step from *X* and adds them to the current set of centers *C*.
  4. For $c_i \in C$ sets w_i to the ratings, the number of points in *X* closer to c_i than to any other point in *C*.
  5. Applies K-Means++ algorithm with weights w_i to the points in *C*, which means that the following probability is used in step:

$$ERRORprocessingmath$$

  The algorithm parameters define the number of candidates *L* selected in each round and number of rounds:

  - Choose *oversamplingFactor* to make $L = O(k)$.
  - Choose nRounds as $ERRORprocessingmath$, where $ERRORprocessingmath$ is the estimation of the goal function when the first center is chosen. [Bahmani2012] recommends to set *nRounds* to a constant value not greater than **8**.

Computation

Computation of the goal function includes computation of the Euclidean distance between vectors $||x_j - m_i||$. The algorithm uses the following modification of the Euclidean distance between feature vectors *a* and *b*: $d(a, b) = d_1(a, b) + d_2(a, b)$, where d_1 is computed for continuous features as

$$d_1(a, b) = \sqrt{\sum_{k=1}^{p1} (a_k - b_k)^2}$$

and d_2 is computed for binary categorical features as

$$d_2(a, b) = \gamma \sqrt{\sum_{k=1}^{p2} (a_k - b_k)^2}$$

In these equations, γ γ weighs the impact of binary categorical features on the clustering, $p1$ is the number of continuous features, and $p2$ is the number of binary categorical features. Note that the algorithm does not support non-binary categorical features.

The K-Means clustering algorithm computes centroids using Lloyd’s method [Lloyd82]. For each feature vector $x_1, \ldots, x_k$, you can also compute the index of the cluster that contains the feature vector.

In some cases, if no vectors are assigned to some clusters on a particular iteration, the iteration produces an empty cluster. It may occur due to bad initialization of centroids or the dataset structure. In this case, the algorithm uses the following strategy to replace the empty cluster centers and decrease the value of the overall goal function:

- Feature vectors, most distant from their assigned centroids, are selected as the new cluster centers. Information about these vectors is gathered automatically during the algorithm execution.
- In the distributed processing mode, most distant vectors from the local nodes are computed (Step 1), stored in *PartialResult*, and collected on the master node (Step 2). For more details, see the *PartialResult* description at Step 1 [Tan2005].

Initialization

The K-Means clustering algorithm requires initialization of centroids as an explicit step. Initialization flow depends on the computation mode. Skip this step if you already calculated initial centroids.

For initialization, the following computation modes are available:

- Batch Processing
- Distributed Processing

Computation

The following computation modes are available:

- Batch Processing
- Distributed Processing

NOTE Distributed mode is not available for oneAPI interfaces and for Python* with DPC++ support.

Examples

oneAPI DPC++

Batch Processing:

- dpc_kmeans_init_dense.cpp
- dpc_kmeans_lloyd_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_lloyd_dense_batch.cpp
- cpp_kmeans_init_dense.cpp

C++ (CPU)

Batch Processing:

- kmeans_dense_batch.cpp
- kmeans_csr_batch.cpp

Distributed Processing:

- kmeans_dense_distr.cpp
- kmeans_csr_distr.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- KMeansDenseBatch.java
- KMeansCSRBatch.java

Distributed Processing

- KMeansDenseDistr.java
- KMeansCSRDistr.java

Python* with DPC++ support

Batch Processing:

- kmeans_batch.py

Python*

Batch Processing:

- kmeans_batch.py

Distributed Processing

- kmeans_spmd.py

Performance Considerations

To get the best overall performance of the K-Means algorithm:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the algorithmFPType class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.
- For the output assignments table, use a homogeneous numeric table of the int type.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Batch Processing

Input

Centroid initialization for K-Means clustering accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for K-Means Initialization (Batch Processing)

Input ID	**Input**
`data`	Pointer to the $nimesp$ numeric table with the data to be clustered.

NOTE The input can be an object of any class derived from `NumericTable`.

Parameters

The following table lists parameters of centroid initialization for K-Means clustering, which depend on the initialization method parameter method.

Algorithm Parameters for K-Means Initialization (Batch Processing)

Parameter	method	Default Value	Description
`algorithmFPType`	any	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	Not applicable	`defaultDense`	Available initialization methods for K-Means clustering: For CPU: • `defaultDense` - uses first nClusters points as initial centroids • `deterministicCSR` - uses first nClusters points as initial centroids for data in a CSR numeric table • `randomDense` - uses random nClusters points as initial centroids • `randomCSR` - uses random nClusters points as initial centroids for data in a CSR numeric table • `plusPlusDense` - uses K-Means++ algorithm [Arthur2007] • `plusPlusCSR` - uses K-Means++ algorithm for data in a CSR numeric table • `parallelPlusDense` - uses parallel K-Means++ algorithm [Bahmani2012] • `parallelPlusCSR` - uses parallel K-Means++ algorithm for data in a CSR numeric table For GPU: • `defaultDense` - uses first nClusters points as initial centroids • `randomDense` - uses random nClusters points as initial centroids
`nClusters`	any	Not applicable	The number of clusters. Required.
`nTrials`	• `parallelPlusDense` • `parallelPlusCSR`	**1**	The number of trails to generate all clusters but the first initial cluster. For details, see [Arthur2007], section 5
`oversamplingFactor`	• `parallelPlusDense` • `parallelPlusCSR`	**0.5**	A fraction of nClusters in each of nRounds of parallel K-Means++. L=nClusters*oversamplingFactor points are sampled in a round. For details, see [Bahmani2012], section 3.3.
`nRounds`	• `parallelPlusDense`	**5**	The number of rounds for parallel K-Means++. (L*nRounds) must be greater than nClusters. For details, see [Bahmani2012], section 3.3.

Parameter	method	Default Value	Description
	- parallelPlusCSR		
engine	any	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally for random numbers generation.

Output

Centroid initialization for K-Means clustering calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for K-Means Initialization (Batch Processing)

Result ID	Result
centroids	Pointer to the $nClusters \times p$ numeric table with the cluster centroids.

> **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Distributed Processing

This mode assumes that the data set is split into `nblocks` blocks across computation nodes.

Parameters

Centroid initialization for K-Means clustering in the distributed processing mode has the following parameters:

Algorithm Parameters for K-Means Initialization (Distributed Processing)

Parameter	Method	Default Valude	Description
computeStep	any	Not applicable	The parameter required to initialize the algorithm. Can be: - `step1Local` - the first step, performed on local nodes. Applicable for all methods. - `step2Master` - the second step, performed on a master node. Applicable for deterministic and random methods only. - `step2Local` - the second step, performed on local nodes. Applicable for `plusPlus` and `parallelPlus` methods only. - `step3Master` - the third step, performed on a master node. Applicable for `plusPlus` and `ParallelPlus` methods only.

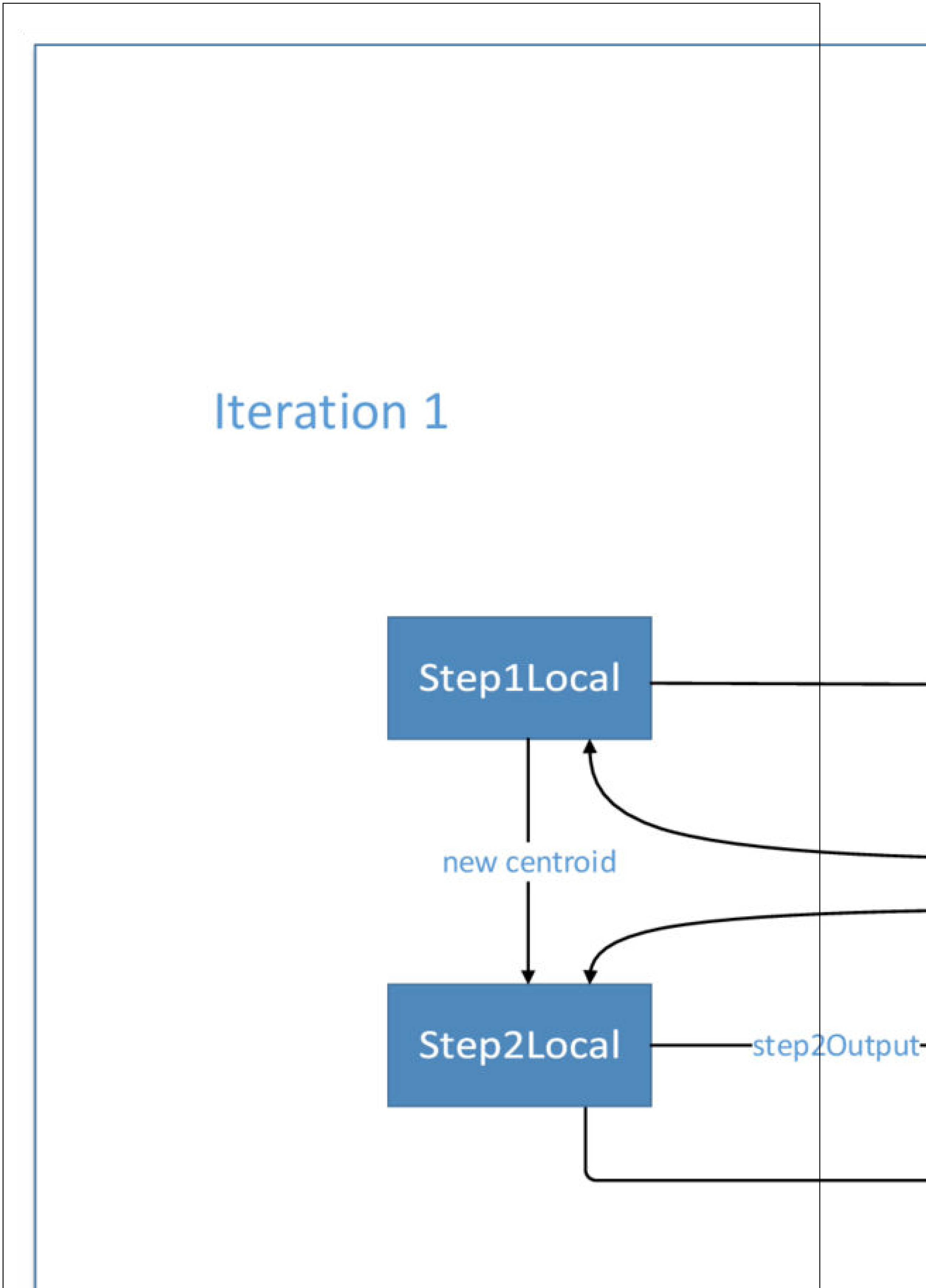
Parameter	Method	Default Valude	Description
			• step4Local - the forth step, performed on local nodes. Applicable for plusPlus and parallelPlus methods only. • step5Master - the fifth step, performed on a master node. Applicable for plusPlus and parallelPlus methods only.
algorithmFPType	any	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	Not applicable	defaultDense	Available initialization methods for K-Means clustering: • defaultDense - uses first nClusters feature vectors as initial centroids • deterministicCSR - uses first nClusters feature vectors as initial centroids for data in a CSR numeric table • randomDense - uses random nClusters feature vectors as initial centroids • randomCSR - uses random nClusters feature vectors as initial centroids for data in a CSR numeric table • plusPlusDense - uses K-Means++ algorithm [Arthur2007] • plusPlusCSR - uses K-Means++ algorithm for data in a CSR numeric table • parallelPlusDense - uses parallel K-Means++ algorithm [Bahmani2012] • parallelPlusCSR - uses parallel K-Means++ algorithm for data in a CSR numeric table For more details, see the algorithm description.
nClusters	any	Not applicable	The number of centroids. Required.
nRowsTotal	any	**0**	The total number of rows in all input data sets on all nodes. Required in the distributed processing mode in the first step.
offset	any	Not applicable	Offset in the total data set specifying the start of a block stored on a given local node. Required.
oversamplingFactor	• parallelPlusDense • parallelPlusCSR	**0.5**	A fraction of nClusters in each of nRounds of parallel K-Means++. $L = \mathrm{nClusters} * \mathrm{oversamplingFactor}$ points are sampled in a round. For details, see [Bahmani2012], section 3.3.

Parameter	Method	Default Valude	Description
nRounds	- parallelPlusDense - parallelPlusCSR	**5**	The number of rounds for parallel K-Means++. $L * \mathrm{nRounds}$ must be greater than nClusters. For details, see [Bahmani2012], section 3.3.
firstIteration	- parallelPlusDense - parallelPlusCSR - plusPlusDense - plusPlusCSR	false	Set to true if step2Local is called for the first time.
outputForStep5Required	- parallelPlusDense - parallelPlusCSR	false	Set to true if step4Local is called on the last iteration of the Step 2 - Step 4 loop.

Centroid initialization for K-Means clustering follows the general schema described in Algorithms.

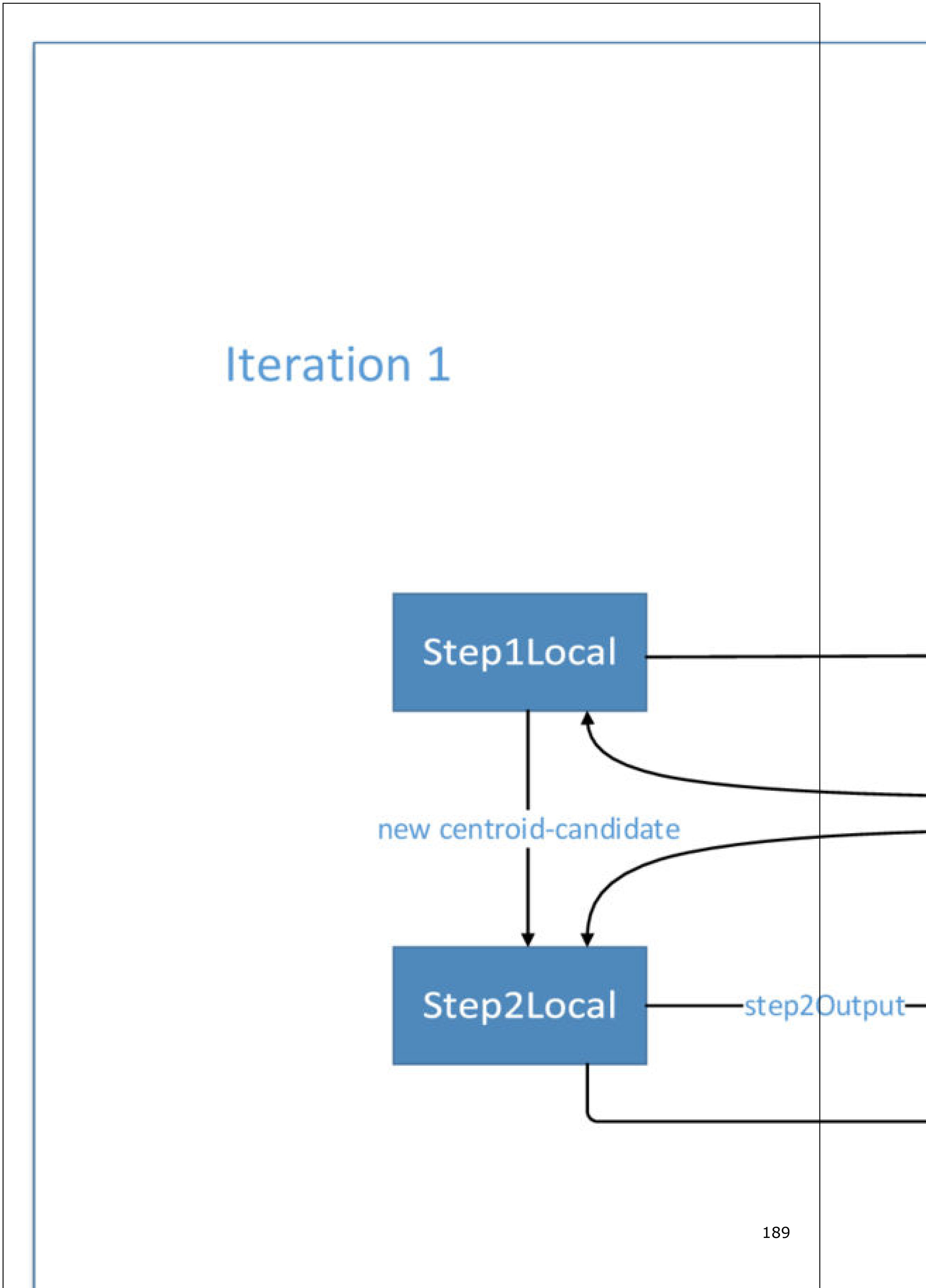
`plusPlus` methods

K-Means Centroid Initialization with plusPlus methods: Distributed Processing



parrallelPlus methods

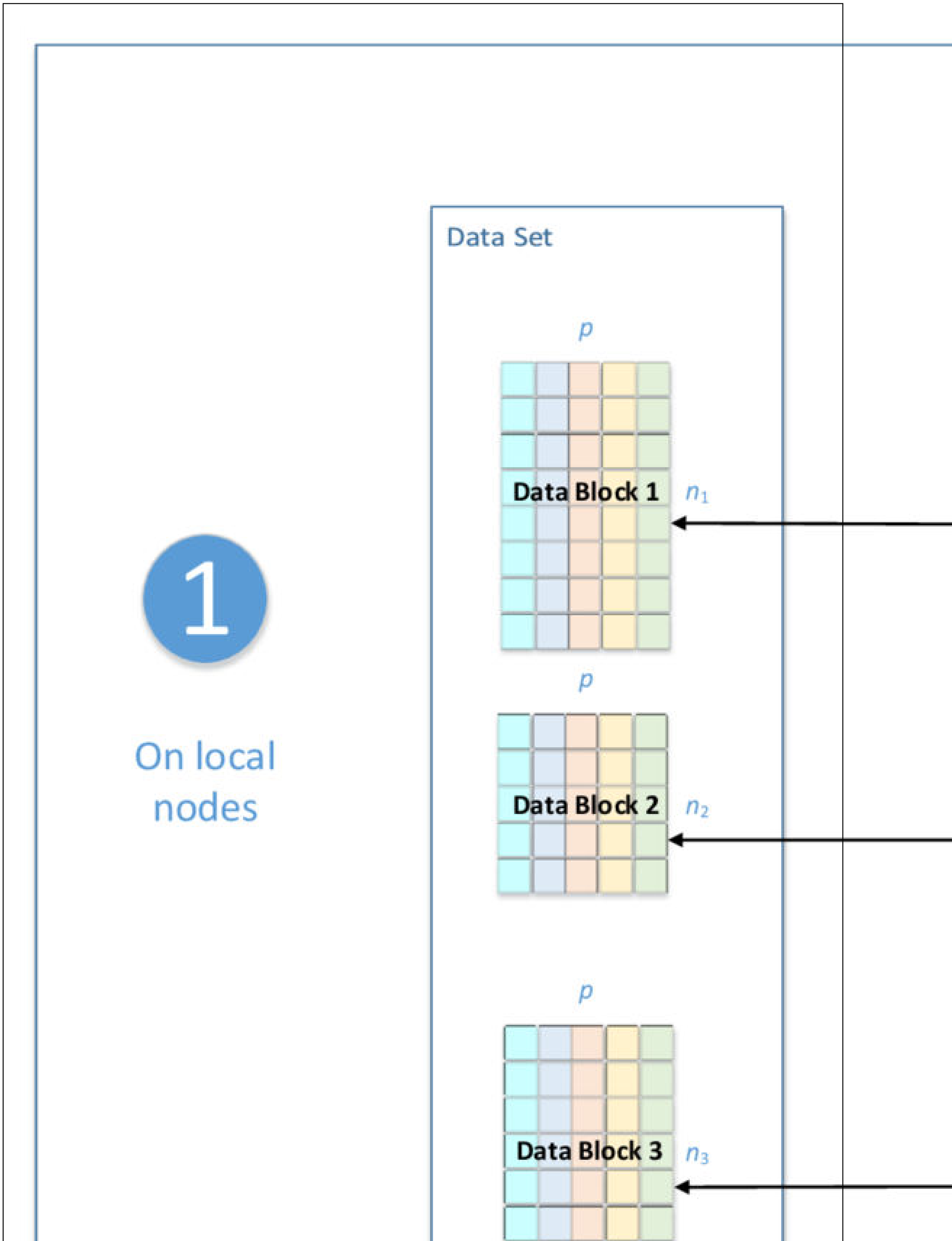
K-Means Centroid Initialization with parrallelPlus methods: Distributed Processing



Step 1 - on Local Nodes (`deterministic`, `random`, `plusPlus`, and `parallelPlus` methods)

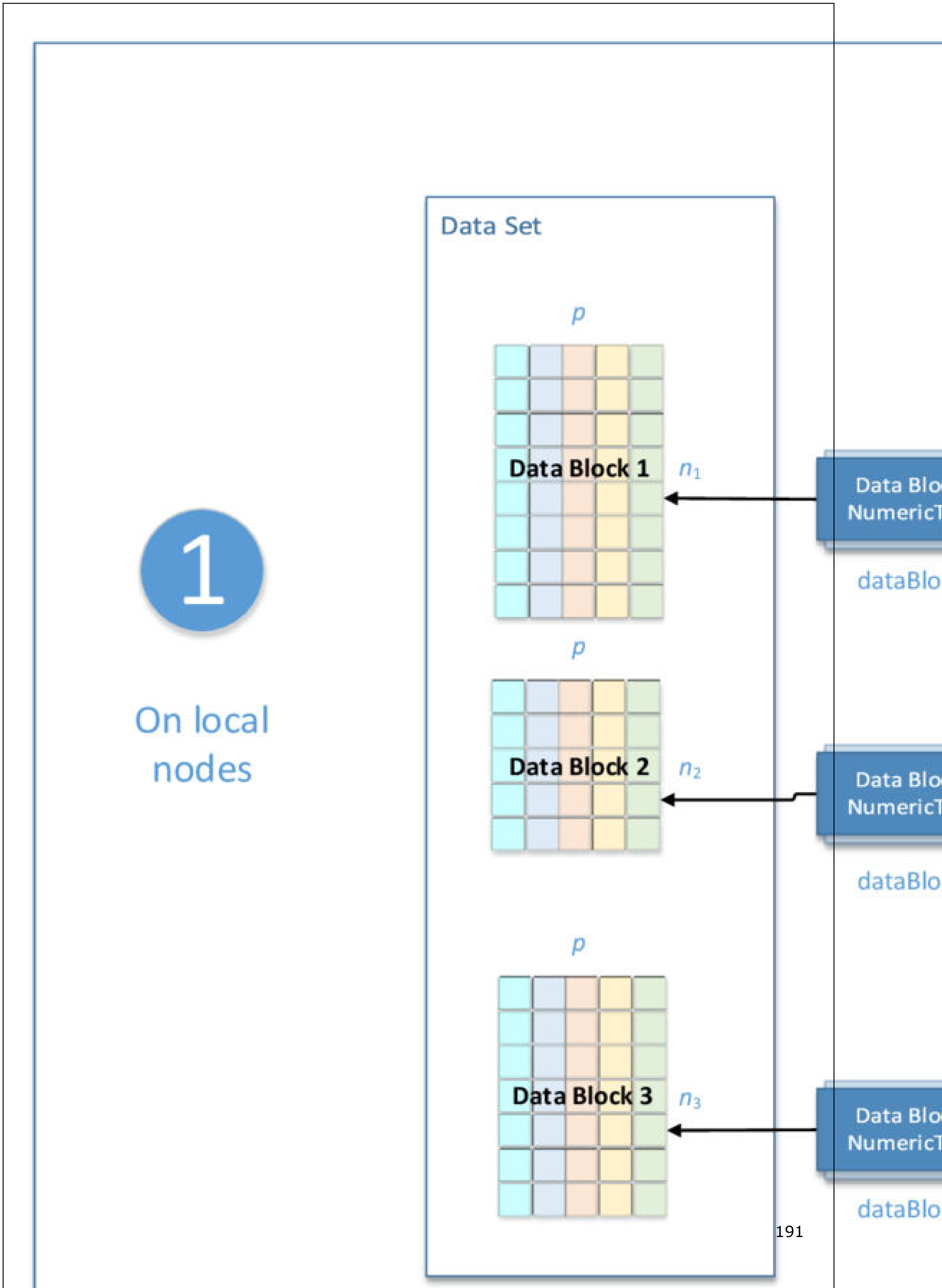
`plusPlus` methods

K-Means Centroid Initialization with plusPlus methods: Distributed Processing, Step 1 - on Local Nodes



`parrallelPlus` methods

K-Means Centroid Initialization with parrallelPlus methods: Distributed Processing, Step 1 - on Local Nodes



In this step, centroid initialization for K-Means clustering accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Initialization (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node. **NOTE** While the input for `defaultDense`, `randomDense`, `plusPlusDense`, and `parallelPlusDense` methods can be an object of any class derived from `NumericTable`, the input for `deterministicCSR`, `randomCSR`, `plusPlusCSR`, and `parallelPlusCSR` methods can only be an object of the `CSRNumericTable` class.

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for K-Means Initialization (Distributed Processing, Step 1)

Result ID	Result
`partialCentroids`	Pointer to the $\mathrm{nClusters} \times p$ numeric table with the centroids computed on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 2 - on Master Node (`deterministic` and `random` methods)

This step is applicable for `deterministic` and `random` methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Initialization (Distributed Processing, Step 2 on Master Node)

Input ID	Input
`partialResuts`	A collection that contains results computed in Step 1 on local nodes (two numeric tables from each local node).

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for K-Means Initialization (Distributed Processing, Step 2 on Master Node)

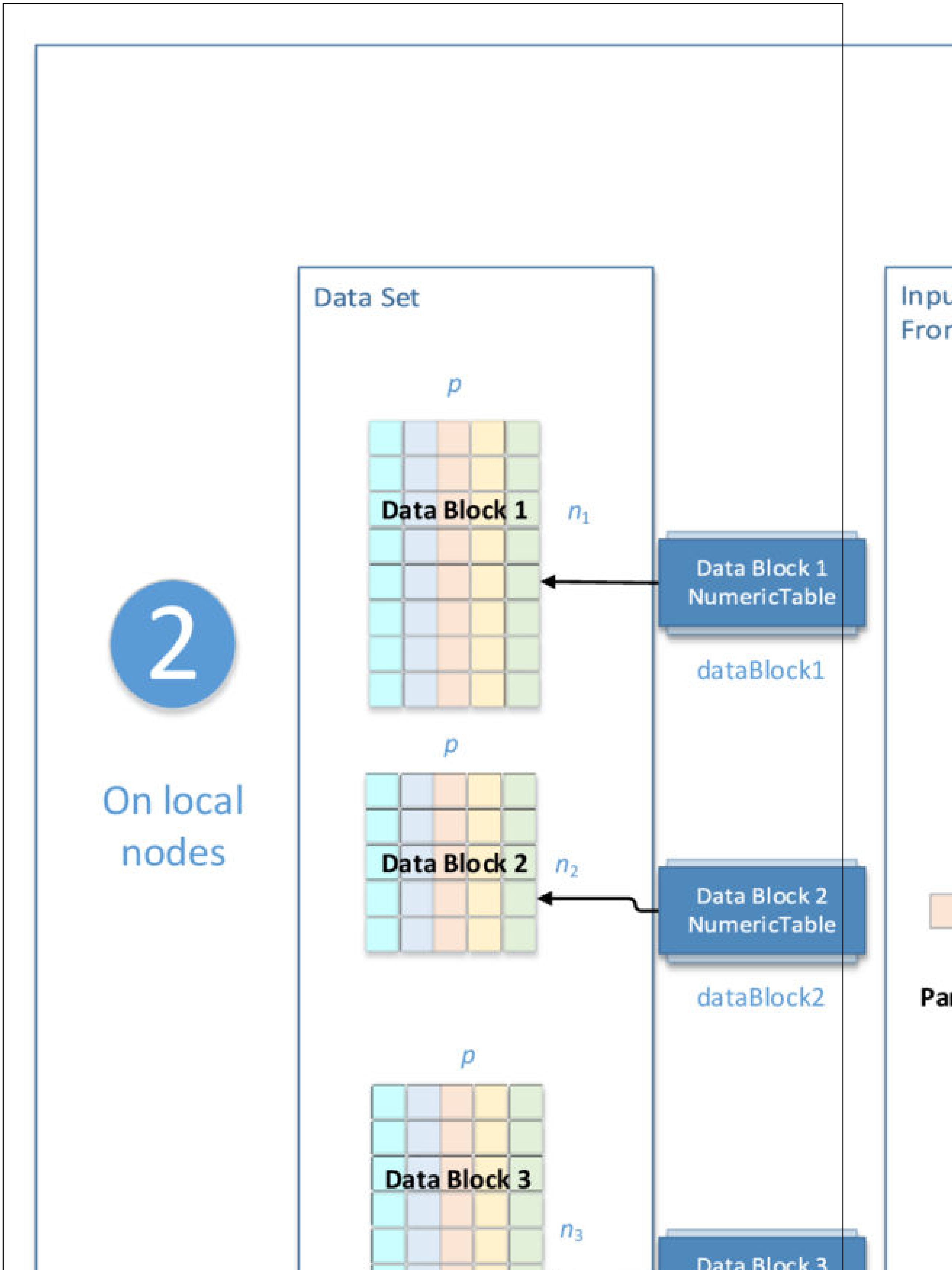
Result ID	Result
`centroids`	Pointer to the $\mathrm{nClusters} \times p$ numeric table with centroids.

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 2 - on Local Nodes (`plusPlus` and `parallelPlus` methods)

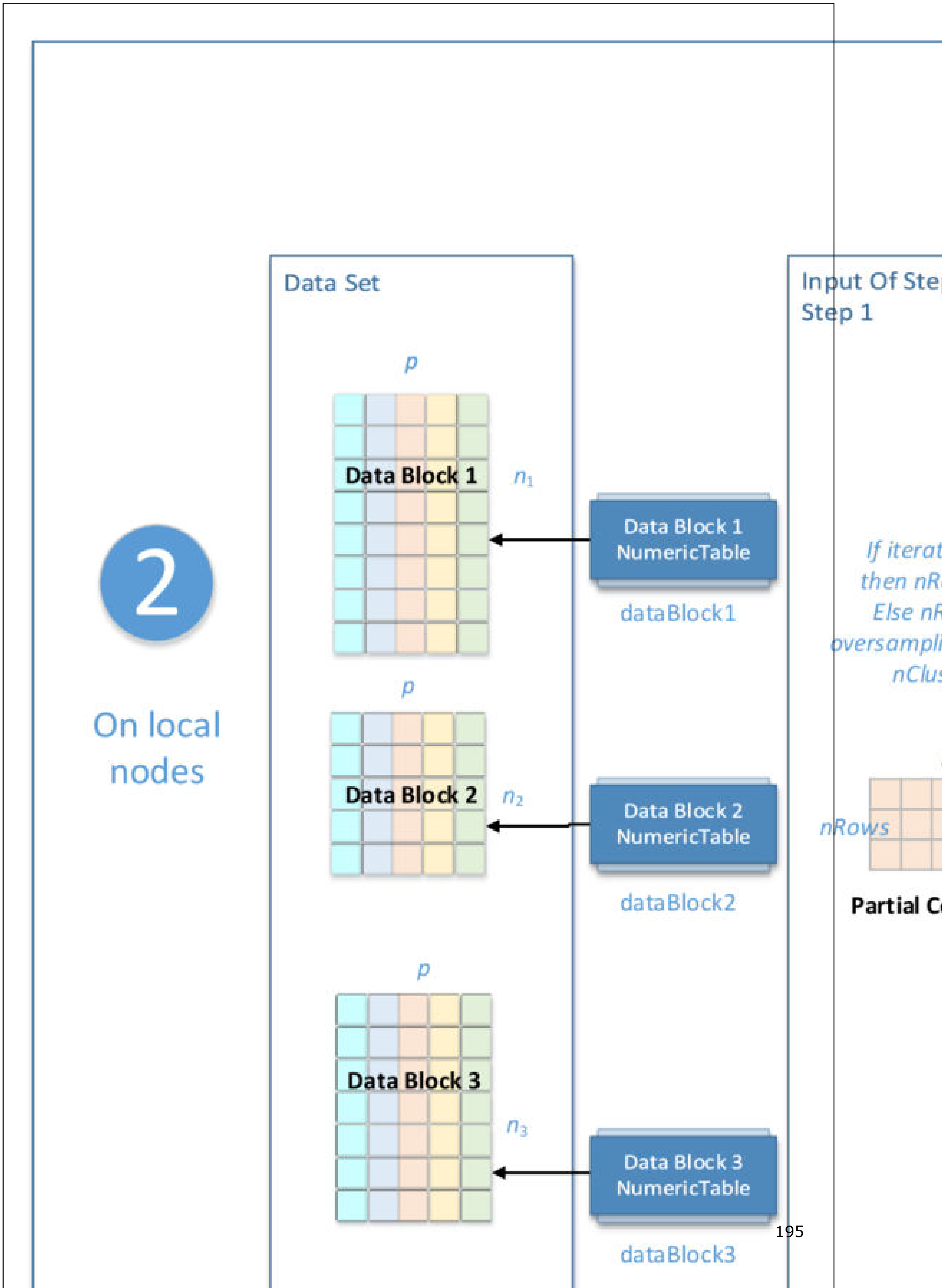
`plusPlus` methods

K-Means Centroid Initialization with plusPlus methods: Distributed Processing, Step 2 - on Local Nodes



`parrallelPlus` methods

K-Means Centroid Initialization with parrallelPlus methods: Distributed Processing, Step 2 - on Local Nodes



This step is applicable for plusPlus and parallelPlus methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Initialization (Distributed Processing, Step 1 on Local Nodes)

Input ID	Input
data	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node. **NOTE** While the input for defaultDense, randomDense, plusPlusDense, and parallelPlusDense methods can be an object of any class derived from NumericTable, the input for deterministicCSR, randomCSR, plusPlusCSR, and parallelPlusCSR methods can only be an object of the CSRNumericTable class.
inputOfStep2	Pointer to the $m \times p$ numeric table with the centroids calculated in the previous steps (Step 1 or Step 4). The value of *m* is defined by the method and iteration of the algorithm: • plusPlus method: $m = 1$ • parallelPlus method: • $m = 1$ for the first iteration of the Step 2 - Step 4 loop • $m = L = \mathrm{nClusters} * \mathrm{oversamplingFactor}$ for other iterations This input can be an object of any class derived from NumericTable, except CSRNumericTable, PackedTriangularMatrix, and PackedSymmetricMatrix.
internalInput	Pointer to the DataCollection object with the internal data of the distributed algorithm used by its local nodes in Step 2 and Step 4. The DataCollection is created in Step 2 when firstIteration is set to true, and then the DataCollection should be set from the partial result as an input for next local steps (Step 2 and Step 4).

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for K-Means Initialization (Distributed Processing, Step 2 on Local Nodes)

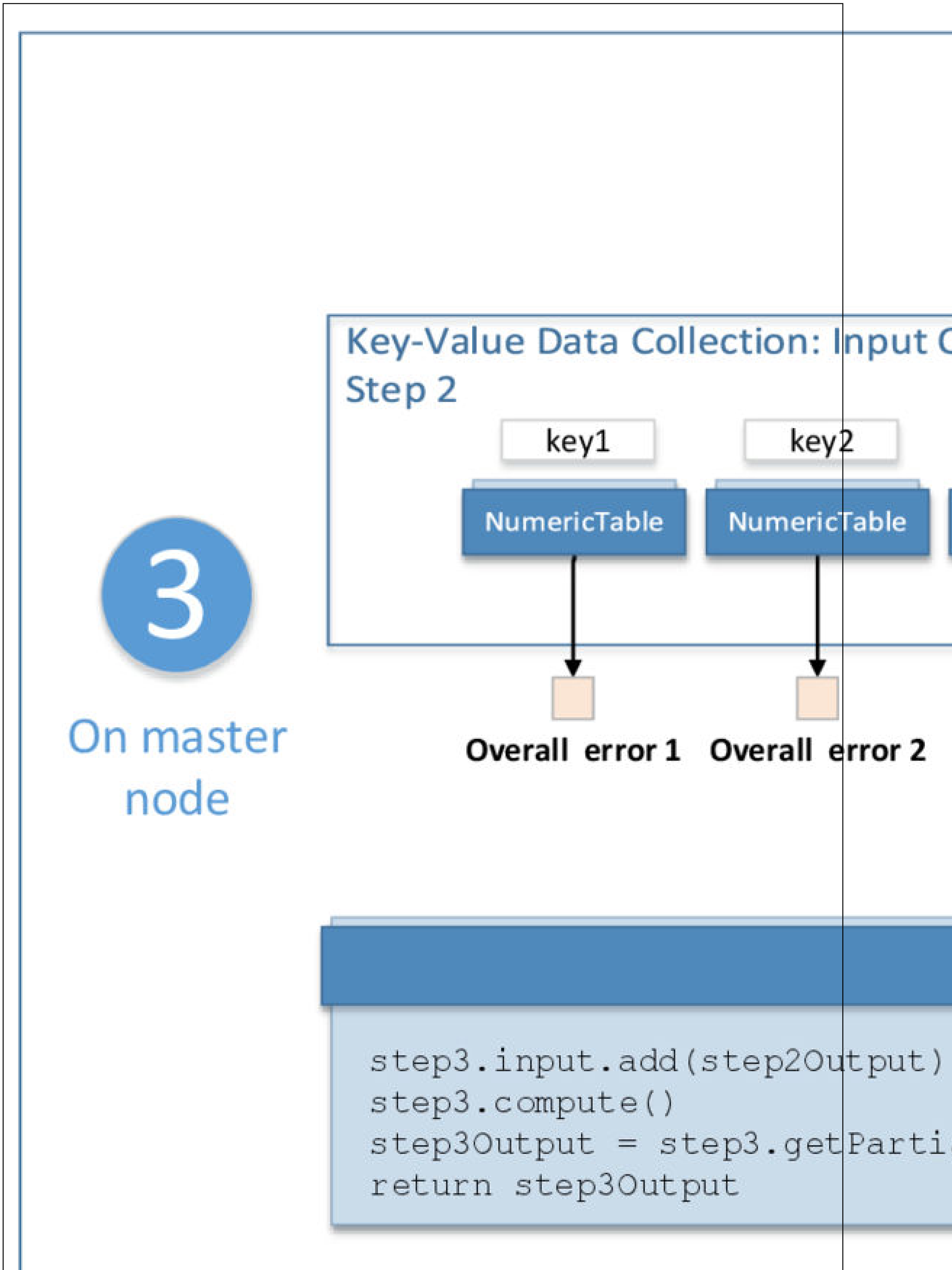
Result ID	Result
outputOfStep2ForStep3	Pointer to the $1imes1$ numeric table that contains the overall error accumulated on the node. For a description of the overall error, see K-Means Clustering Details.
outputOfStep2ForStep5	Applicable for parallelPlus methods only and calculated when outputForStep5Required is set to true. Pointer to the $1 \times m$ numeric table with the ratings of centroid candidates computed on the previous steps and $m = \mathrm{oversamplingFactor} * \mathrm{nClusters} * \mathrm{nRounds} + 1$. For a description of ratings, see K-Means Clustering Details.

NOTE By default, these results are objects of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 3 - on Master Node (`plusPlus` and `parallelPlus` methods)

`plusPlus` methods

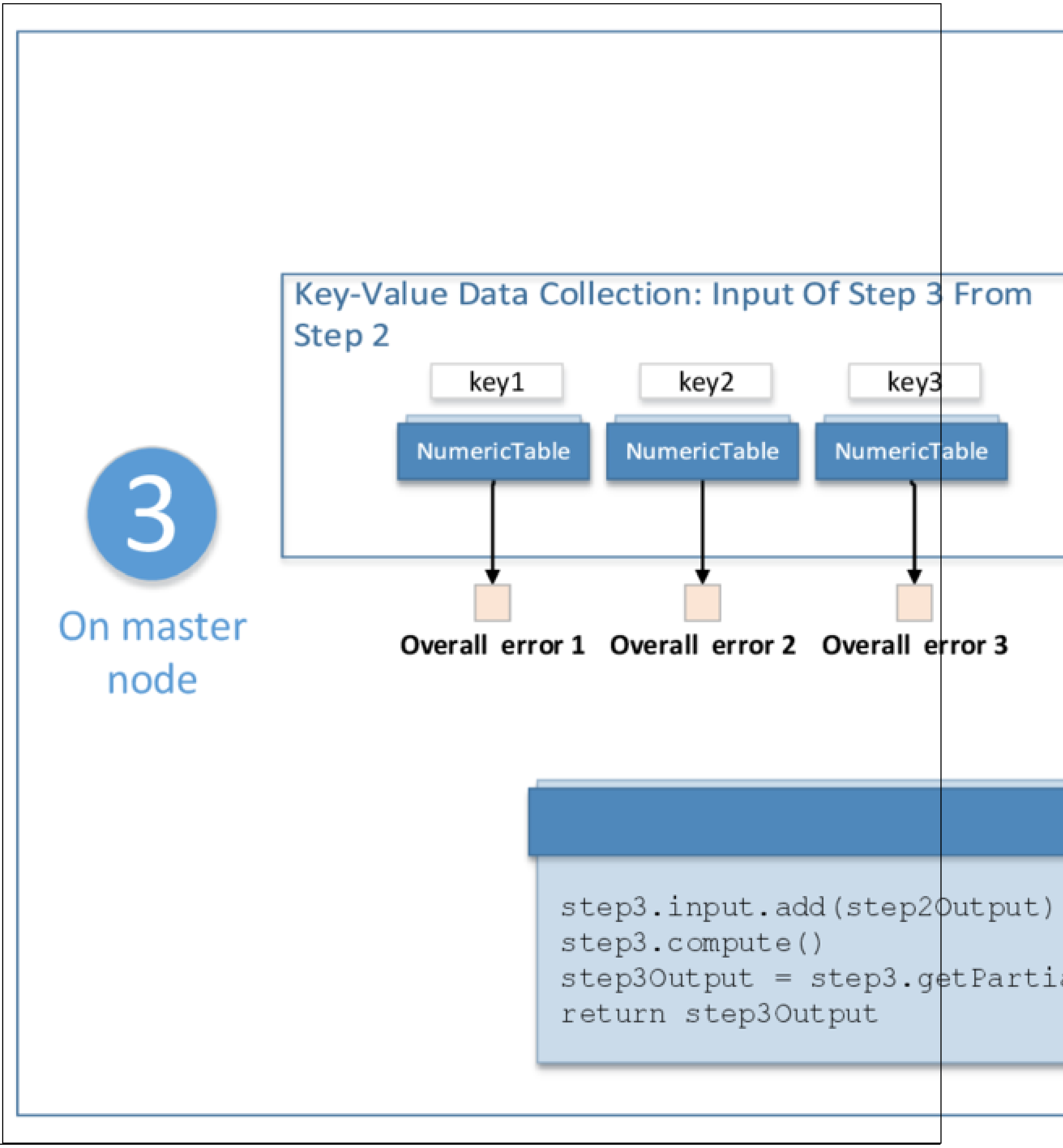
K-Means Centroid Initialization with plusPlus methods: Distributed Processing, Step 3 - on Master Node



```
step3.input.add(step2Output)
step3.compute()
step3Output = step3.getParti
return step3Output
```

`parrallelPlus` methods

K-Means Centroid Initialization with parrallelPlus methods: Distributed Processing, Step 3 - on Master Node



This step is applicable for plusPlus and parallelPlus methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Initialization (Distributed Processing, Step 3)

Input ID	Input
`inputOfStep3FromStep2`	A key-value data collection that maps parts of the accumulated error to the local nodes: *i*-th element of this collection is a numeric table that contains overall error accumulated on the *i*-th node.

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

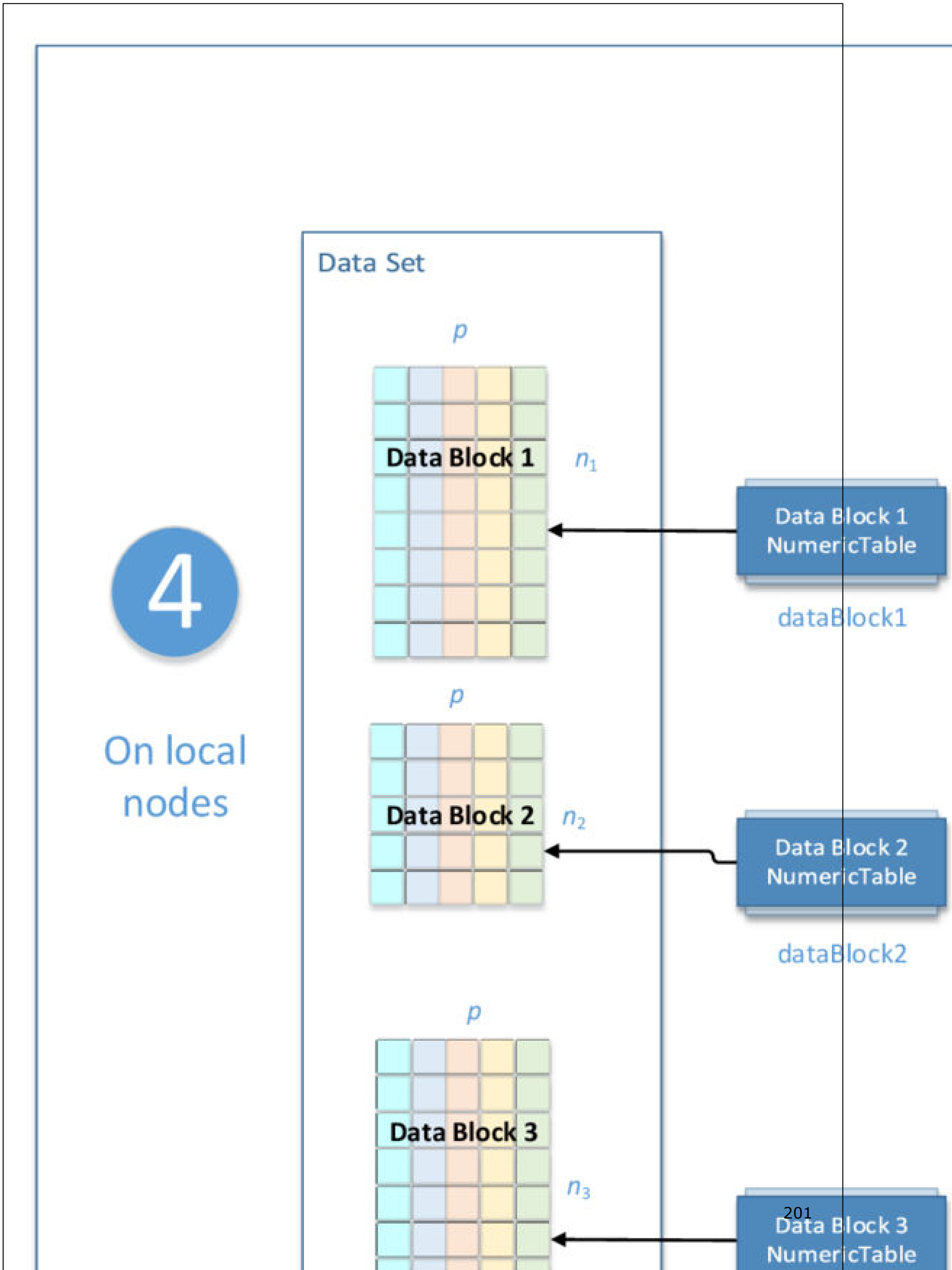
Output for K-Means Initialization (Distributed Processing, Step 3)

Result ID	Result
`outputOfStep3ForStep4`	A key-value data collection that maps the input from Step 4 to local nodes: *i*-th element of this collection is a numeric table that contains the input from Step 4 on the i-th node. Note that Step 3 may produce no input for Step 4 on some local nodes, which means the collection may not contain the *i*-th node entry. The single element of this numeric table $v \leq \Phi_X(C)$, where the overall error $\Phi_X(C)$ calculated on the node. For a description of the overall error, see K-Means Clustering Details. This value defines the probability to sample a new centroid on the *i*-th node.
`outputOfStep3ForStep5`	Applicable for parallelPlus methods only. Pointer to the service data to be used in Step 5.

Step 4 - on Local Nodes (`plusPlus` and `parallelPlus` methods)

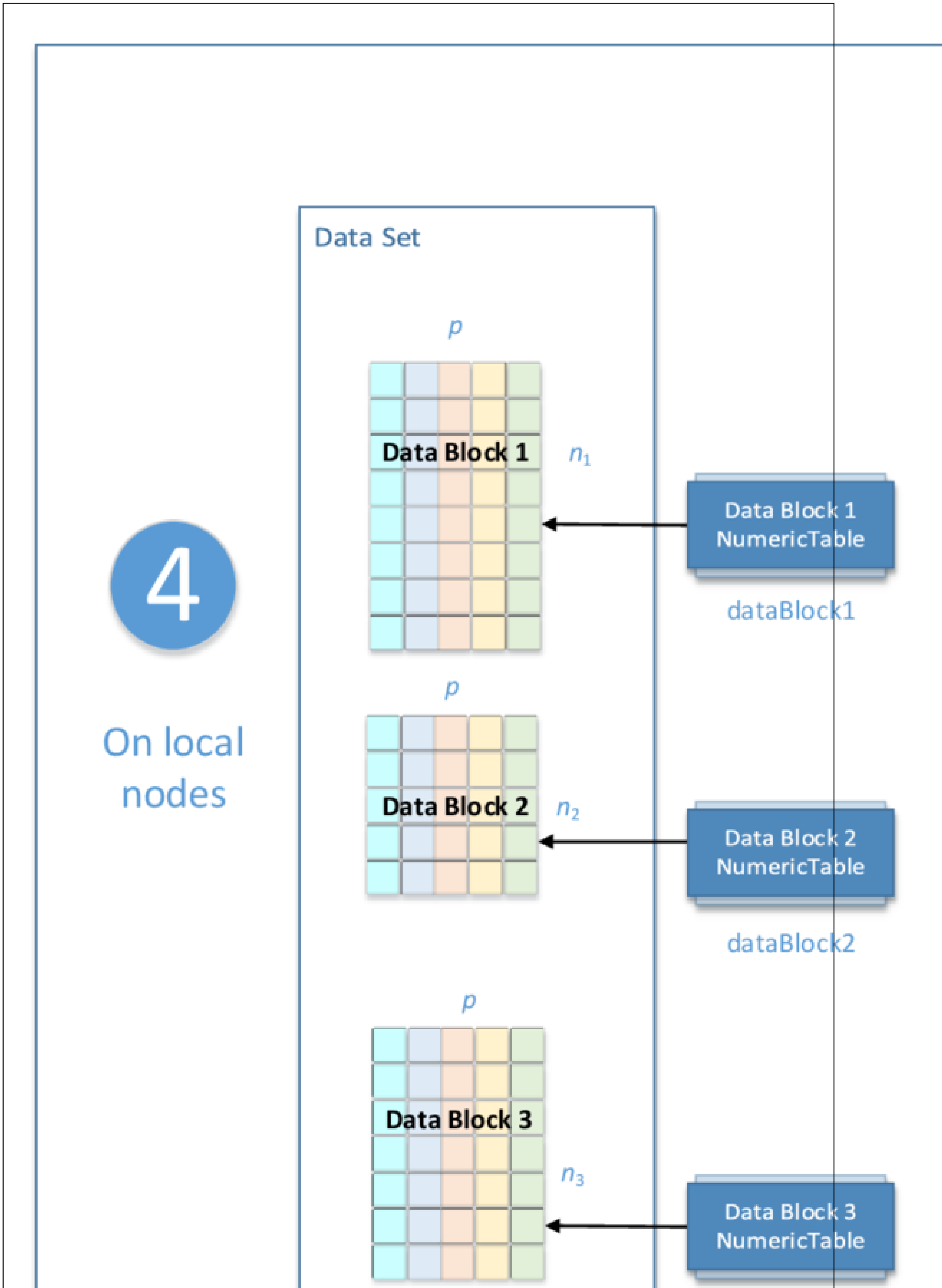
`plusPlus` methods

K-Means Centroid Initialization with plusPlus methods: Distributed Processing, Step 4 - on Local Nodes



`parrallelPlus` methods

K-Means Centroid Initialization with parrallelPlus methods: Distributed Processing, Step 4 - on Local Nodes



This step is applicable for plusPlus and parallelPlus methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Initialization (Distributed Processing, Step 4)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node. **NOTE** While the input for `defaultDense`, `randomDense`, `plusPlusDense`, and `parallelPlusDense` methods can be an object of any class derived from `NumericTable`, the input for `deterministicCSR`, `randomCSR`, `plusPlusCSR`, and `parallelPlusCSR` methods can only be an object of the `CSRNumericTable` class.
`inputOfStep4FromStep3`	Pointer to the $l \times m$ numeric table with the values calculated in Step 3. The value of *m* is defined by the method of the algorithm: • `plusPlus` method: $m = 1$ • `parallelPlus` method: $m \leq L$, $L = \mathrm{nClusters} * \mathrm{oversamplingFactor}$ This input can be an object of any class derived from `NumericTable`, except `CSRNumericTable`, `PackedTriangularMatrix`, and `PackedSymmetricMatrix`.
`internalInput`	Pointer to the `DataCollection` object with the internal data of the distributed algorithm used by its local nodes in Step 2 and Step 4. The `DataCollection` is created in Step 2 when `firstIteration` is set to `true`, and then the `DataCollection` should be set from the partial result as the input for next local steps (Step 2 and Step 4).

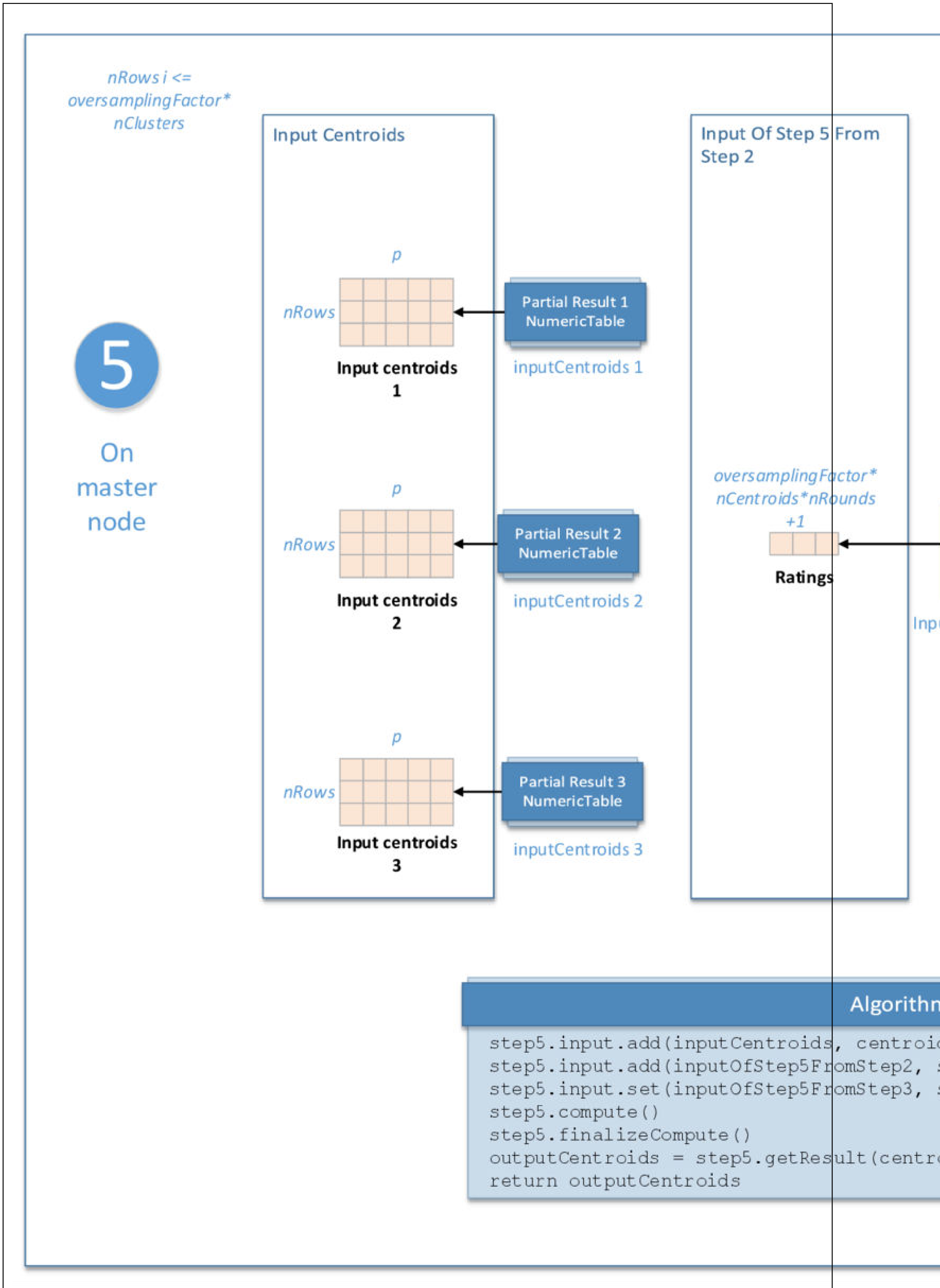
In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for K-Means Initialization (Distributed Processing, Step 4)

Result ID	Result
`outputOfStep4`	Pointer to the $m \times p$ numeric table that contains centroids computed on this local node, where *m* equals to the one in `inputOfStep4FromStep3`. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `CSRNumericTable`, `PackedTriangularMatrix`, and `PackedSymmetricMatrix`.

Step 5 - on Master Node (`parallelPlus` methods)

K-Means Centroid Initialization with parralelPlus methods: Distributed Processing, Step 5 - on Master Node


nRows i <= oversamplingFactor* nClusters
5
On master node
Input Centroids
p
nRows
Input centroids 1
Partial Result 1 NumericTable
inputCentroids 1
p
nRows
Input centroids 2
Partial Result 2 NumericTable
inputCentroids 2
p
nRows
Input centroids 3
Partial Result 3 NumericTable
inputCentroids 3
Input Of Step 5 From Step 2
oversamplingFactor* nCentroids*nRounds +1
Ratings
Inpu
Algorithn

```
step5.input.add(inputCentroids, centroi
step5.input.add(inputOfStep5FromStep2,
step5.input.set(inputOfStep5FromStep3,
step5.compute()
step5.finalizeCompute()
outputCentroids = step5.getResult(centr
return outputCentroids
```

This step is applicable for parallelPlus methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Initialization (Distributed Processing, Step 5)

Input ID	Input
inputCentroids	A data collection with the centroids calculated in Step 1 or Step 4. Each item in the collection is the pointer to $m \times p$ numeric table, where the value of *m* is defined by the method and the iteration of the algorithm: `parallelPlus` method: • $m = 1$ for the data added as the output of Step 1 • $m \leq L$, $L = \text{nClusters} * \text{oversamplingFactor}$ for the data added as the output of Step 4 Each numeric table can be an object of any class derived from `NumericTable`, except `CSRNumericTable`, `PackedTriangularMatrix`, and `PackedSymmetricMatrix`.
`inputOfStep5FromStep2`	A data collection with the items calculated in Step 2 on local nodes. For a detailed definition, see `outputOfStep2ForStep5` above.
`inputOfStep5FromStep3`	Pointer to the service data generated as the output of Step 3 on master node. For a detailed definition, see `outputOfStep3ForStep5` above.

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for K-Means Initialization (Distributed Processing, Step 5)

Result ID	Result
`centroids`	Pointer to the $\text{nClusters} \times p$ numeric table with centroids. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Batch Processing

Algorithm Input

The K-Means clustering algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for K-Means Computaion (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the data to be clustered.

Input ID	Input
inputCentroids	Pointer to the $nClusters \times p$ numeric table with the initial centroids.

NOTE The input for data and inputCentroids can be an object of any class derived from NumericTable.

Algorithm Parameters

The K-Means clustering algorithm has the following parameters:

Algorithm Parameters for K-Means Computaion (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available computation methods for K-Means clustering: For CPU: • defaultDense - implementation of Lloyd's algorithm • lloydCSR - implementation of Lloyd's algorithm for CSR numeric tables For GPU: • defaultDense - implementation of Lloyd's algorithm
nClusters	Not applicable	The number of clusters. Required to initialize the algorithm.
maxIterations	Not applicable	The number of iterations. Required to initialize the algorithm.
accuracyThreshold	**0.0**	The threshold for termination of the algorithm.
gamma	**1.0**	The weight to be used in distance calculation for binary categorical features.
distanceType	euclidean	The measure of closeness between points (observations) being clustered. The only distance type supported so far is the Euclidian distance.
DEPRECATED:assignFlag **USE INSTEAD:** resultsToEvaluate	true	A flag that enables computation of assignments, that is, assigning cluster indices to respective observations.

Parameter	Default Value	Description
resultsToEvaluate	computeCentroids | computeAssignments | computeExactObjectiveFunction	The 64-bit integer flag that specifies which extra characteristics of the K-Means algorithm to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: • computeCentroids for computation centroids. • computeAssignments for computation of assignments, that is, assigning cluster indices to respective observations. • computeExactObjectiveFunction for computation of exact ObjectiveFunction.

Algorithm Output

The K-Means clustering algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm.

Algorithm Output for K-Means Computaion (Batch Processing)

Result ID	Result
centroids	Pointer to the $nClusters \times p$ numeric table with the cluster centroids, computed when computeCentroids option is enabled. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
assignments	Pointer to the $nimes1$ numeric table with assignments of cluster indices to feature vectors in the input data, computed when computeAssignments option is enabled. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
objectiveFunction	Pointer to the $1imes1$ numeric table with the minimum value of the objective function obtained at the last iteration of the algorithm, might be inexact. When computeExactObjectiveFunction option is enabled, exact objective function is computed. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
nIterations	Pointer to the $1imes1$ numeric table with the actual number of iterations done by the algorithm.

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

NOTE You can skip update of centroids and objectiveFunction in the result and compute assignments using original inputCentroids. To do this, set `resultsToEvaluate` flag only to `computeAssignments` and `maxIterations` to zero.

Distributed Processing

This mode assumes that the data set is split into `nblocks` blocks across computation nodes.

Algorithm Parameters

The K-Means clustering algorithm in the distributed processing mode has the following parameters:

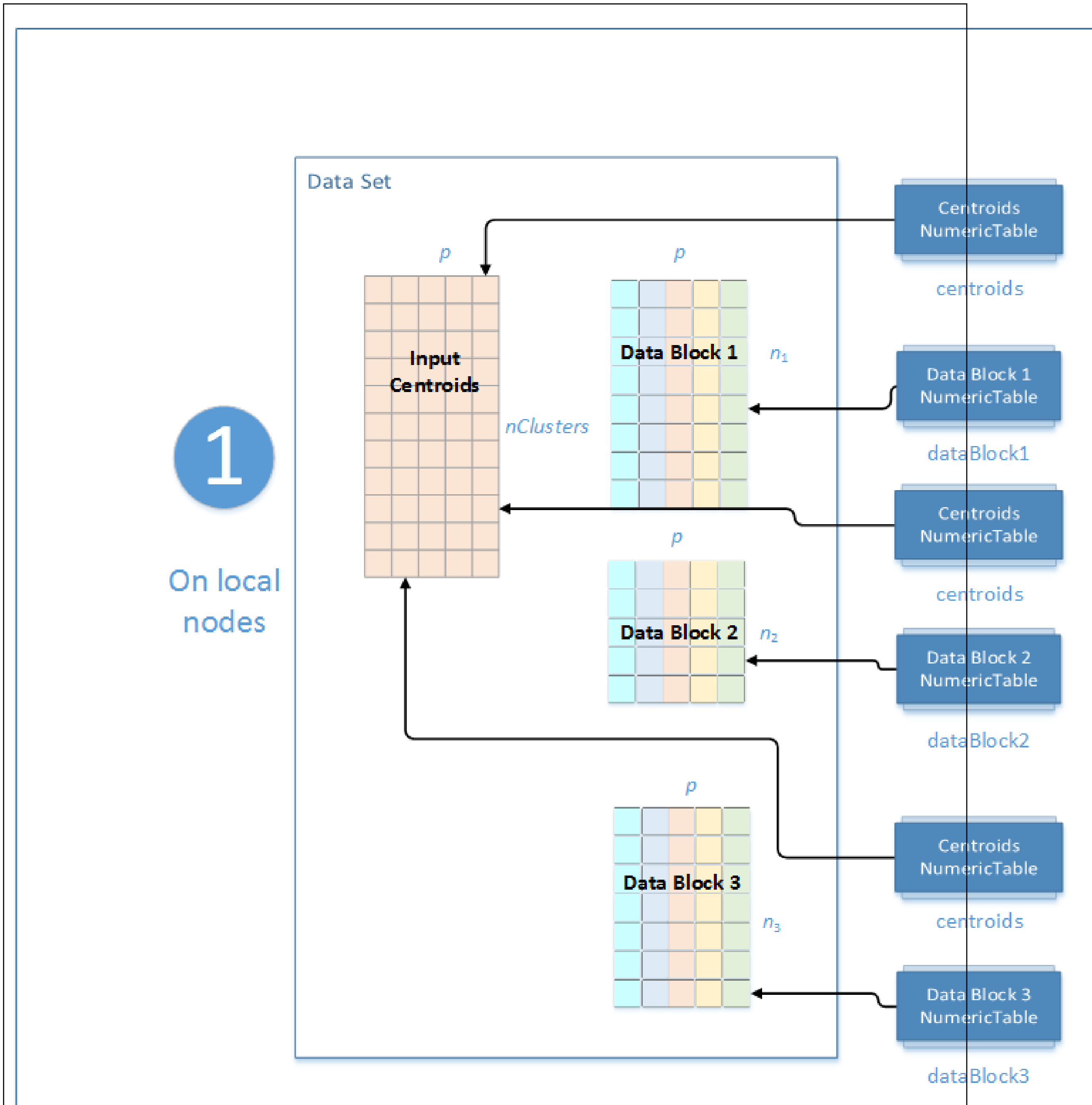
Algorithm Parameters for K-Means Computaion (Distributed Processing)

Parameter	Default Value	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available computation methods for K-Means clustering: • `defaultDense` - implementation of Lloyd's algorithm • `lloydCSR` - implementation of Lloyd's algorithm for CSR numeric tables
`nClusters`	Not applicable	The number of clusters. Required to initialize the algorithm.
`gamma`	**1.0**	The weight to be used in distance calculation for binary categorical features.
`distanceType`	`euclidean`	The measure of closeness between points (observations) being clustered. The only distance type supported so far is the Euclidian distance.
`assignFlag`	`false`	A flag that enables computation of assignments, that is, assigning cluster indices to respective observations.

To compute K-Means clustering in the distributed processing mode, use the general schema described in Algorithms as follows:

Step 1 - on Local Nodes

K-Means Computaion: Distributed Processing, Step 1 - on Local Nodes



Algorithm on *i*-th node

```
alg.input.set(data, dataBlocki)
alg.input.set(inputCentroids, centroids)
If (assignments are requested)
then
     alg.parameter.assignFlag = true
alg.compute()
partialResulti = alg.getPartialResult()
If (assignments are requested)
then
     alg.finalizeCompute()
     assignmentsi = alg.getResult(assignments)
```

In this step, the K-Means clustering algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Computaion (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node. The input can be an object of any class derived from `NumericTable`.
`inputCentroids`	Pointer to the $\mathrm{nClusters} \times p$ numeric table with the initial cluster centroids. This input can be an object of any class derived from NumericTable.

In this step, the K-Means clustering algorithm calculates the partial results and results described below. Pass the `Partial Result ID` or `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Partial Results for K-Means Computaion (Distributed Processing, Step 1)

Partial Result ID	Result
`nObservations`	Pointer to the $\mathrm{nClusters} \times 1$ numeric table that contains the number of observations assigned to the clusters on local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define this result as an object of any class derived from `NumericTable` except `CSRNumericTable`.
`partialSums`	Pointer to the $\mathrm{nClusters} \times p$ numeric table with partial sums of observations assigned to the clusters on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`partialObjectiveFunction`	Pointer to the $1imes1$ numeric table that contains the value of the partial objective function for observations processed on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define this result as an object of any class derived from `NumericTable` except `CSRNumericTable`.
`partialCandidatesDistances`	Pointer to the $\mathrm{nClusters} \times 1$ numeric table that contains the value of the `nClusters` largest objective function for the observations processed on the local node and stored in descending order.

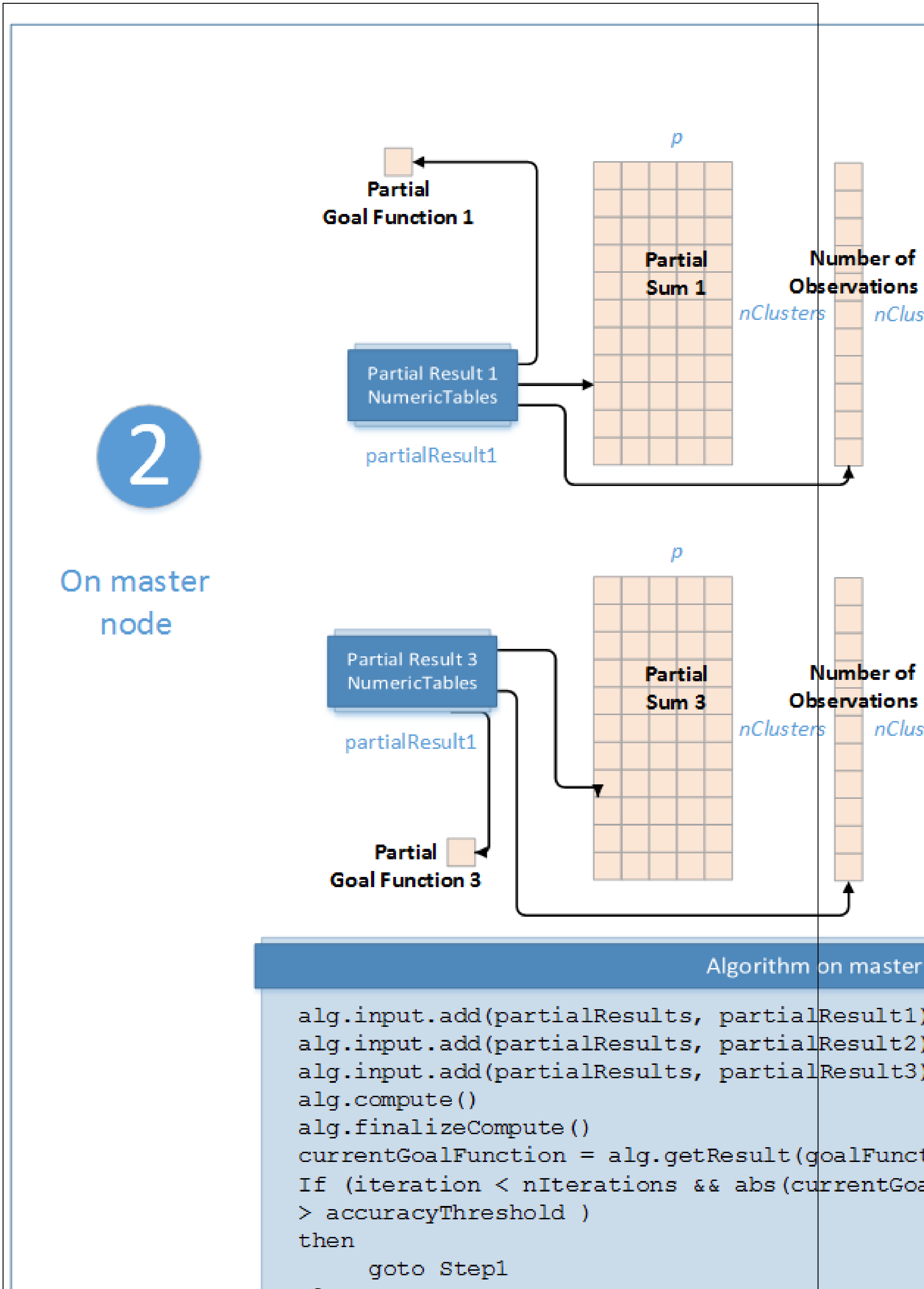
Partial Result ID	Result
	NOTE By default, this result if an object of the `HomogenNumericTable` class, but you can define this result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, `CSRNumericTable`.
`partialCandidatesCentroids`	Pointer to the $\mathrm{nClusters} \times 1$ numeric table that contains the observations of the `nClusters` largest objective function value processed on the local node and stored in descending order of the objective function. **NOTE** By default, this result if an object of the `HomogenNumericTable` class, but you can define this result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, `CSRNumericTable`.

Output for K-Means Computaion (Distributed Processing, Step 1)

Result ID	Result
`assignments`	Use when `assignFlag` = `true`. Pointer to the $n_i \times 1$ numeric table with 32-bit integer assignments of cluster indices to feature vectors in the input data on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define this result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 2 - on Master Node

K-Means Computaion: Distributed Processing, Step 2 - on Master Node



```
alg.input.add(partialResults, partialResult1)
alg.input.add(partialResults, partialResult2)
alg.input.add(partialResults, partialResult3)
alg.compute()
alg.finalizeCompute()
currentGoalFunction = alg.getResult(goalFunct
If (iteration < nIterations && abs(currentGoa
> accuracyThreshold )
then
     goto Step1
```

In this step, the K-Means clustering algorithm accepts the input from each local node described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for K-Means Computaion (Distributed Processing, Step 2)

Input ID	Input
`partialResuts`	A collection that contains results computed in Step 1 on local nodes.

In this step, the K-Means clustering algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for K-Means Computaion (Distributed Processing, Step 2)

Result ID	Result
`centroids`	Pointer to the $\mathrm{nClusters} \times p$ numeric table with centroids. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`objectiveFunction`	Pointer to the $1imes1$ numeric table that contains the value of the objective function. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define this result as an object of any class derived from `NumericTable` except `CSRNumericTable`.

Important The algorithm computes assignments using input centroids. Therefore, to compute assignments using final computed centroids, after the last call to `Step2compute()` method on the master node, on each local node set assignFlag to true and do one additional call to `Step1compute()` and `finalizeCompute()` methods. Always set assignFlag to true and call `finalizeCompute()` to obtain assignments in each step.

NOTE To compute assignments using original `inputCentroids` on the given node, you can use K-Means clustering algorithm in the batch processing mode with the subset of the data available on this node. See Batch Processing for more details.

Density-Based Spatial Clustering of Applications with Noise

Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed in [Ester96]. It is a density-based clustering non-parametric algorithm: given a set of observations in some space, it groups together observations that are closely packed together (observations with many nearby neighbors), marking as outliers observations that lie alone in low-density regions (whose nearest neighbors are too far away).

Details

Given the set $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors (further referred as observations), a positive floating-point number `epsilon` and a positive integer `minObservations`, the problem is to get clustering assignments for each input observation, based on the definitions below [Ester96]:

core observation

An observation *x* is called core observation if at least `minObservations` input observations (including *x*) are within distance `epsilon` from observation *x*;

directly reachable

An observation *y* is directly reachable from *x* if *y* is within distance `epsilon` from core observation*x*. Observations are only said to be directly reachable from core observations.

reachable

An observation *y* is reachable from an observation *x* if there is a path $x_1, \ldots, x_m$ with $x_1 = x$ and $x_m = y$, where each x_{i+1} is directly reachable from x_i. This implies that all observations on the path must be core observations, with the possible exception of *y*.

noise observation

Noise observations are observations that are not reachable from any other observation.

cluster

Two observations *x* and *y* are considered to be in the same cluster if there is a core observation*z*, and *x* and *y* are both reachable from *z*.

Each cluster gets a unique identifier, an integer number from **0** to $\text{total number of clusters} - 1$. Each observation is assigned an identifier of the cluster it belongs to, or **-1** if the observation considered to be a noise observation.

Computation

The following computation modes are available:

- Batch Processing
- Distributed Processing

Examples

C++ (CPU)

Batch Processing:

- dbscan_dense_batch.cpp

Distributed Processing:

- dbscan_dense_distr.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- DBSCANDenseBatch.java

Distributed Processing:

- DBSCANDenseDistr.java

Python* with DPC++ support

Batch Processing:

- dbscan_batch.py

Python*

Batch Processing:

- dbscan_batch.py

Distributed Processing:

- dbscan_spmd.py

Batch Processing

Algorithm Parameters

The DBSCAN clustering algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Batch Processing)

Paramete r	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of DBSCAN algorithm: • `defaultDense` – uses brute-force for neighborhood computation
`epsilon`	Not applicable	The maximum distance between observations lying in the same neighborhood.
`minObservations`	Not applicable	The number of observations in a neighborhood for an observation to be considered as a core one.
`memorySavingMode`	`false`	If flag is set to false, all neighborhoods will be computed and stored prior to clustering. It will require up to $O(\|\text{sum of sizes of all observations' neighborhoods}\|)$ of additional memory, which in worst case can be $O(\|\text{number of observations}\|^2)$. However, in general, performance may be better. **NOTE** On GPU, the `memorySavingMode` flag can only be set to `true`. You will get an error if the flag is set to `false`.
`resultsToCompute`	**0**	The 64-bit integer flag that specifies which extra characteristics of the DBSCAN algorithm to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: • `computeCoreIndices` for indices of core observations • `computeCoreObservations` for core observations

Algorithm Input

The DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for DBSCAN (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the data to be clustered. **NOTE** The input can be an object of any class derived from `NumericTable`.
`weights`	Optional input. Pointer to the $nimes1$ numeric table with weights of observations. **NOTE** The input can be an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`. By default all weights are equal to **1**. **NOTE** This parameter is ignored on GPU.

Algorithm Output

The DBSCAN algorithms calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the result of your algorithm. For more details, see Algorithms.

Algorithm Output for DBSCAN (Batch Processing)

Result ID	Result
`assignments`	Pointer to the $nimes1$ numeric table with assignments of cluster indices to observations in the input data. Noise observations have the assignment equal to **-1**.
`nClusters`	Pointer to the $1imes1$ numeric table with the total number of clusters found by the algorithm.
`coreIndices`	Pointer to the numeric table with **1** column and arbitrary number of rows, containing indices of core observations.
`coreObservations`	Pointer to the numeric table with *p* columns and arbitrary number of rows, containing core observations.

NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Distributed Processing

This mode assumes that the data set is split into `nBlocks` blocks across computation nodes.

To compute DBSCAN algorithm in the distributed processing mode, use the general schema described in Algorithms with the following steps:

Step 1 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of DBSCAN algorithm: • `defaultDense` – uses brute-force for neighborhood computation
`blockIndex`	Not applicable	Unique identifier of block initially passed for computation on the local node.
`nBlocks`	Not applicable	The number of blocks initially passed for computation on all nodes.

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 1)

Input ID	Input
`step1Data`	Pointer to the $nimesp$ numeric table with the observations to be clustered. **NOTE** The input can be an object of any class derived from NumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the `Partial Result ID` as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 1)

Partial Result ID	Result
`partialOrder`	Pointer to the $n \times 2$ numeric table containing information about observations: identifier of initial block and index in initial block. This information will be required to reconstruct initial blocks after transferring observations among nodes. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 2 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of DBSCAN algorithm: • `defaultDense` – uses brute-force for neighborhood computation
`blockIndex`	Not applicable	Unique identifier of block initially passed for computation on the local node.
`nBlocks`	Not applicable	The number of blocks initially passed for computation on all nodes.

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 2)

Input ID	Input
`partialData`	Pointer to the collection of numeric tables with *p* columns and arbitrary number of rows, containing observations to be clustered. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable`.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the `Partial Result ID` as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 2)

Partial Result ID	Result
`boundingBox`	Pointer to the $2 \times p$ numeric table containing bounding box of input observations: first row contains minimum value of each feature, second row contains maximum value of each feature. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 3 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Paramete r	Default Valude	Description
algorith mFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultD ense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation
leftBloc ks	Not applicable	The number of blocks that will process observations with value of selected split feature smaller than selected split value.
rightBlo cks	Not applicable	The number of blocks that will process observations with value of selected split feature greater than selected split value.

In this step, the DBSCAN algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 3)

Input ID	Input
partialDat a	Pointer to the collection of numeric tables with *p* columns and arbitrary number of rows, containing observations to be clustered. **NOTE** The input can be an object of any class derived from DataCollection. The numeric tables in the collection can be an object of any class derived from NumericTable.
step3Parti alBounding Boxes	Pointer to the collection of the $2 \times p$ numeric tables containing bounding boxes computed on step 2 and collected from all nodes participating in current iteration of geometric repartitioning process. **NOTE** The numeric tables in collection can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the Partial Result ID as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 3)

Partial Result ID	Result
split	Pointer to the 1×2 numeric table containing information about split for current iteration of geometric repartitioning.

Partial Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 4 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of DBSCAN algorithm: • `defaultDense` – uses brute-force for neighborhood computation
`leftBlocks`	Not applicable	The number of blocks that will process observations with value of selected split feature smaller than selected split value.
`rightBlocks`	Not applicable	The number of blocks that will process observations with value of selected split feature greater than selected split value.

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 4)

Input ID	Input
`partialData`	Pointer to the collection of numeric tables with *p* columns and arbitrary number of rows, containing observations to be clustered. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable`.
`step4PartialOrders`	Pointer to the collection of numeric table with **2** columns and arbitrary number of rows containing information about observations: identifier of initial block and index in initial block. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Input ID	Input
step4PartialSplits	Pointer to the collection of the 1×2 numeric table containing information about split computed on step 3 and collected from all nodes participating in current iteration of geometric repartitioning process. **NOTE** The input can be an object of any class derived from DataCollection. The numeric tables in the collection can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the Partial Result ID as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 4)

Partial Result ID	Result
partitionedData	Pointer to the collection of (leftBlocks + rightBlocks) numeric tables with *p* columns and arbitrary number of rows containing observations for processing on nodes participating in current iteration of geometric repartitioning. • First leftBlocks numeric tables in collection have the value of selected split feature smaller than selected split value. • Next rightBlocks numeric tables in collection have the value of selected split feature larger than selected split value. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 5 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing, Step 5)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation
blockIndex	Not applicable	Unique identifier of block initially passed for computation on the local node.

Parameter	Default Valude	Description
`nBlocks`	Not applicable	The number of blocks initially passed for computation on all nodes.
`epsilon`	Not applicable	The maximum distance between observations lying in the same neighborhood.

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 5)

Input ID	Input
`partialData`	Pointer to the collection of numeric tables with *p* columns and arbitrary number of rows, containing observations to be clustered. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable`.
`step5PartialBoundingBoxes`	Pointer to the collection of $2 \times p$ numeric table containing bounding boxes computed on step 2 and collected from all nodes. Numeric tables in collection should be ordered by the identifiers of initial block of nodes. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the `Partial Result ID` as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 5)

Partial Result ID	Result
`partitionedHaloData`	Pointer to the collection of `nBlocks` numeric tables with *p* columns and arbitrary number of rows containing observations from current node that should be used as halo observations on each node. Numeric tables in the collection are ordered by the identifiers of initial block of nodes.
`partitionedHaloDataIndices`	Pointer to the collection of `nBlocks` numeric tables with **1** column and arbitrary number of rows containing indices of observations from current node that should be used as halo observations on each node. Numeric tables in the collection are ordered by the identifiers of initial block of nodes.

NOTE By default, this result is an object of the `DataCollection` class. The numeric tables in the collection can be an object of any class derived from `NumericTable`` except for ``PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 6 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing, Step 6)

Parameter	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of DBSCAN algorithm: • `defaultDense` – uses brute-force for neighborhood computation
`blockIndex`	Not applicable	Unique identifier of block initially passed for computation on the local node.
`nBlocks`	Not applicable	The number of blocks initially passed for computation on all nodes.
`epsilon`	Not applicable	The maximum distance between observations lying in the same neighborhood.
`minObservations`	Not applicable	The number of observations in a neighborhood for an observation to be considered as a core.
`memorySavingMode`	`false`	If flag is set to false, all neighborhoods will be computed and stored prior to clustering. It will require up to $O(|\mathrm{sum\ of\ sizes\ of\ neighborhoods}|)$ of additional memory, which in worst case can be $O(|\mathrm{number\ of\ observations}|^2)$. However, in general, performance may be better.

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 6)

Input ID	Input
`partialData`	Pointer to the collection of numeric tables with *p* columns and arbitrary number of rows, containing observations to be clustered. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable`.
`haloData`	Pointer to the collection of numeric tables with *p* columns and arbitrary number of rows, containing halo observations for current node computed on step 5.

Input ID	Input
	NOTE The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable`.
`haloDataIndices`	Pointer to the collection of numeric tables with **1** column and arbitrary number of rows, containing indices for halo observations for current node computed on step 5. Size of this collection should be equal to the size of collection for `haloData`'s `Input ID`. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`haloDataBlocks`	Pointer to the collection of $1imes1$ numeric tables containing identifiers of initial block for halo observations for current node computed on step 5. Size of this collection should be equal to the size of collection for `haloData`'s `Input ID`. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the `Partial Result ID` as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 6)

Partial Result ID	Result
`step6ClusterStructure`	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`step6FinishedFlag`	Pointer to $1imes1$ numeric table containing the flag indicating that the clustering process is finished for current node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Partial Result ID	Result
step6NClusters	Pointer to $1imes1$ numeric table containing the current number of clusters found on the local node. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step6Queries	Pointer to the collection of nBlocks numeric tables with **3** columns and arbitrary number of rows containing clustering queries that should be processed on each node. Numeric tables in collection ordered by the identifiers of initial block of nodes. **NOTE** By default, this result is an object of the DataCollection class. The numeric tables in the collection can be an object of any class derived from NumericTable` except for ``PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 7 - on Master Node

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing, Step 5)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation

In this step, the DBSCAN algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 7)

Input ID	Input
partialFinishedFlags	Pointer to the collection of $1imes1$ numeric table containing the flag indicating that the clustering process is finished collected from all nodes. **NOTE** The input can be an object of any class derived from DataCollection. The numeric tables in the collection can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the results and partial results described below. Pass the `Result ID` as a parameter to the methods that access the result and partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 7)

Partial Result ID	Result
`finishedFlag`	Pointer to $1imes1$ numeric table containing the flag indicating that the clustering process is finished on all nodes. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Step 8 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of DBSCAN algorithm: • `defaultDense` – uses brute-force for neighborhood computation
`blockIndex`	Not applicable	Unique identifier of block initially passed for computation on the local node.
`nBlocks`	Not applicable	The number of blocks initially passed for computation on all nodes.

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 8)

Input ID	Input
`step8InputClusterStructure`	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node. **NOTE** The input can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`step8InputNClusters`	Pointer to $1imes1$ numeric tables containing the current number of clusters found on the local node.

Input ID	Input
	NOTE The input can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`step8PartialQueries`	Pointer to the collection of numeric tables with **3** columns and arbitrary number of rows containing clustering queries that should be processed on the local node collected from all nodes. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the `Partial Result ID` as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 8)

Partial Result ID	Result
`step8ClusterStructure`	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`step8FinishedFlag`	Pointer to $1imes1$ numeric table containing the flag indicating that the clustering process is finished for current node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`step8NClusters`	Pointer to $1imes1$ numeric table containing the current number of clusters found on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Partial Result ID	Result
step8Queries	Pointer to the collection of nBlocks numeric tables with **3** columns and arbitrary number of rows containing clustering queries that should be processed on each node. Numeric tables in collection ordered by the identifiers of initial block of nodes. **NOTE** By default, this result is an object of the DataCollection class. The numeric tables in the collection can be an object of any class derived from NumericTable` except for ``PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 9 - on Master Node

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing, Step 5)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation

In this step, the DBSCAN algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 9)

Input ID	Input
partialNClusters	Pointer to the collection of $1imes1$ numeric table containing the number of clusters found on each node. **NOTE** The input can be an object of any class derived from DataCollection. The numeric tables in the collection can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the results and partial results described below. Pass the Result ID as a parameter to the methods that access the result and partial result of your algorithm. For more details, Algorithms.

Algorithm Output for DBSCAN (Distributed Processing, Step 9)

Result ID	Result
step9NClusters	Pointer to $1imes1$ numeric table containing the number of clusters found on all nodes.

Result ID	Result
	NOTE By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Partial Results for DBSCAN (Distributed Processing, Step 9)

Partial Result ID	Result
clusterOffsets	Pointer to the collection of $1imes1$ numeric tables containing offsets for cluster numeration for each node. Numeric tables with offsets are given in the same order as in the collection for partialNClustersInput ID. **NOTE** By default, this result is an object of the DataCollection class. The numeric tables in the collection can be an object of any class derived from NumericTable` except for ``PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 10 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation
blockIndex	Not applicable	Unique identifier of block initially passed for computation on the local node.
nBlocks	Not applicable	The number of blocks initially passed for computation on all nodes.

In this step, the DBSCAN algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 10)

Input ID	Input
step10InputClusterStructure	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node.

Input ID	Input
	NOTE The input can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step10ClusterOffset	Pointer to $1imes1$ numeric table containing the offset for cluster numeration on the local node computed on step 9. **NOTE** The input can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the Partial Result ID as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 10)

Partial Result ID	Result
step10ClusterStructure	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step10FinishedFlag	Pointer to $1imes1$ numeric table containing the flag indicating that the clusters numeration process is finished for current node. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step10Queries	Pointer to the collection of nBlocks numeric tables with **4** columns and arbitrary number of rows containing clusters numeration queries that should be processed on each node. Numeric tables in collection ordered by the identifiers of initial block of nodes. **NOTE** By default, this result is an object of the DataCollection class. The numeric tables in the collection can be an object of any class derived from NumericTable` except for ``PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 11 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation
blockIndex	Not applicable	Unique identifier of block initially passed for computation on the local node.
nBlocks	Not applicable	The number of blocks initially passed for computation on all nodes.

In this step, the DBSCAN algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 11)

Input ID	Input
step11InputClusterStructure	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node. **NOTE** The input can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step11PartialQueries	Pointer to the collection of numeric tables with **4** columns and arbitrary number of rows containing clusters numeration queries that should be processed on the local node collected from all nodes. **NOTE** The input can be an object of any class derived from DataCollection. The numeric tables in the collection can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the Partial Result ID as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 11)

Partial Result ID	Result
step11ClusterStructure	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node.

Partial Result ID	Result
	NOTE By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step11FinishedFlag	Pointer to $1imes1$ numeric table containing the flag indicating that the clusters numeration process is finished for current node. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step11Queries	Pointer to the collection of nBlocks numeric tables with **4** columns and arbitrary number of rows containing clusters numeration queries that should be processed on each node. Numeric tables in the collection are ordered by the identifiers of initial block of nodes. **NOTE** By default, this result is an object of the DataCollection class. The numeric tables in the collection can be an object of any class derived from NumericTable` except for ``PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 12 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation
blockIndex	Not applicable	Unique identifier of block initially passed for computation on the local node.
nBlocks	Not applicable	The number of blocks initially passed for computation on all nodes.

In this step, the DBSCAN algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 12)

Input ID	Input
step12InputClusterStructure	Pointer to the numeric table with **4** columns and arbitrary number of rows containing information about current clustering state of observations processed on the local node. **NOTE** The input can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
step12PartialOrders	Pointer to the collection of $n \times 2$ numeric tables containing information about observations: identifier of initial block and index in initial block. This information will be required to reconstruct initial blocks after transferring observations among nodes. **NOTE** The input can be an object of any class derived from DataCollection. The numeric tables in the collection can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Algorithm Output

In this step, the DBSCAN algorithms calculates the partial results described below. Pass the Partial Result ID as a parameter to the methods that access the partial result of your algorithm. For more details, Algorithms.

Partial Results for DBSCAN (Distributed Processing, Step 12)

Partial Result ID	Result
assignmentQueries	Pointer to the collection of nBlocks numeric tables with **2** columns and arbitrary number of rows containing clusters assigning queries that should be processed on each node. Numeric tables in the collection are ordered by the identifiers of initial block of nodes.

NOTE By default, this result is an object of the DataCollection class. The numeric tables in the collection can be an object of any class derived from NumericTable` except for ``PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.

Step 13 - on Local Nodes

In this step, the DBSCAN algorithm has the following parameters:

Algorithm Parameters for DBSCAN (Distributed Processing, Step 5)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of DBSCAN algorithm: • defaultDense – uses brute-force for neighborhood computation

In this step, the DBSCAN algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, Algorithms.

Algorithm Input for DBSCAN (Distributed Processing, Step 13)

Input ID	Input
`partialAssignmentQueries`	Pointer to the collection of numeric tables with **2** columns and arbitrary number of rows containing clusters assigning queries that should be processed on the local node collected from all nodes. **NOTE** The input can be an object of any class derived from `DataCollection`. The numeric tables in the collection can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Algorithm Output

In this step, the DBSCAN algorithms calculates the results and partial results described below. Pass the `Result ID` as a parameter to the methods that access the result and partial result of your algorithm. For more details, Algorithms.

Algorithm Output for DBSCAN (Distributed Processing, Step 13)

Result ID	Result
`step13Assignments`	Pointer to the $nimes1$ numeric table with assignments of cluster indices to observations processed on step 1 on the local node. Noise observations have the assignment equal to **-1**. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Partial Results for DBSCAN (Distributed Processing, Step 13)

Partial Result ID	Result
`step13AssignmentsQueries`	Pointer to the numeric table with **2** columns and arbitrary number of rows containing clusters assigning queries that should be processed on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Correlation and Variance-Covariance Matrices

Variance-covariance and correlation matrices are among the most important quantitative measures of a data set that characterize statistical relationships involving dependence.

Specifically, the covariance measures the extent to which variables “fluctuate together” (that is, co-vary). The correlation is the covariance normalized to be between -1 and +1. A positive correlation indicates the extent to which variables increase or decrease simultaneously. A negative correlation indicates the extent to which one variable increases while the other one decreases. Values close to +1 and -1 indicate a high degree of linear dependence between variables.

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to compute the sample means and variance-covariance matrix or correlation matrix:

Correlation and Variance-Covariance Matrices

Statistic	Definition
Means	$M = (m(1), \ldots, m(p))$, where $m\left(j\right) = \frac{1}{n}\sum_i x_{ij}$
Variance-covariance matrix	$Cov = (v_{ij})$, where $v_{ij} = \frac{1}{n-1}\sum_{k=1}^{n}(x_{ki} - m(i))(x_{kj} - m(j))$, $i = \overline{1,p}$, $j = \overline{1,p}$
Correlation matrix	$Cor = (c_{ij})$, where $c_{ij} = \frac{v_{ij}}{\sqrt{v_{ii} \cdot v_{jj}}}$, $i = \overline{1,p}$, $j = \overline{1,p}$

Computation

The following computation modes are available:

- Batch Processing
- Online Processing
- Distributed Processing

Examples

C++ (CPU)

Batch Processing:

- cov_dense_batch.cpp
- cov_csr_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- CovDenseBatch.java
- CovCSRBatch.java

Python* with DPC++ support

Batch Processing:

- covariance_batch.py

Online Processing:

- covariance_streaming.py

Python*

Batch Processing:

- covariance_batch.py

Online Processing:

- covariance_streaming.py

Distributed Processing:

- covariance_spmd.py

Performance Considerations

To get the best overall performance when computing correlation or variance-covariance matrices:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the algorithmFPType class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Batch Processing

Algorithm Input

The correlation and variance-covariance matrices algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Correlation and Variance-Covariance Matrices Algorithm (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table for which the variance-covariance or correlation matrix *C* is computed. While the input for `defaultDense`, `singlePassDense`, or `sumDense` method can be an object of any class derived from `NumericTable`, the input for `fastCSR`, `singlePassCSR`, or `sumCSR` method can only be an object of the `CSRNumericTable` class.

Algorithm Parameters

The correlation and variance-covariance matrices algorithm has the following parameters:

Algorithm Parameters for Correlation and Variance-Covariance Matrices Algorithm (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of correlation and variance-covariance matrices: For CPU:

Parameter	Default Value	Description
		• defaultDense - default performance-oriented method • singlePassDense - implementation of the single-pass algorithm proposed by D.H.D. West • sumDense - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined • fastCSR - performance-oriented method for CSR numeric tables • singlePassCSR - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables • sumCSR - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined For GPU: • defaultDense - default performance-oriented method
outputMatrixType	covarianceMatrix	The type of the output matrix. Can be: • covarianceMatrix - variance-covariance matrix • correlationMatrix - correlation matrix

Algorithm Output

The correlation and variance-covariance matrices algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Correlation and Variance-Covariance Matrices Algorithm (Batch Processing)

Result ID	Result
covariance	Use when outputMatrixType=covarianceMatrix. Pointer to the numeric table with the $pimesp$ variance-covariance matrix. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedTriangularMatrix and CSRNumericTable.
correlation	Use when outputMatrixType=correlationMatrix. Pointer to the numeric table with the $pimesp$ correlation matrix.

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.
`mean`	Pointer to the $1imesp$ numeric table with means. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Online Processing

Online processing computation mode assumes that data arrives in blocks $i = 1, 2, 3, \ldots \text{nblocks}$.

Computation of correlation and variance-covariance matrices in the online processing mode follows the general computation schema for online processing described in Algorithms.

Algorithm Input

The correlation and variance-covariance matrices algorithm in the online processing mode accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Correlation and Variance-Covariance Matrices Algorithm (Online Processing)

Input ID	Input
`data`	Pointer to the numeric table of size $n_i \times p$ that represents the current data block. While the input for `defaultDense`, `singlePassDense`, or `sumDense` method can be an object of any class derived from `NumericTable`, the input for `fastCSR`, `singlePassCSR`, or `sumCSR` method can only be an object of the `CSRNumericTable` class.

Algorithm Parameters

The correlation and variance-covariance matrices algorithm has the following parameters in the online processing mode:

Algorithm Parameters for for Correlation and Variance-Covariance Matrices Algorithm (Online Processing)

Parameter	Default Valude	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of correlation and variance-covariance matrices: defaultDense: default performance-oriented method singlePassDense: implementation of the single-pass algorithm proposed by D.H.D. West sumDense: implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined fastCSR: performance-oriented method for CSR numeric tables singlePassCSR: implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables sumCSR: implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
outputMatrixType	covarianceMatrix	The type of the output matrix. Can be: • covarianceMatrix - variance-covariance matrix • correlationMatrix - correlation matrix
initializationProcedure	Not applicable	The procedure for setting initial parameters of the algorithm in the online processing mode. By default, the algorithm sets the nObservations, sum, and crossProduct parameters to zero.

Partial Results

The correlation and variance-covariance matrices algorithm in the online processing mode calculates partial results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Partial Results for Correlation and Variance-Covariance Matrices Algorithm (Online Processing)

Result ID	Result
nObservations	Pointer to the $1imes1$ numeric table that contains the number of observations processed so far. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except CSRNumericTable.

Result ID	Result
`crossProduct`	Pointer to $pimesp$ numeric table with the cross-product matrix computed so far. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`sum`	Pointer to $1imesp$ numeric table with partial sums computed so far. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Algorithm Output

The correlation and variance-covariance matrices algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Correlation and Variance-Covariance Matrices Algorithm (Online Processing)

Result ID	Result
`covariance`	Use when `outputMatrixType``=``covarianceMatrix`. Pointer to the numeric table with the $pimesp$ variance-covariance matrix. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.
`correlation`	Use when `outputMatrixType``=``correlationMatrix`. Pointer to the numeric table with the $pimesp$ correlation matrix. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.
`mean`	Pointer to the $1imesp$ numeric table with means. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Distributed Processing

This mode assumes that the data set is split into `nblocks` blocks across computation nodes.

Algorithm Parameters

The correlation and variance-covariance matrices algorithm in the distributed processing mode has the following parameters:

Algorithm Parameters for Correlation and Variance-Covariance Matrices Algorithm (Distributed Processing)

Parameter	**Default Valude**	**Description**
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of low order moments: defaultDense: default performance-oriented method singlePassDense: implementation of the single-pass algorithm proposed by D.H.D. West sumDense: implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined fastCSR: performance-oriented method for CSR numeric tables singlePassCSR: implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables sumCSR: implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
`outputMatrixType`	`covarianceMatrix`	The type of the output matrix. Can be: • `covarianceMatrix` - variance-covariance matrix • `correlationMatrix` - correlation matrix

Computation of correlation and variance-covariance matrices follows the general schema described in Algorithms:

Step 1 - on Local Nodes

In this step, the correlation and variance-covariance matrices algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Step 1: Algorithm Input for Correlation and Variance-Covariance Matrices Algorithm (Distributed Processing)

Input ID	Input
`data`	Pointer to the numeric table of size $n_i \times p$ that represents the *i*-th data block on the local node. While the input for `defaultDense`, `singlePassDense`, or `sumDense` method can be an object of any class derived from `NumericTable`, the input for `fastCSR`, `singlePassCSR`, or `sumCSR` method can only be an object of the `CSRNumericTable` class.

In this step, the correlation and variance-covariance matrices algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Step 1: Algorithm Output for Correlation and Variance-Covariance Matrices Algorithm (Distributed Processing)

Result ID	Result
`nObservations`	Pointer to the $1imes1$ numeric table that contains the number of observations processed so far on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `CSRNumericTable`.
`crossProduct`	Pointer to $pimesp$ numeric table with the cross-product matrix computed so far on the local node. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`sum`	Pointer to $1imesp$ numeric table with partial sums computed so far on the local node. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Step 2 - on Master Node

In this step, the correlation and variance-covariance matrices algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Step 2: Algorithm Input for Correlation and Variance-Covariance Matrices Algorithm (Distributed Processing)

Input ID	Input
`partialResults`	A collection that contains results computed in Step 1 on local nodes (`nObservations`, `crossProduct`, and `sum`). **NOTE** The collection can contain objects of any class derived from the `NumericTable` class except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

In this step, the correlation and variance-covariance matrices algorithm calculates the results described in the following table. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Step 2: Algorithm Output for for Correlation and Variance-Covariance Matrices Algorithm (Distributed Processing)

Result ID	Result
`covariance`	Use when `outputMatrixType``=``covarianceMatrix`. Pointer to the numeric table with the $pimesp$ variance-covariance matrix. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.
`correlation`	Use when `outputMatrixType``=``correlationMatrix`. Pointer to the numeric table with the $pimesp$ correlation matrix. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.
`mean`	Pointer to the $1imesp$ numeric table with means. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Principal Component Analysis

NOTE Principal Component Analysis is also available with oneAPI interfaces:

- Principal Components Analysis (PCA)

Principal Component Analysis (PCA) is a method for exploratory data analysis. PCA transforms a set of observations of possibly correlated variables to a new set of uncorrelated variables, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Because all principal components are orthogonal to each other, there is no redundant information. This is a way of replacing a group of variables with a smaller set of new variables. PCA is one of powerful techniques for dimension reduction.

Details

Given a set $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *p*-dimensional feature vectors or a $pimesp$ correlation matrix and the number of principal components p_r, the problem is to compute p_r principal directions (eigenvectors) for the data set. The library returns the transformation matrix *T* of size $p_r \times p$, which contains eigenvectors in the row-major order and a vector of respective eigenvalues in descending order.

oneDAL provides two methods for running PCA:

- SVD
- Correlation

Eigenvectors computed by PCA are not uniquely defined due to sign ambiguity. PCA supports fast ad-hoc "sign flip" technique described in the paper [Bro07]. It modifies the signs of eigenvectors shown below:

$$\hat{T_i} = T_i \cdot sgn(\max_{1 \leq j \leq p} |T_{i,j}|), i = 1, \ldots, p_r$$

where *T*-transformation matrix is computed by PCA, T_i - *i*-th row in the matrix, *j* - column number, *sgn* - signum function:

$$sgn(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0 \end{cases}$$

You can provide these types of input data to the PCA algorithms of the library:

- Original, non-normalized data set
- Normalized data set, where each feature has the zero mean and unit variance
- Correlation matrix

Computation

The following computation modes are available:

- Batch Processing
- Online Processing
- Distributed Processing

Examples

oneAPI DPC++

Batch Processing:

- dpc_pca_cor_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_pca_dense_batch.cpp

C++ (CPU)

Batch Processing:

- pca_cor_dense_batch.cpp
- pca_cor_csr_batch.cpp
- pca_svd_dense_batch.cpp

Online Processing:

- pca_cor_dense_online.cpp
- pca_cor_csr_online.cpp
- pca_svd_dense_online.cpp

Distributed Processing:

- pca_cor_dense_distr.cpp
- pca_cor_csr_distr.cpp
- pca_svd_dense_distr.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- PCACorDenseBatch.java
- PCACorCSRBatch.java
- PCASVDDenseBatch.java

Online Processing:

- PCACorDenseOnline.java
- PCACorCSROnline.java
- PCASVDDenseOnline.java

Distributed Processing:

- PCACorDenseDistr.java
- PCACorCSRDistr.java
- PCASVDDenseDistr.java

Python* with DPC++ support

Batch Processing:

- pca_batch.py

Python*

Batch Processing:

- pca_batch.py

Distributed Processing:

- pca_spmd.py

Performance Considerations

To get the best overall performance of the PCA algorithm:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the algorithmFPType class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

PCA computation using the correlation method involves the correlation and variance-covariance matrices algorithm. Depending on the method of this algorithm, the performance of PCA computations may vary. For sparse data sets, use the methods of this algorithm for sparse data.

Batch Processing

Because the PCA in the batch processing mode performs normalization for data passed as Input ID, to achieve the best performance, normalize the input data set. To inform the algorithm that the data is normalized, set the normalization flag for the input numeric table that represents your data set by calling the setNormalizationFlag() method of the NumericTableIface class.

Because the PCA with the correlation method (defaultDense) in the batch processing mode is based on the computation of the correlation matrix, to achieve the best performance, precompute the correlation matrix. To pass the precomputed correlation matrix to the algorithm, use correlation as Input ID.

Online Processing

PCA with the SVD method (svdDense) in the online processing mode is at least as computationally complex as in the batch processing mode and has high memory requirements for storing auxiliary data between calls to compute(). On the other hand, the online version of the PCA with the SVD method may enable you to hide the latency of reading data from a slow data source. To do this, implement load prefetching of the next data block in parallel with the compute() method for the current block.

Distributed Processing

PCA with the SVD method (svdDense) in the distributed processing mode requires gathering local-node $pimesp$ numeric tables on the master node. When the amount of local-node work is small, that is, when the local-node data set is small, the network data transfer may become a bottleneck. To avoid this situation, ensure that local nodes have a sufficient amount of work. For example, distribute the input data set across a smaller number of nodes.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Batch Processing

Algorithm Input

The PCA algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Principal Component Analysis (Batch Processing)

Input ID	Input
`data`	Use when the input data is a normalized or non-normalized data set. Pointer to the $nimesp$ numeric table that contains the input data set. **NOTE** This input can be an object of any class derived from `NumericTable`.
`correlation`	Use when the input data is a correlation matrix. Pointer to the $pimesp$ numeric table that contains the correlation matrix. **NOTE** This input can be an object of any class derived from `NumericTable` except `PackedTriangularMatrix`.

Algorithm Parameters

The PCA algorithm has the following parameters, depending on the computation method parameter method:

Algorithm Parameters for Principal Component Analysis (Batch Processing)

Parameter	method	Default Value	Description
`algorithmFPType`	`defaultDense` or `svdDense`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	Not applicable	`defaultDense`	Available methods for PCA computation: For CPU: • `defaultDense` - the correlation method • `svdDense` - the SVD method For GPU: • `defaultDense` - the correlation method
`covariance`	`defaultDense`	**SharedPtr<covariance::Batch<algorithmFPType, covariance::defaultDense> >**	The correlation and variance-covariance matrices algorithm to be used for PCA computations with the correlation method.
`normalization`	`svdDense`	**SharedPtr<normalization::zscore::Batch<algorithmFPType, normalization::zscore::defaultDense>>**	The data normalization algorithm to be used for PCA computations with the SVD method.

Parameter	method	Default Value	Description
nComponents	defaultDense, svdDense	**0**	The number of principal components p_r. If it is zero, the algorithm will compute the result for $p_r = p$.
isDeterministic	defaultDense, svdDense	false	If true, the algorithm applies the "sign flip" technique to the results.
resultsToCompute	defaultDense, svdDense	none	The 64-bit integer flag that specifies which optional result to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: • mean • variance • eigenvalue

Algorithm Output

The PCA algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Principal Component Analysis (Batch Processing)

Result ID	Result
eigenvalues	Pointer to the $1 \times p_r$ numeric table that contains eigenvalues in the descending order. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.
eigenvectors	Pointer to the $p_r \times p$ numeric table that contains eigenvectors in the row-major order. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.
means	Pointer to the $1 \times p_r$ numeric table that contains mean values for each feature. Optional. If correlation is provided then the vector is filed with zeroes.
variances	Pointer to the $1 \times p_r$ numeric table that contains mean values for each feature. Optional. If correlation is provided then the vector is filed with zeroes.
dataForTransform	Pointer to key value data collection containing the aggregated data for normalization and whitening with the following key value pairs: • mean - mean

Result ID	Result
	- variance - variance - eigenvalue - eigenvalue If `resultsToCompute` does not contain mean, the dataForTransform means table is NULL. If `resultsToCompute` does not contain variances, the `dataForTransform` variances table is NULL. If `resultsToCompute` does not contain eigenvalues, the `dataForTransform` eigenvalues table is NULL.

Please note the following:

> **NOTE**
>
> - If the function result is not requested through the `resultsToCompute` parameter, the respective element of the result contains a NULL pointer.
> - By default, each numeric table specified by the collection elements is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
> - For the `svdDense` method *n* should not be less than *p*. If $n > p$, svdDense returns an error.

Online Processing

> **NOTE** Online processing mode for Principal Component Analysis is not available on GPU.

Online processing computation mode assumes that data arrives in blocks $i = 1, 2, 3, \ldots, \mathrm{nblocks}$.

PCA computation in the online processing mode follows the general computation schema for online processing described in Algorithms.

Algorithm Input

The PCA algorithm in the online processing mode accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Principal Component Analysis (Online Processing)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the current data block. The input can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The PCA algorithm in the online processing mode has the following parameters, depending on the computation method parameter method:

Algorithm Parameters for Principal Component Analysis (Online Processing)

Parameter	Method	Default Value	Description
`algorithmFPType`	`defaultDense` or `svdDense`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	Not applicable	`defaultDense`	Available computation methods for PCA computation: • `defaultDense` - the correlation method • `svdDense` - the SVD method
`initializationProcedure`	`defaultDense` or `svdDense`	Not applicable	The procedure for setting initial parameters of the algorithm in the online processing mode. • By default, the algorithm with the `defaultDense` method initializes `nObservationsCorrelation`, `sumCorrelation`, and `crossProductCorrelation` with zeros. • By default, the algorithm with the `svdDense` method initializes `nObservationsSVD`, `sumSVD`, and `sumSquaresSVD` with zeros.
`covariance`	`defaultDense`	**SharedPtr<covariance::Online<algorithmFPType, covariance::defaultDense> >**	The correlation and variance-covariance matrices algorithm to be used for PCA computations with the correlation method. For details, see Correlation and Variance-covariance Matrices. Online Processing.

Partial Results

The PCA algorithm in the online processing mode calculates partial results described below. They depend on the computation method. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Correlation method (`defaultDense`)

Partial Results for Principal Component Analysis using Correlation method (Online Processing)

Result ID	Result
`nObservationsCorrelation`	Pointer to the $1imes1$ numeric table with the number of observations processed so far. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `CSRNumericTable`.
`crossProductCorrelation`	Pointer to the $pimesp$ numeric table with the partial cross-product matrix computed so far.

Result ID	Result
	NOTE By default, this table is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.
sumCorrelation	Pointer to the $1imesp$ numeric table with partial sums computed so far. **NOTE** By default, this table is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

SVD method (svdDense)

Partial Results for Principal Component Analysis using SVD method (Online Processing)

Result ID	Result
nObservationsCorrelation	Pointer to the $1imes1$ numeric table with the number of observations processed so far. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except CSRNumericTable.
sumSVD	Pointer to the $1imesp$ numeric table with partial sums computed so far. **NOTE** By default, this table is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.
sumSquaresSVD	Pointer to the $1imesp$ numeric table with partial sums of squares computed so far. **NOTE** By default, this table is an object of the HomogenNumericTable class, but you can define it as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Algorithm Output

The PCA algorithm in the online processing mode calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Principal Component Analysis (Online Processing)

Result ID	Result
eigenvalues	Pointer to the $1imesp$ numeric table that contains eigenvalues in the descending order.

Result ID	Result
eigenvectors	Pointer to the $pimesp$ numeric table that contains eigenvectors in the row-major order.

NOTE By default, these results are objects of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Distributed Processing

NOTE Distributed processing mode for Principal Component Analysis is not available on GPU.

This mode assumes that data set is split in nblocks blocks across computation nodes.

PCA computation in the distributed processing mode follows the general schema described in Algorithms.

Algorithm Parameters

The PCA algorithm in the distributed processing mode has the following parameters, depending on the computation method parameter method:

Algorithm Parameters for Principal Component Analysis (Distributed Processing)

Parameter	Method	Default Value	Description
`computeStep`	`defaultDense` or `svdDense`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node
`algorithmFPType`	`defaultDense` or `svdDense`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	Not applicable	`defaultDense`	Available computation methods for PCA computation: • `defaultDense` - the correlation method • `svdDense` - the SVD method
`covariance`	`defaultDense`	**SharedPtr<covariance::Distributed <computeStep, algorithmFPType,**	The correlation and variance-covariance matrices algorithm to be used for PCA computations with the correlation method. For details, see Correlation and Variance-covariance Matrices. Distributed Processing.

Parameter	Method	Default Value	Description
		covariance::defaultDense> >	

Use the following two-step schema:

Step 1 - on Local Nodes

Correlation method (`defaultDense`)

In this step, the PCA algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Principal Component Analysis using Correlation method (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the Lmath:**i**-th data block on the local node. The input can be an object of any class derived from `NumericTable`.

In this step, PCA calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Principal Component Analysis using Correlation method (Distributed Processing, Step 1)

Result ID	Result
`nObservationsCorrelation`	Pointer to the $1imes1$ numeric table with the number of observations processed so far on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `CSRNumericTable`.
`crossProductCorrelation`	Pointer to the $pimesp$ numeric table with the cross-product matrix computed so far on the local node. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`sumCorrelation`	Pointer to the $1imesp$ numeric table with partial sums computed so far on the local node. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

SVD method (`svdDense`)

In this step, the PCA algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Principal Component Analysis using SVD method (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the Lmath:**i**-th data block on the local node. The input can be an object of any class derived from `NumericTable`.

In this step, PCA calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Principal Component Analysis using SVD method (Distributed Processing, Step 1)

Result ID	Result
`nObservationsCorrelation`	Pointer to the $1imes1$ numeric table with the number of observations processed so far on the local node. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `CSRNumericTable`.
`sumSVD`	Pointer to the $1imesp$ numeric table with partial sums computed so far on the local node. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`sumSquaresSVD`	Pointer to the $1imesp$ numeric table with partial sums of squares computed so far on the local node. **NOTE** By default, this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`auxiliaryDataSVD`	A collection of numeric tables each with the partial result to transmit to the master node for Step 2. **NOTE** The collection can contain objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

Step 2 - on Master Node

Correlation method (`defaultDense`)

In this step, the PCA algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Principal Component Analysis using Correlation method (Distributed Processing, Step 2)

Input ID	Input
partialResults	A collection that contains results computed in Step 1 on local nodes (nObservationsCorrelation, crossProductCorrelation, and sumCorrelation). **NOTE** The collection can contain objects of any class derived from NumericTable except the PackedSymmetricMatrix and PackedTriangularMatrix.

In this step, PCA calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Principal Component Analysis using Correlation method (Distributed Processing, Step 2)

Result ID	Result
eigenvalues	Pointer to the $1imesp$ numeric table that contains eigenvalues in the descending order.
eigenvectors	Pointer to the $pimesp$ numeric table that contains eigenvectors in the row-major order.

NOTE By default, these results are object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

SVD method (svdDense)

In this step, the PCA algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Principal Component Analysis using SVD method (Distributed Processing, Step 2)

Input ID	Input
partialResults	A collection that contains results computed in Step 1 on local nodes (nObservationsSVD, sumSVD, sumSquaresSVD, and auxiliaryDataSVD). **NOTE** The collection can contain objects of any class derived from NumericTable except the PackedSymmetricMatrix and PackedTriangularMatrix.

In this step, PCA calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Principal Component Analysis using SVD method (Distributed Processing, Step 2)

Result ID	Result
eigenvalues	Pointer to the $1imesp$ numeric table that contains eigenvalues in the descending order.

Result ID	Result
eigenvectors	Pointer to the $pimesp$ numeric table that contains eigenvectors in the row-major order.

NOTE By default, these results are object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Principal Components Analysis Transform

The PCA transform algorithm transforms the data set to principal components.

Details

Given a transformation matrix *T* computed by PCA (eigenvectors in row-major order) and data set *X* as input, the PCA Transform algorithm transforms input data set *X* of size $nimesp$ to the data set *Y* of size $n \times p_r$, $pr \leq p$.

Batch Processing

Algorithm Input

The PCA Transform algorithm accepts the input described below. Pass the `` `Input ID` `` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Principal Components Analysis Transform (Batch Processing)

Input ID	Input
data	Use when the input data is a normalized or non-normalized data set. Pointer to the $nimesp$ numeric table that contains the input data set. This input can be an object of any class derived from `NumericTable`.
eigenvectors	Principal components computed using the PCA algorithm. Pointer to the $p_r \times p$ numeric table $(p_r \leq p)$. You can define it as an object of any class derived from `NumericTable`, except for `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
dataForTransform	Optional. Pointer to the key value-data collection containing the following data for PCA. The collection contains the following key-value pairs: mean means variance variances eigenvalue eigenvalues

Input ID	Input
	NOTE - If you do not provide the collection, the library will not apply the corresponding centering, normalization or whitening operation. - If one of the numeric tables in collection is `NULL`, the corresponding operation will not be applied: centering for means, normalization for variances, whitening for eigenvalues. - If mean or variance tables exist, it should be a pointer to the $1imesp$ numeric table. - If eigenvalue table is not `NULL`, it is the pointer to ($1 \times \text{nColumns}$) numeric table, where the number of the columns is greater than or equal to `nComponents`.

Algorithm Parameters

The PCA Transform algorithm has the following parameters:

Algorithm Parameters for Principal Components Analysis Transform (Batch Processing)

Parameter	method	Default Value	Description
`algorithmFPType`	`defaultDense` or `svdDense`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`nComponents`	`defaultDense`	**0**	The number of principal components $(p_r \leq p)$. If zero, the algorithm will compute the result for $\text{nComponents} = p_r$.

Algorithm Output

The PCA Transform algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Principal Components Analysis Transform (Batch Processing)

Result ID	Result
`transformedData`	Pointer to the $n \times p_r$ numeric table that contains data projected to the principal components basis. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- pca_transform_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- PCATransformDenseBatch.java

Python* with DPC++ support

Batch Processing:

- pca_transform_batch.py

Python*

Batch Processing:

- pca_transform_batch.py

Singular Value Decomposition

Singular Value Decomposition (SVD) is one of matrix factorization techniques. It has a broad range of applications including dimensionality reduction, solving linear inverse problems, and data fitting.

Details

Given the matrix *X* of size $nimesp$, the problem is to compute the Singular Value Decomposition (SVD) $X = U\Sigma V^t$, where:

- *U* is an orthogonal matrix of size $nimesn$
- Σ is a rectangular diagonal matrix of size $nimesp$ with non-negative values on the diagonal, called singular values
- V_t is an orthogonal matrix of size $pimesp$

Columns of the matrices *U* and *V* are called left and right singular vectors, respectively.

Computation

The following computation modes are available:

- Batch and Online Processing
- Distributed Processing

Examples

C++ (CPU)

Batch Processing:

- svd_dense_batch.cpp

Online Processing:

- svd_dense_online.cpp

Distributed Processing:

- svd_dense_distr.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- SVDDenseBatch.java

Online Processing:

- SVDDenseOnline.java

Distributed Processing:

- SVDDenseDistr.java

Python*

Batch Processing:

- svd_batch.py

Online Processing:

- svd_streaming.py

Distributed Processing:

- svd_spmd.py

Performance Considerations

To get the best overall performance of singular value decomposition (SVD), for input, output, and auxiliary data, use homogeneous numeric tables of the same type as specified in the algorithmFPType class template parameter.

Online Processing

SVD in the online processing mode is at least as computationally complex as in the batch processing mode and has high memory requirements for storing auxiliary data between calls to the compute() method. On the other hand, the online version of SVD may enable you to hide the latency of reading data from a slow data source. To do this, implement load prefetching of the next data block in parallel with the compute() method for the current block.

Online processing mostly benefits SVD when the matrix of left singular vectors is not required. In this case, memory requirements for storing auxiliary data goes down from $O(p \cdot n)$ to $O(p \cdot p \cdot \mathrm{nblocks})$.

Distributed Processing

Using SVD in the distributed processing mode requires gathering local-node $pimesp$ numeric tables on the master node. When the amount of local-node work is small, that is, when the local-node data set is small, the network data transfer may become a bottleneck. To avoid this situation, ensure that local nodes have a sufficient amount of work. For example, distribute input data set across a smaller number of nodes.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Batch and Online Processing

Online processing computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \ldots \mathrm{nblocks}$.

Algorithm Input

The SVD algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Singular Value Decomposition (Batch and Online Processing)

Input ID	Input
`data`	Pointer to the numeric table that represents: • For batch processing, the entire $nimesp$ matrix *X* to be factorized. • For online processing, the $n_i \times p$ submatrix of *X* that represents the current data block in the online processing mode. The input can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The SVD algorithm has the following parameters:

Algorithm Parameters for Singular Value Decomposition (Batch and Online Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`leftSingularMatrix`	`requiredInPackedForm`	Specifies whether the matrix of left singular vectors is required. Can be: • `notRequired` - the matrix is not required • `requiredInPackedForm` - the matrix in the packed format is required
`rightSingularMatrix`	`requiredInPackedForm`	Specifies whether the matrix of left singular vectors is required. Can be: • `notRequired` - the matrix is not required • `requiredInPackedForm` - the matrix in the packed format is required

Algorithm Output

The SVD algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Singular Value Decomposition (Batch and Online Processing)

Result ID	Result
`singularValues`	Pointer to the $1imesp$ numeric table with singular values (the diagonal of the matrix Σ).
`leftSingularMatrix`	Pointer to the $nimesp$ numeric table with left singular vectors (matrix *U*). Pass `NULL` if left singular vectors are not required.

Result ID	Result
`rightSingularMatrix`	Pointer to the *pimesp* numeric table with right singular vectors (matrix *V*). Pass `NULL` if right singular vectors are not required.

NOTE By default, these results are objects of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Distributed Processing

This mode assumes that data set is split in `nblocks` blocks across computation nodes.

Algorithm Parameters

The SVD algorithm in the distributed processing mode has the following parameters:

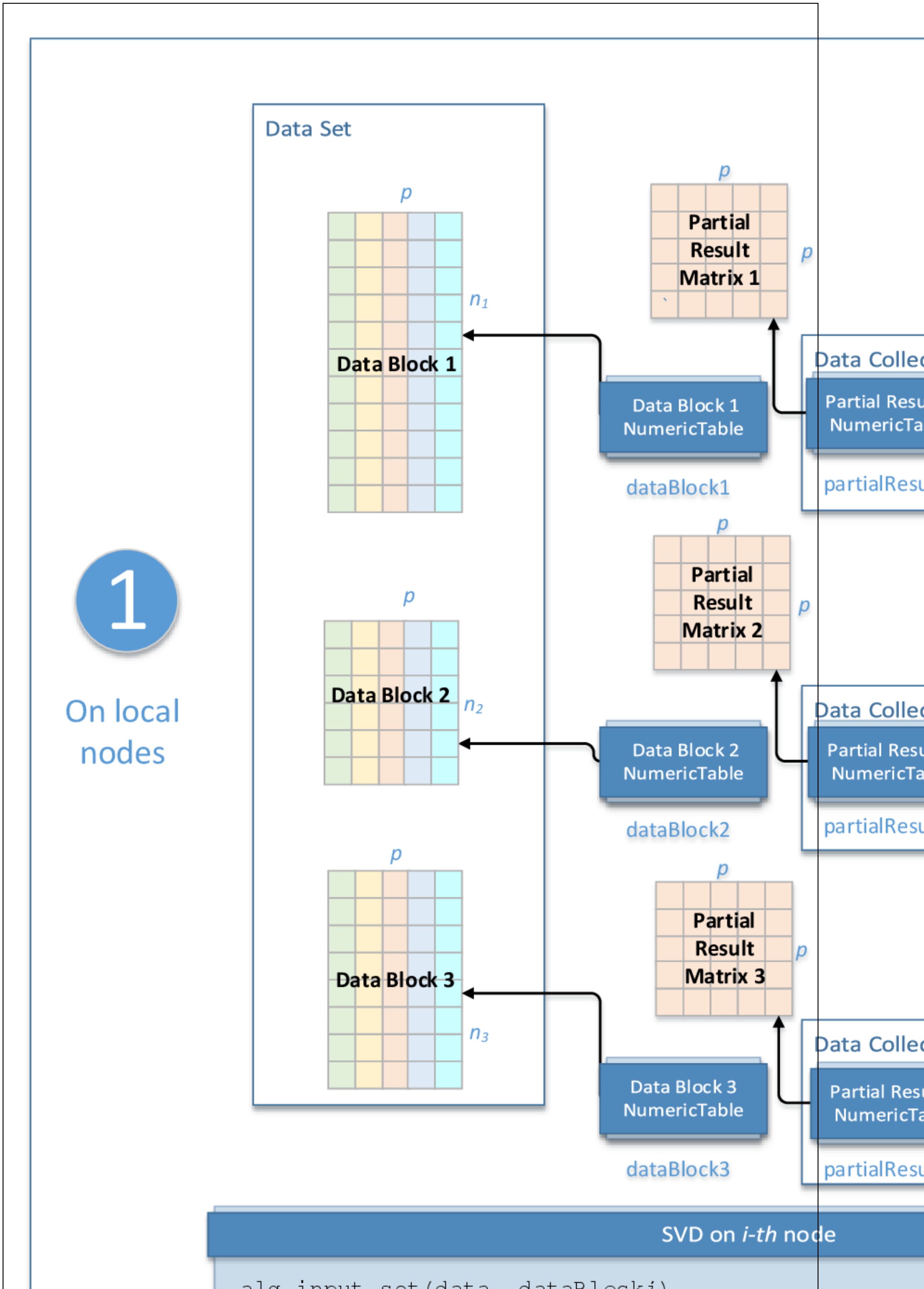
Algorithm Parameters for Singular Value Decomposition (Distributed Processing)

Parameter	Default Valude	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node • `step3Local` - the final step, performed on local nodes
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`leftSingularMatrix`	`requiredInPackedForm`	Specifies whether the matrix of left singular vectors is required. Can be: • `notRequired` - the matrix is not required • `requiredInPackedForm` - the matrix in the packed format is required
`rightSingularMatrix`	`requiredInPackedForm`	Specifies whether the matrix of right singular vectors is required. Can be: • `notRequired` - the matrix is not required • `requiredInPackedForm` - the matrix in the packed format is required

Use the three-step computation schema to compute SVD:

Step 1 - on Local Nodes

Singular Value Decomposition: Distributed Processing, Step 1 - on Local Nodes



```
alg.input.set(data, dataBlocki)
```

In this step, SVD accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Singular Value Decomposition (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node. **NOTE** The input can be an object of any class derived from `NumericTable`.

In this step, SVD calculates the results described below. Pass the `Partial Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

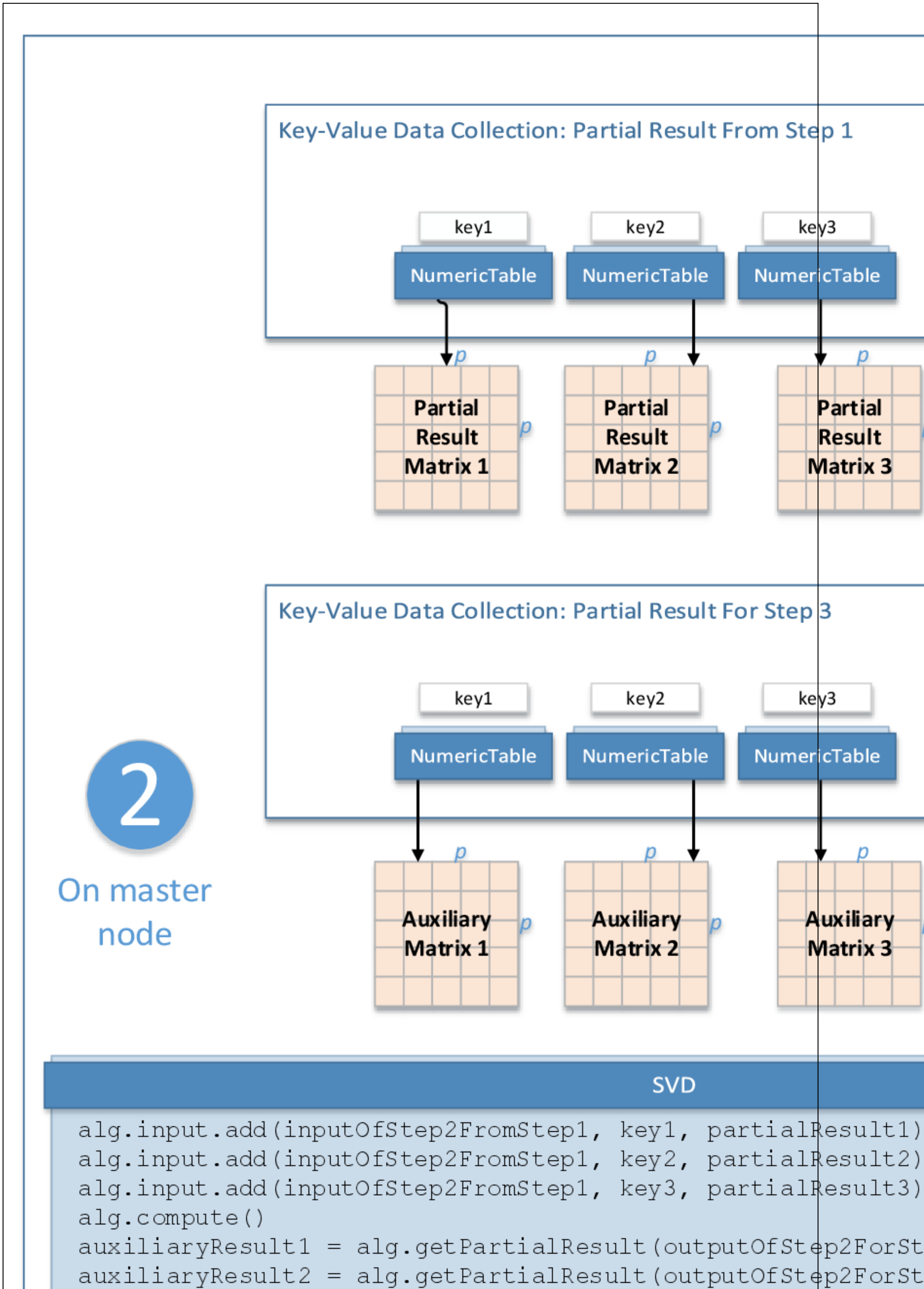
Partial Results for Singular Value Decomposition (Distributed Processing, Step 1)

Partial Result ID	Result
`outputOfStep1ForStep2`	A collection that contains numeric tables each with the partial result to transmit to the master node for Step 2.
`outputOfStep1ForStep3`	A collection that contains numeric tables each with the partial result to keep on the local node for Step 3.

NOTE By default, the tables in these collections are objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Step 2 - on Master Node

Singular Value Decomposition: Distributed Processing, Step 2 - on Master Node



```
alg.input.add(inputOfStep2FromStep1, key1, partialResult1)
alg.input.add(inputOfStep2FromStep1, key2, partialResult2)
alg.input.add(inputOfStep2FromStep1, key3, partialResult3)
alg.compute()
auxiliaryResult1 = alg.getPartialResult(outputOfStep2ForSt
auxiliaryResult2 = alg.getPartialResult(outputOfStep2ForSt
```

In this step, SVD accepts the input from each local node described below. Pass the `` `Input ID` `` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Singular Value Decomposition (Distributed Processing, Step 2)

Input ID	Input
inputOfStep2FromStep1	A collection that contains results computed in Step 1 on local nodes (`outputOfStep1ForStep2`). **NOTE** The collection can contain objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.
key	A key, a number of type `int`. Keys enable tracking the order in which partial results from Step 1 (`inputOfStep2FromStep1`) come to the master node, so that the partial results computed in Step 2 (`outputOfStep2ForStep3`) can be delivered back to local nodes in exactly the same order.

In this step, SVD calculates the results described below. Pass the `Partial Result ID` or `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Partial Results for Singular Value Decomposition (Distributed Processing, Step 2)

Partial Result ID	Result
outputOfStep2ForStep3	A collection that contains numeric tables to be split across local nodes to compute left singular vectors. Set to `NULL` if you do not need left singular vectors. **NOTE** By default, these tables are objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

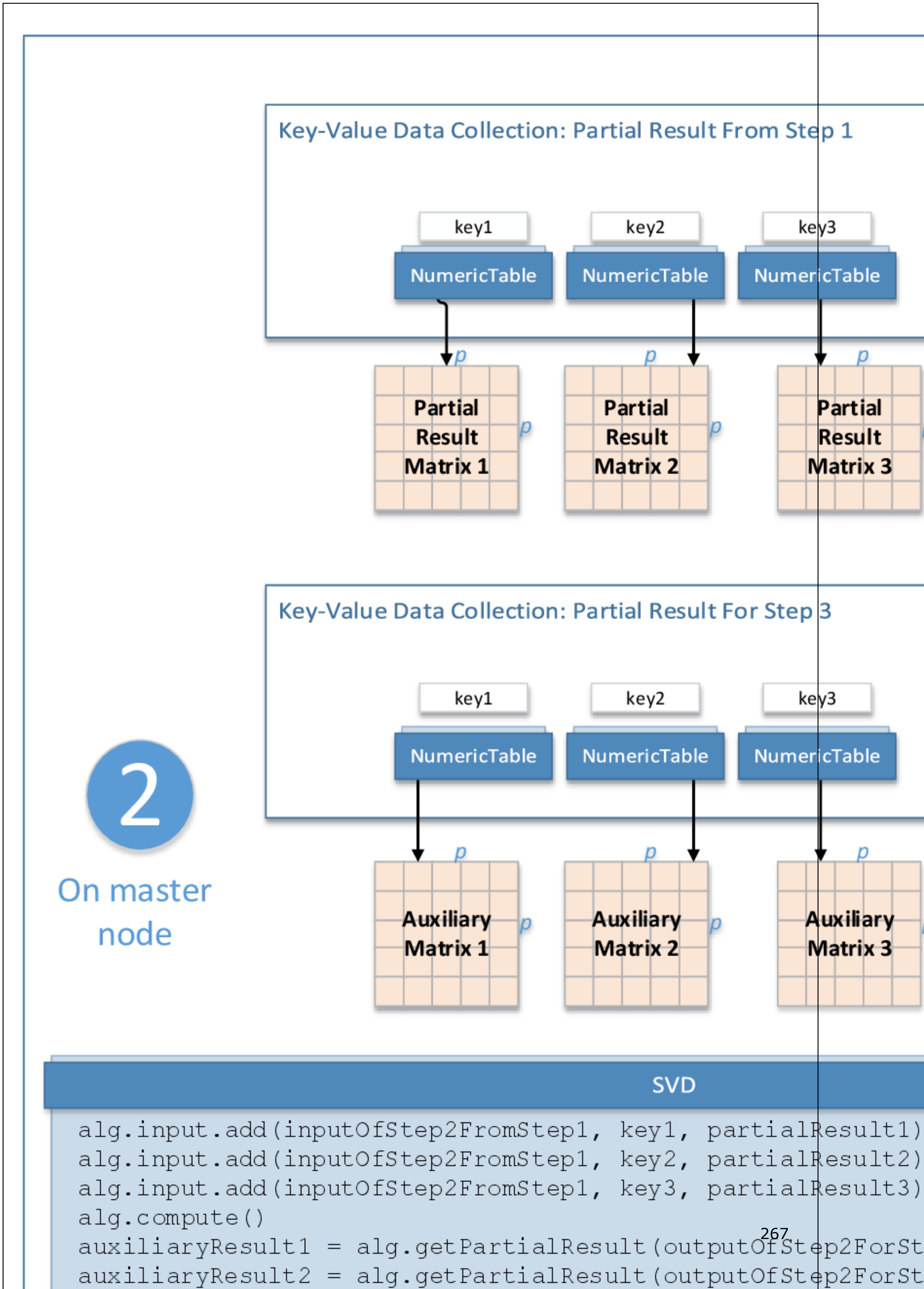
Output for Singular Value Decomposition (Distributed Processing, Step 2)

Result ID	Result
singularValues	Pointer to the $1imesp$ numeric table with singular values (the diagonal of the matrix Σ). **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
rightSingularMatrix	Pointer to the $pimesp$ numeric table with right singular vectors (matrix *V*). Pass `NULL` if right singular vectors are not required.

Result ID	Result
	NOTE By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Step 3 - on Local Nodes

Singular Value Decomposition: Distributed Processing, Step 3 - on Local Nodes



```
alg.input.add(inputOfStep2FromStep1, key1, partialResult1)
alg.input.add(inputOfStep2FromStep1, key2, partialResult2)
alg.input.add(inputOfStep2FromStep1, key3, partialResult3)
alg.compute()
auxiliaryResult1 = alg.getPartialResult(outputOfStep2ForSt
auxiliaryResult2 = alg.getPartialResult(outputOfStep2ForSt
```

In this step, SVD accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Singular Value Decomposition (Distributed Processing, Step 3)

Input ID	Input
`inputOfStep3FromStep1`	A collection that contains results computed in Step 1 on local nodes (`outputOfStep1ForStep3`). **NOTE** The collection can contain objects of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.
`inputOfStep3FromStep2`	A collection that contains results computed in Step 2 on local nodes (`outputOfStep2ForStep3`). **NOTE** The collection can contain objects of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

In this step, SVD calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Singular Value Decomposition (Distributed Processing, Step 3)

Result ID	Result
`leftSingularMatrix`	Pointer to the $nimesp$ numeric table with left singular vectors (matrix *U*). Pass `NULL` if left singular vectors are not required. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Association Rules

Association rules mining is the method for uncovering the most important relationships between variables. Its main application is a store basket analysis, which aims at discovery of a relationship between groups of products with some level of confidence.

Details

The library provides Apriori algorithm for association rule mining [Agrawal94].

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items (products) and subset $T \subset I$ is a transaction associated with item set I. The association rule has the form: $X \Rightarrow Y$, where $X \subset I, Y \subset I$, and intersection of *X* and *Y* is empty: $X \cap Y = \emptyset$. The left-hand-side set of items (*itemset*) *X* is called antecedent, while the right-hand-side itemset Y is called consequent of the rule.

Let $D = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions, each associated with item set I. Item subset $X \subset I$ has support *s* in the transaction set *D* if *s* percent of transactions in *D* contains *X*.

The association rule $X \Rightarrow Y$ in the transaction set *D* holds with confidence *c* if *c* percent of transactions in *D* that contain *X* also contains *Y*. Confidence of the rule can be represented as conditional probability:

$$confidence(X \Rightarrow Y) = support(X \cup Y)/support(X)$$

For a given set of transactions $D = \{T_1, T_2, \ldots, T_n\}$, the minimum support s and minimum confidence c discover all item sets *X* with support greater than *s* and generate all association rules $X \Rightarrow Y$ with confidence greater than *c*.

Therefore, the association rule discovery is decomposed into two stages: mining (training) and discovery (prediction). The mining stage involves generation of large item sets, that is, the sets that have support greater than the given parameters. At the discovery stage, the algorithm generates association rules using the large item sets identified at the mining stage.

Batch Processing

Algorithm Input

The association rules algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Association Rules (Batch Processing)

Input ID	Input
`data`	Pointer to the $n \times 2$ numeric table t with the mining data. Each row consists of two integers: • Transaction ID, the number between 0 and *nTransactions - 1*. • Item ID, the number between 0 and *nUniqueItems - 1*. The input can be an object of any class derived from NumericTable except PackedTriangularMatrix and PackedSymmetricMatrix.

Algorithm Parameters

The association rules algorithm has the following parameters:

Algorithm Parameters for Association Rules (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the algorithm. The only method supported so far is Apriori.
`minSupport`	**0.01**	Minimal support, a number in the [0,1) interval.
`minConfidence`	**0.6**	Minimal confidence, a number in the [0,1) interval.
`nUniqueItems`	**0**	The total number of unique items. If set to zero, the library automatically determines the number of unique items from the input data.

Parameter	Default Value	Description
`nTransactions`	**0**	The total number of transactions. If set to zero, the library automatically determines the number transactions from the input data.
`discoverRules`	`true`	A flag that enables generation of the rules from large item sets.
`itemsetsOrder`	`itemsetsUnsorted`	The sort order of returned item sets: • itemsetsUnsorted - not sorted • itemsetsSortedBySupport - sorted by support in a descending order
`rulesOrder`	`rulesUnsorted`	The sort order of returned rules: • rulesUnsorted - not sorted • rulesSortedByConfidence - sorted by support in a descending order
`minItemsetSize`	**0**	A parameter that defines the minimal size of item sets to be included into the array of results. The value of zero imposes no limitations on the minimal size of item sets.
`maxItemsetSize`	**0**	A parameter that defines the maximal size of item sets to be included into the array of results. The value of zero imposes no limitations on the maximal size of item sets.

Algorithm Output

The association rules algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Association Rules (Batch Processing)

Result ID	Result
`largeItemsets`	Pointer to the numeric table with large item sets. The number of rows in the table equals the number of items in the large item sets. Each row contains two integers: • ID of the large item set, the number between 0 and nLargeItemsets -1. • ID of the item, the number between 0 and *nUniqueItems-1*.
`largeItemsetsSupport`	Pointer to the $nLargeItemsets \times 2$ numeric table of support values. Each row contains two integers: • ID of the large item set, the number between 0 and nLargeItemsets-1. • The support value, the number of times the item set is met in the array of transactions.
`antecedentItemsets`	Pointer to the $nAntecedentItems \times 2$ numeric table that contains the left-hand-side (X) part of the association rules. Each row contains two integers: • Rule ID, the number between 0 and *nAntecedentItems-1*. • Item ID, the number between 0 and *nUniqueItems-1*.
`conseqentItemsets`	Pointer to the $nConsequentItems \times 2$ numeric table that contains the right-hand-side (Y) part of the association rules. Each row contains two integers: • Rule ID, the number between 0 and *nConsequentItems-1*. • Item ID, the number between 0 and *nUniqueItems-1*.

Result ID	Result
confidence	Pointer to the $nRules \times 1$ numeric table that contains confidence values of rules, floating-point numbers between 0 and 1. Confidence value in the i-th position corresponds to the rule with the index i.

By default, the result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

NOTE

- The library requires transactions and items for each transaction to be passed in the ascending order.
- Numbering of rules starts at 0.
- The library calculates the sizes of numeric tables intended for results in a call to the algorithm. Avoid allocating the memory in numeric tables intended for results because, in general, it is impossible to accurately estimate the required memory size. If the memory interfaced by the numeric tables is allocated and its amount is insufficient to store the results, the algorithm returns an error.

Examples

C++ (CPU)

Batch Processing:

- assoc_rules_apriori_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- AssocRulesAprioriBatch.java

Python*

Batch Processing:

- association_rules_batch.py

Performance Considerations

To get the best overall performance of the association rules algorithm, whenever possible use the following numeric tables and data types:

- A SOA numeric table of type int to store features.
- A homogenous numeric table of type int to store large item sets, support values, and left-hand-side and right-hand-side parts of association rules.
- A numeric table with the confidence values of the same data type as specified in the algorithmFPType template parameter of the class.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Kernel Functions

NOTE Kernel functions are also available with oneAPI interfaces:

- Linear kernel
- Radial Basis Function (RBF) kernel

Kernel functions form a class of algorithms for pattern analysis. The main characteristic of kernel functions is a distinct approach to this problem. Instead of reducing the dimension of the original data, kernel functions map the data into higher-dimensional spaces in order to make the data more easily separable there.

Linear Kernel

A linear kernel is the simplest kernel function.

Problem Statement

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p* and a set *Y* of *m* feature vectors $y_1 = (y_{11}, \ldots, y_{1p}), \ldots, y_m = (y_{m1}, \ldots, x_{mp})$, the problem is to compute the linear kernel function $K(x_i, , y_i)$ for any pair of input vectors: $K(x_i, y_i) = kX_i{}^T y_i + b$.

Batch Processing

Algorithm Input

The linear kernel function accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Linear Kernel (Batch Processing)

Input ID	Input
X	Pointer to the $nimesp$ numeric table that represents the matrix X. This table can be an object of any class derived from NumericTable.
Y	Pointer to the $m \times p$ numeric table that represents the matrix Y. This table can be an object of any class derived from NumericTable.

Algorithm Parameters

The linear kernel function has the following parameters:

Algorithm Parameters for Linear Kernel (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available computation methods: • defaultDense - default performance-oriented method • fastCSR - performance-oriented method for CSR numeric tables
computationMode	matrixMatrix	Computation mode for the kernel function. Can be: For CPU: • vectorVector - compute the kernel function for given feature vectors x_i and y_j • matrixVector - compute the kernel function for all vectors in the set *X* and a given feature vector y_j • matrixMatrix - compute the kernel function for all vectors in the sets *X* and *Y*. In oneDAL, this mode requires equal numbers of observations in both input tables: $n = m$. For GPU: • matrixMatrix - compute the kernel function for all vectors in the sets *X* and *Y*. In oneDAL, this mode requires equal numbers of observations in both input tables: $n = m$.
rowIndexX	**0**	Index i of the vector in the set *X* for the vectorVector computation mode.
rowIndexY	**0**	Index *j* of the vector in the set *Y* for the vectorVector or matrixVector computation mode.
rowIndexResult	**0**	Row index in the values numeric table to locate the result of the computation for the vectorVector computation mode.
k	**1.0**	The coefficient *k* of the linear kernel.
b	**0.0**	The coefficient *b* of the linear kernel.

Algorithm Output

The linear kernel function calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Linear Kernel (Batch Processing)

Result ID	Result
values	Pointer to the $n \times m$ numeric table with the values of the kernel function. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Examples

oneAPI DPC++

Batch Processing:

- dpc_linear_kernel_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_linear_kernel_dense_batch.cpp

C++ (CPU)

Batch Processing:

- kernel_func_lin_dense_batch.cpp
- kernel_func_lin_csr_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- KernelFuncLinDenseBatch.java
- KernelFuncLinCSRBatch.java

Radial Basis Function Kernel

The Radial Basis Function (RBF) kernel is a popular kernel function used in kernelized learning algorithms.

Problem Statement

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \dots, x_{1p}), \dots, x_n = (x_{n1}, \dots, x_{np})$ of dimension *p* and a set *Y* of *m* feature vectors $y_1 = (y_{11}, \dots, y_{1p}), \dots, y_m = (y_{m1}, \dots, x_{mp})$, the problem is to compute the RBF kernel function $K(x_i, , y_i)$ for any pair of input vectors:

$$K\left(x_i, y_j\right) = exp\left(-\frac{\left(\|x_i - y_j\|\right)^2}{2\sigma^2}\right)$$

Batch Processing

Algorithm Input

The RBF kernel accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Radial Basis Function Kernel (Batch Processing)

Input ID	Input
X	Pointer to the $nimesp$ numeric table that represents the matrix *X*. This table can be an object of any class derived from `NumericTable`.

Input ID	Input
Y	Pointer to the $m \times p$ numeric table that represents the matrix *Y*. This table can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The RBF kernel has the following parameters:

Algorithm Parameters for Radial Basis Function Kernel (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available computation methods: • `defaultDense` - default performance-oriented method • `fastCSR` - performance-oriented method for CSR numeric tables
`computationMode`	`matrixMatrix`	Computation mode for the kernel function. Can be: For CPU: • `vectorVector` - compute the kernel function for given feature vectors x_i and y_j • `matrixVector` - compute the kernel function for all vectors in the set *X* and a given feature vector y_j • `matrixMatrix` - compute the kernel function for all vectors in the sets *X* and *Y*. In oneDAL, this mode requires equal numbers of observations in both input tables: $n = m$. For GPU: • `matrixMatrix` - compute the kernel function for all vectors in the sets *X* and *Y*. In oneDAL, this mode requires equal numbers of observations in both input tables: $n = m$.
`rowIndexX`	**0**	Index *i* of the vector in the set *X* for the `vectorVector` computation mode.
`rowIndexY`	**0**	Index *j* of the vector in the set *Y* for the `vectorVector` or `matrixVector` computation mode.
`rowIndexResult`	**0**	Row index in the values numeric table to locate the result of the computation for the `vectorVector` computation mode.
`sigma`	**1.0**	The coefficient σ of the RBF kernel.

Algorithm Output

The RBF kernel calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Radial Basis Function Kernel (Batch Processing)

Result ID	Result
values	Pointer to the $n \times m$ numeric table with the values of the kernel function. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Examples

oneAPI DPC++

Batch Processing:

- dpc_rbf_kernel_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_rbf_kernel_dense_batch.cpp

C++ (CPU)

Batch Processing:

- kernel_func_rbf_dense_batch.cpp
- kernel_func_rbf_csr_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- KernelFuncRbfDenseBatch.java
- KernelFuncRbfCSRBatch.java

Expectation-Maximization

Expectation-Maximization (EM) algorithm is an iterative method for finding the maximum likelihood and maximum a posteriori estimates of parameters in models that typically depend on hidden variables.

While serving as a clustering technique, EM is also used in non-linear dimensionality reduction, missing value problems, and other areas.

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to find a maximum-likelihood estimate of the parameters of the underlying distribution when the data is incomplete or has missing values.

Expectation-Maximization (EM) Algorithm in the General Form

Let *X* be the observed data which has log-likelihood $l(\theta; X)$ depending on the parameters θ. Let X^m be the latent or missing data, so that $T = (X, X^m)$ is the complete data with log-likelihood $l_0(\theta; X)$. The algorithm for solving the problem in its general form is the following EM algorithm ([Dempster77], [Hastie2009]):

1. Choose initial values of the parameters $\theta^{(0)}$.
2. *Expectation step*: in the *j*-th step, compute $Q(\theta', \theta^{(j)}) = E(l_0(\theta'; T)|X, \theta^{(j)})$ as a function of the dummy argument θ'.
3. *Maximization step*: in the *j*-th step, calculate the new estimate $\theta^{(j+1)}$ by maximizing $Q(\theta', \theta^{(j)})$ over θ'.
4. Repeat steps 2 and 3 until convergence.

EM algorithm for the Gaussian Mixture Model

Gaussian Mixture Model (GMM) is a mixture of k p-dimensional multivariate Gaussian distributions represented as

$$F\left(x|\alpha_1, \ldots, \alpha_k; \theta_1, \ldots, \theta_k\right) = \sum_{i=1}^{k} \alpha_i \int_{-\infty}^{x} pd\left(y|\theta_i\right),$$

where $\sum_{i=1}^{k} \alpha_i = 1$ and $\alpha_i \geq 0$.

The $pd(x|\theta_i)$ is the probability density function with parameters $\theta_i = (m_i, \Sigma_i)$, where m_i the vector of means, and Σ_i is the variance-covariance matrix. The probability density function for a *p*-dimensional multivariate Gaussian distribution is defined as follows:

$$pd\left(x|\theta_i\right) = \frac{\exp\left(-\frac{1}{2}(x - m_i)^T \sum_i^{-1} (x - m_i)\right)}{\sqrt{(2\pi)^p |\sum_i|}}.$$

Let $x_{ij} = I\{x_i \text{belongs to j mixture component}\}$ be the indicator function and $\theta = (\alpha_1, \ldots, \alpha_k; \theta_1, \ldots, \theta_k)$.

Computation

The EM algorithm for GMM includes the following steps:

Define the weights as follows:

$$w_{ij} = \frac{pd\left(x_i|z_{ij}, \theta_j\right) \alpha_j}{\sum_{r=1}^{k} pd\left(x_i|z_{ir}, \theta_r\right) \alpha_r}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, k$.

1. Choose initial values of the parameters: $\theta^{(0)} = \left(\alpha_1^{(0)}, \ldots, \alpha_k^{(0)}; \theta_1^{(0)}, \ldots, \theta_k^{(0)}\right)$
2. *Expectation step*: in the *j*-th step, compute the matrix $W = (w_{ij})_{nxk}$ with the weights w_{ij}
3. Maximization step: in the *j*-th step, for all $r = 1, \ldots, k$ compute:

a. The mixture weights $\alpha_r^{(j+1)} = \frac{n_r}{n}$, where $n_r = \sum_{i=1}^{n} w_{ir}$ is the “amount” of the feature vectors that are assigned to the *r*-th mixture component

b. Mean estimates $m_r^{(j+1)} = \frac{1}{n_r} \sum_{i=1}^{n} w_{ir} x_i$

c. Covariance estimate $\sum_r^{(j+1)} = (\sigma_{r,hg}^{(j+1)})$ of size $pimesp$ with $\sigma_{r,hg}^{(j+1)} = \frac{1}{n_r} \sum_{l=1}^{n} w_{lr}(x_{lh} - m_{r,h}^{(j+1)})(x_{lg} - m_{r,g}^{(j+1)})$

4. Repeat steps 2 and 3 until any of these conditions is met:

   - $|\log(\theta^{(j+1)} - \theta^{(j)})| < \epsilon$, where the likelihood function is:

     $\log(\theta) = \sum_{i=1}^{n} \log(\sum_{j=1}^{k} pd(x_i|z_j, \theta_j)\alpha_j)$
   - The number of iterations exceeds the predefined level.

Initialization

The EM algorithm for GMM requires initialized vector of weights, vectors of means, and variance-covariance [Biernacki2003, Maitra2009].

The EM initialization algorithm for GMM includes the following steps:

1. Perform nTrials starts of the EM algorithm with nIterations iterations and start values:
   - Initial means - *k* different random observations from the input data set
   - Initial weights - the values of $1/k$
   - Initial covariance matrices - the covariance of the input data
2. Regard the result of the best EM algorithm in terms of the likelihood function values as the result of initialization

Initialization

The EM algorithm for GMM requires initialized vector of weights, vectors of means, and variance-covariance. Skip the initialization step if you already calculated initial weights, means, and covariance matrices.

Batch Processing

Algorithm Input

The EM for GMM initialization algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Expectation-Maximization Initialization (Batch Processing)

Input ID	Input
data	Pointer to the $nimesp$ numeric table with the data to which the EM initialization algorithm is applied. The input can be an object of any class derived from NumericTable.

Algorithm Parameters

The EM for GMM initialization algorithm has the following parameters:

Algorithm Parameters for Expectation-Maximization Initialization (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`nComponents`	Not applicable	The number of components in the Gaussian Mixture Model, a required parameter.
`nTrials`	**20**	The number of starts of the EM algorithm.
`nIterations`	**10**	The maximal number of iterations in each start of the EM algorithm.
`accuracyThreshold`	1.0e-04	The threshold for termination of the algorithm.
`covarianceStorage`	`full`	Covariance matrix storage scheme in the Gaussian Mixture Model: • `full` - covariance matrices are stored as numeric tables of size $pimesp$. All elements of the matrix are updated during the processing. • `diagonal` - covariance matrices are stored as numeric tables of size $1imesp$. Only diagonal elements of the matrix are updated during the processing, and the rest are assumed to be zero.
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally to get the initial means in each EM start.

Algorithm Output

The EM for GMM initialization algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Expectation-Maximization Initialization (Batch Processing)

Result ID	Result
`weights`	Pointer to the $1imesk$ numeric table with mixture weights. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`means`	Pointer to the $k \times p$ numeric table with each row containing the estimate of the means for the *i*-th mixture component, where $i = 0, 1, \ldots, k-1$.

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`covariances`	Pointer to the `DataCollection` object that contains *k* numeric tables, each with the $pimesp$ variance-covariance matrix for the *i*-th mixture component of size: • $pimesp$ - for the full covariance matrix storage scheme • $1imesp$ - for the diagonal covariance matrix storage scheme **NOTE** By default, the collection contains objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.

Computation

Batch Processing

Algorithm Input

The EM for GMM algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm.

Algorithm Input for Expectation-Maximization Computaion (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the data to which the EM algorithm is applied. The input can be an object of any class derived from `NumericTable`.
`inputWeights`	Pointer to the $1imesk$ numeric table with initial mixture weights. This input can be an object of any class derived from NumericTable.
`inputMeans`	Pointer to a $k \times p$ numeric table. Each row in this table contains the initial value of the means for the *i*-th mixture component, where $i = 0, 1, \ldots, k - 1$. This input can be an object of any class derived from `NumericTable`.
`inputCovariances`	Pointer to the `DataCollection` object that contains *k* numeric tables, each with the $pimesp$ variance-covariance matrix for the *i*-th mixture component of size: • $pimesp$ - for the full covariance matrix storage scheme • $1imesp$ - for the diagonal covariance matrix storage scheme The collection can contain objects of any class derived from NumericTable.

Input ID	Input
`inputValues`	Pointer to the result of the EM for GMM initialization algorithm. The result of initialization contains weights, means, and a collection of covariances. You can use this input to set the initial values for the EM for GMM algorithm instead of explicitly specifying the weights, means, and covariance collection.

Algorithm Parameters

The EM for GMM algorithm has the following parameters:

Algorithm Parameters for Expectation-Maximization Computaion (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`nComponents`	Not applicable	The number of components in the Gaussian Mixture Model, a required parameter.
`maxIterations`	**10**	The maximal number of iterations in the algorithm.
`accuracyThreshold`	1.0e-04	The threshold for termination of the algorithm.
`covariance`	Pointer to an object of the BatchIface class	Pointer to the algorithm that computes the covariance matrix. **NOTE** By default, the respective oneDAL algorithm is used, implemented in the class derived from `BatchIface`.
`regularizationFactor`	**0.01**	Factor for covariance regularization in the case of ill-conditional data.
`covarianceStorage`	`full`	Covariance matrix storage scheme in the Gaussian Mixture Model: • `full` - covariance matrices are stored as numeric tables of size $pimesp$. All elements of the matrix are updated during the processing. • `diagonal` - covariance matrices are stored as numeric tables of size $1imesp$. Only diagonal elements of the matrix are updated during the processing, and the rest are assumed to be zero.

Algorithm Output

The EM for GMM algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Expectation-Maximization Computaion (Batch Processing)

Result ID	Result
`weights`	Pointer to the $1imesk$ numeric table with mixture weights. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`means`	Pointer to the $k \times p$ numeric table with each row containing the estimate of the means for the *i*-th mixture component, where $i = 0, 1, \ldots, k - 1$. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`covariances`	Pointer to the DataCollection object that contains *k* numeric tables, each with the $pimesp$ variance-covariance matrix for the *i*-th mixture component of size: • $pimesp$ - for the full covariance matrix storage scheme • $1imesp$ - for the diagonal covariance matrix storage scheme **NOTE** By default, the collection contains objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.
`goalFunction`	Pointer to the $1imes1$ numeric table with the value of the logarithm of the likelihood function after the last iteration. **NOTE** By default, this result is an object of the `HomogenNumericTable` class.
`nIterations`	Pointer to the $1imes1$ numeric table with the number of iterations computed after completion of the algorithm. **NOTE** By default, this result is an object of the `HomogenNumericTable` class.

Examples

C++ (CPU)

Batch Processing:

- em_gmm_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- EmGmmDenseBatch.java

Python*

Batch Processing:

- em_gmm_batch.py

Performance Considerations

To get the best overall performance of the expectation-maximization algorithm at the initialization and computation stages:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the algorithmFPType class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Cholesky Decomposition

Cholesky decomposition is a matrix factorization technique that decomposes a symmetric positive-definite matrix into a product of a lower triangular matrix and its conjugate transpose.

Because of numerical stability and superior efficiency in comparison with other methods, Cholesky decomposition is widely used in numerical methods for solving symmetric linear systems. It is also used in non-linear optimization problems, Monte Carlo simulation, and Kalman filtration.

Details

Given a symmetric positive-definite matrix *X* of size $pimesp$, the problem is to compute the Cholesky decomposition $X = LL^T$, where *L* is a lower triangular matrix.

Batch Processing

Algorithm Input

Cholesky decomposition accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Cholesky Decomposition (Batch Processing)

Input ID	Input
`data`	Pointer to the $pimesp$ numeric table that represents the symmetric positive-definite matrix *X* for which the Cholesky decomposition is computed.

Input ID	Input
	The input can be an object of any class derived from `NumericTable` that can represent symmetric matrices. For example, the `PackedTriangularMatrix` class cannot represent a symmetric matrix.

Algorithm Parameters

Cholesky decomposition has the following parameters:

Algorithm Parameters for Cholesky Decomposition (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

Cholesky decomposition calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Cholesky Decomposition (Batch Processing)

Result ID	Result
`choleskyFactor`	Pointer to the $pimesp$ numeric table that represents the lower triangular matrix *L* (Cholesky factor). By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `upperPackedTriangularMatrix` layout.

Examples

C++ (CPU)

Batch Processing:

- cholesky_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- CholeskyDenseBatch.java

Python*

Batch Processing:

- cholesky_batch.py

Performance Considerations

To get the best overall performance when Cholesky decomposition:

- If input data is homogeneous, for input matrix *X* and output matrix *L* use homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

QR Decomposition

QR decomposition is a matrix factorization technique that decomposes a matrix into a product of an orthogonal matrix *Q* and an upper triangular matrix *R*.

QR decomposition is used in solving linear inverse and least squares problems. It also serves as a basis for algorithms that find eigenvalues and eigenvectors.

- QR Decomposition without Pivoting
- Pivoted QR Decomposition

Performance Considerations

To get the best overall performance of the QR decomposition, for input, output, and auxiliary data, use homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.

Online Processing

QR decomposition in the online processing mode is at least as computationally complex as in the batch processing mode and has high memory requirements for storing auxiliary data between calls to the `compute()s` method. On the other hand, the online version of QR decomposition may enable you to hide the latency of reading data from a slow data source. To do this, implement load prefetching of the next data block in parallel with the `compute()` method for the current block.

Online processing mostly benefits QR decomposition when the matrix Q is not required. In this case, memory requirements for storing auxiliary data goes down from $O(p \cdot n)$ to $O(p \cdot p \cdot \mathrm{nblocks})$.

Distributed Processing

Using QR decomposition in the distributed processing mode requires gathering local-node $p imes p$ numeric tables on the master node. When the amount of local-node work is small, that is, when the local-node data set is small, the network data transfer may become a bottleneck. To avoid this situation, ensure that local nodes have a sufficient amount of work. For example, distribute the input data set across a smaller number of nodes.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

QR Decomposition without Pivoting

Given the matrix *X* of size $nimesp$, the problem is to compute the QR decomposition $X = QR$, where

- *Q* is an orthogonal matrix of size $nimesn$
- *R* is a rectangular upper triangular matrix of size $nimesp$

The library requires $n > p$. In this case:

$$X = QR = [Q_1, Q_2] \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1$$

where the matrix Q_1 has the size $nimesp$ and R_1 has the size $pimesp$.

Computation

The following computation modes are available:

- Batch and Online Processing
- Distributed Processing

Examples

C++ (CPU)

Batch Processing:

- qr_dense_batch.cpp

Online Processing:

- qr_dense_online.cpp

Distributed Processing:

- qr_dense_distr.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- QRDenseBatch.java

Online Processing:

- QRDenseOnline.java

Distributed Processing:

- QRDenseDistr.java

Python*

Batch Processing:

- qr_batch.py

Online Processing:

- qr_streaming.py

Distributed Processing:

- qr_spmd.py

Batch and Online Processing

Online processing computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \ldots \text{nblocks}$.

Algorithm Input

QR decomposition accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for QR Decomposition without Pivoting (Batch and Online Processing)

Input ID	Input
`data`	Pointer to the numeric table that represents: • For batch processing: the entire $nimesp$ matrix *X* to be factorized. • For online processing: the $n_i \times p$ submatrix of *X* that represents the current data block in the online processing mode. Note that each current data block must have sufficient size: $n_i > p$. The input can be an object of any class derived from `NumericTable`.

Algorithm Parameters

QR decomposition has the following parameters:

Algorithm Parameters for QR Decomposition without Pivoting (Batch and Online Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

QR decomposition calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for QR Decomposition without Pivoting (Batch and Online Processing)

Result ID	Result
`matrixQ`	Pointer to the numeric table with the $nimesp$ matrix Q_1. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`matrixR`	Pointer to the numeric table with the $pimesp$ upper triangular matrix R_1.

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

Distributed Processing

This mode assumes that the data set is split into `nblocks` blocks across computation nodes.

Algorithm Parameters

QR decomposition in the distributed processing mode has the following parameters:

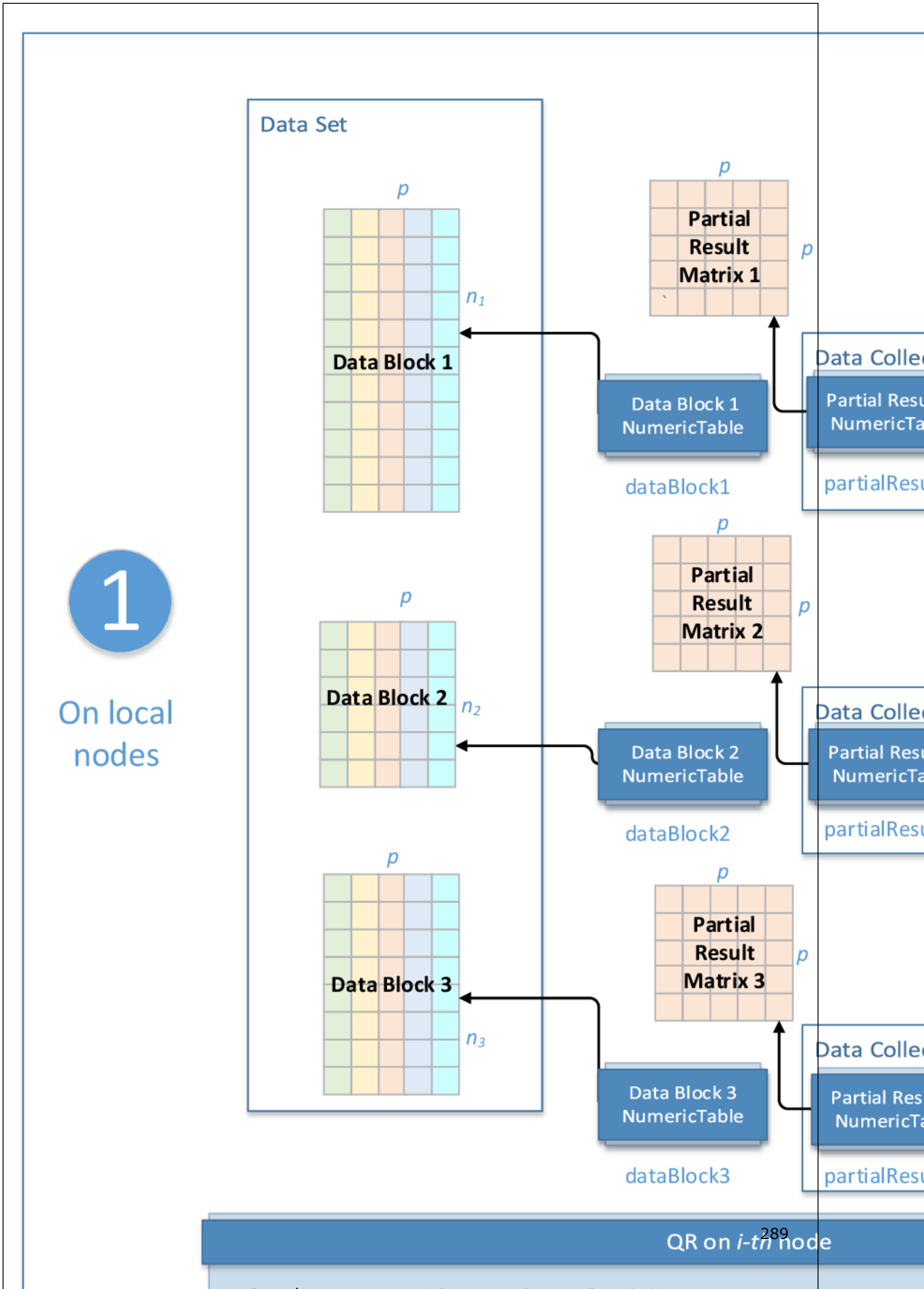
Algorithm Parameters for QR Decomposition without Pivoting (Distributed Processing)

Parameter	Default Valude	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node • `step3Local` - the final step, performed on local nodes
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Use the three-step computation schema to compute QR decomposition:

Step 1 - on Local Nodes

QR Decomposition without Pivoting: Distributed Processing, Step 1 - on Local Nodes



In this step, QR decomposition accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for QR Decomposition without Pivoting (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node. Note that each data block must have sufficient size: $n_i > p$. **NOTE** The input can be an object of any class derived from `NumericTable`.

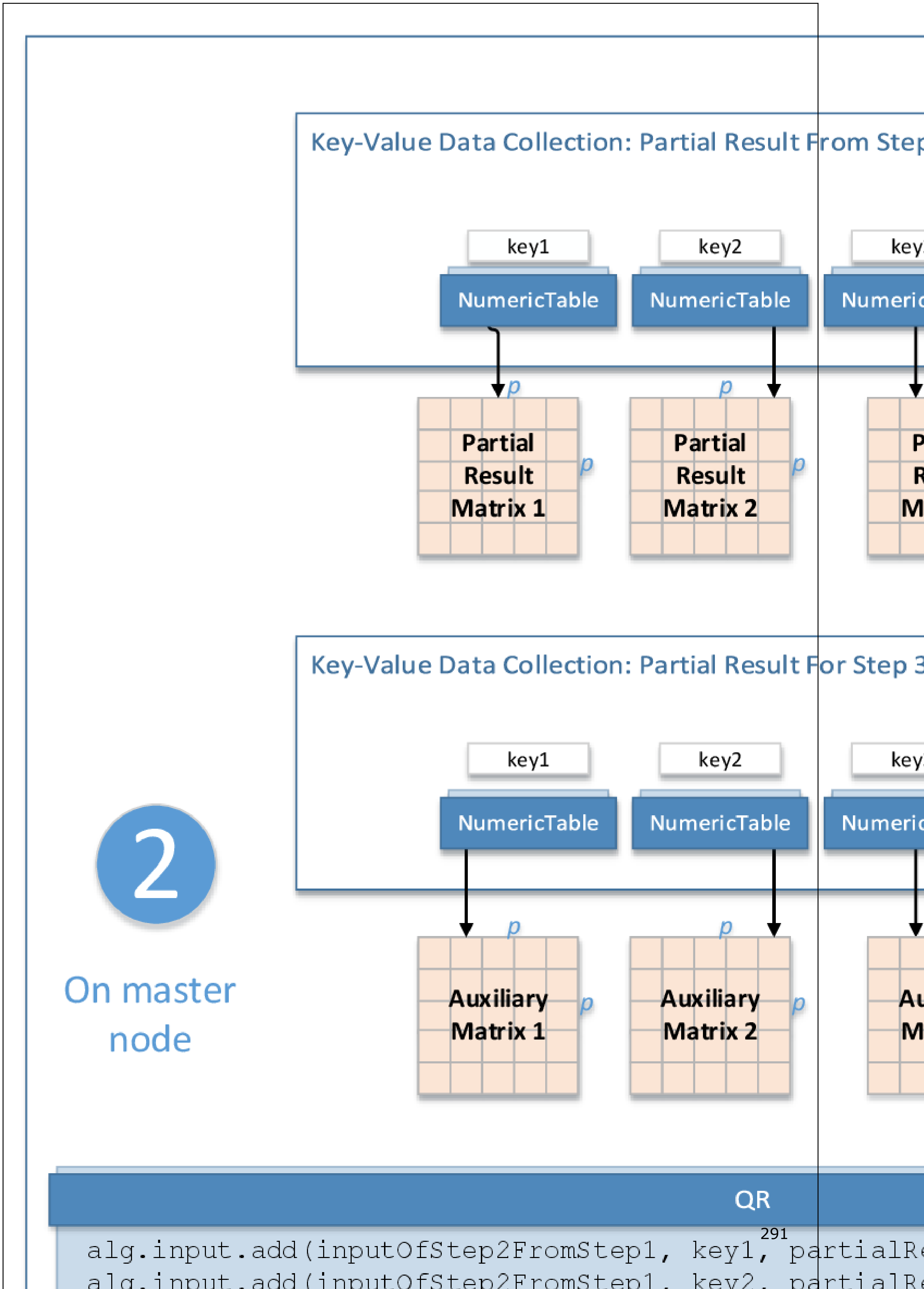
In this step, QR decomposition calculates the results described below. Pass the `Partial Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Partial Results for QR Decomposition without Pivoting (Distributed Processing, Step 1)

Partial Result ID	Result
`outputOfStep1ForStep2`	A collection that contains numeric tables each with the partial result to transmit to the master node for Step 2. **NOTE** By default, these tables are objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.
`outputOfStep1ForStep3`	A collection that contains numeric tables each with the partial result to keep on the local node for Step 3. **NOTE** By default, these tables are objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Step 2 - on Master Node

QR Decomposition without Pivoting: Distributed Processing, Step 2 - on Master Node



```
alg.input.add(inputOfStep2FromStep1, key1, partialRe
alg.input.add(inputOfStep2FromStep1, key2, partialRe
```

In this step, QR decomposition accepts the input from each local node described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for QR Decomposition without Pivoting (Distributed Processing, Step 2)

Input ID	Input
`inputOfStep2FromStep1`	A collection that contains results computed in Step 1 on local nodes (`outputOfStep1ForStep2`). **NOTE** This collection can contain objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.
`key`	A key, a number of type int. Keys enable tracking the order in which partial results from Step 1 (`inputOfStep2FromStep1`) come to the master node, so that the partial results computed in Step 2 (`outputOfStep2ForStep3`) can be delivered back to local nodes in exactly the same order.

In this step, QR decomposition calculates the results described below. Pass the `Result ID` or `Partial Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Partial Results for QR Decomposition without Pivoting (Distributed Processing, Step 2)

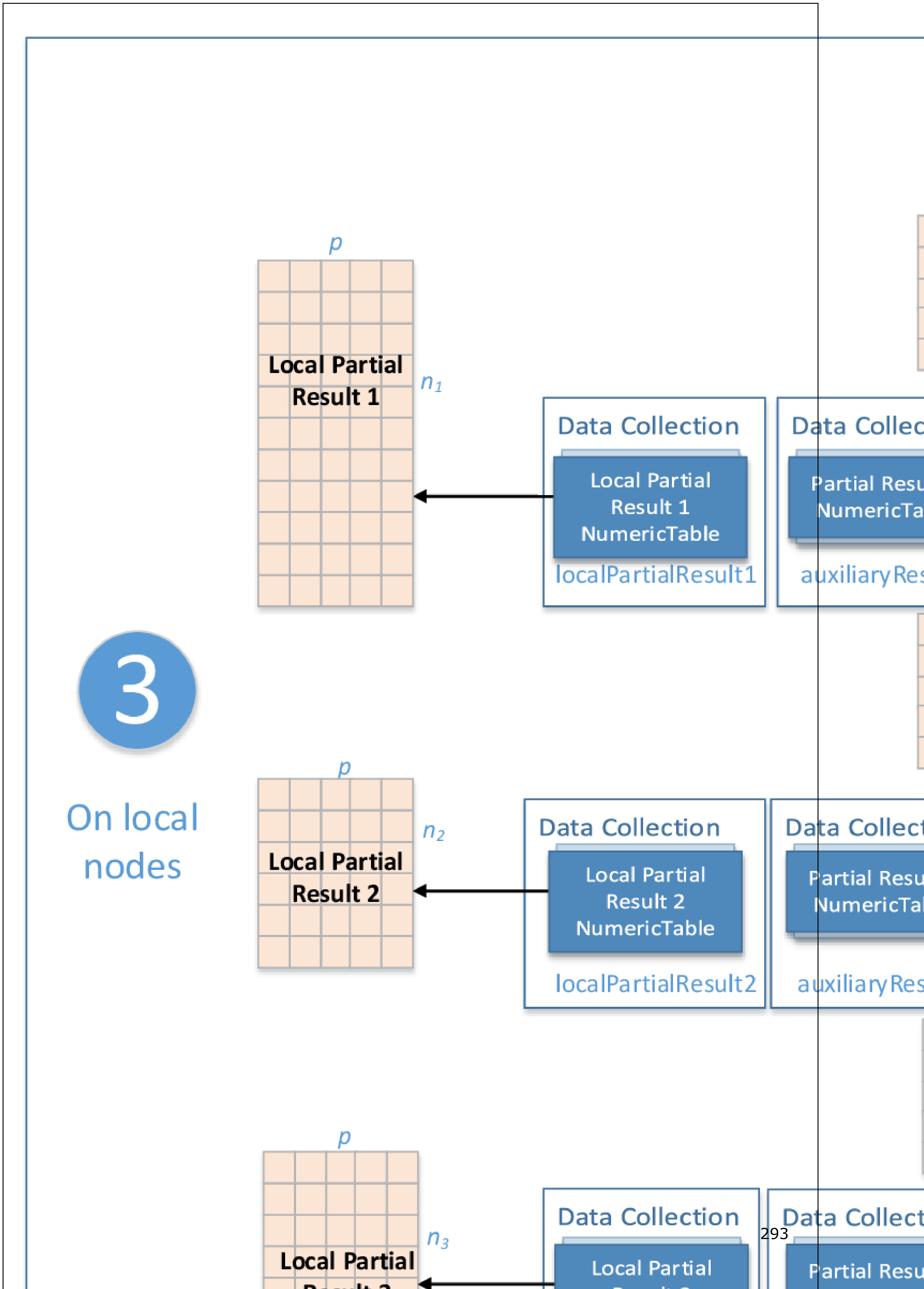
Partial Result ID	Result
`outputOfStep2ForStep3`	A collection that contains numeric tables to be split across local nodes to compute Q_1. **NOTE** By default, these tables are objects of the `HomogenNumericTable` class, but you can define them as objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

Output for QR Decomposition without Pivoting (Distributed Processing, Step 2)

Result ID	Result
`matrixR`	Pointer to the numeric table with the $pimesp$ upper triangular matrix R_1. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

Step 3 - on Local Nodes

QR Decomposition without Pivoting: Distributed Processing, Step 3 - on Local Nodes



In this step, QR decomposition accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for QR Decomposition without Pivoting (Distributed Processing, Step 3)

Input ID	Input
`inputOfStep3FromStep1`	A collection that contains results computed in Step 1 on local nodes (`outputOfStep1ForStep3`). **NOTE** The collection can contain objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` and `PackedTriangularMatrix`.
`inputOfStep3FromStep2`	A collection that contains results computed in Step 2 on local nodes (`outputOfStep2ForStep3`). **NOTE** The collection can contain objects of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

In this step, QR decomposition calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for QR Decomposition without Pivoting (Distributed Processing, Step 3)

Result ID	Result
`matrixQ`	Pointer to the numeric table with the $nimesp$ matrix Q_1. **NOTE** By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Pivoted QR Decomposition

Given the matrix *X* of size $nimesp$, the problem is to compute the QR decomposition with column pivoting $XP = QR$, where

- *Q* is an orthogonal matrix of size $nimesn$
- *R* is a rectangular upper triangular matrix of size $nimesp$
- *P* is a permutation matrix of size $nimesn$

The library requires $n > p$. In this case:

$$XP = QR = [Q_1, Q_2] \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1$$

where the matrix Q_1 has the size $nimesp$ and R_1 has the size $pimesp$.

Batch Processing

Algorithm Input

Pivoted QR decomposition accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Pivoted QR Decomposition (Batch Processing)

Input ID	Input
`data`	Pointer to the numeric table that represents the $nimesp$ matrix *X* to be factorized. The input can be an object of any class derived from `NumericTable`.

Algorithm Parameters

Pivoted QR decomposition has the following parameters:

Algorithm Parameters for Pivoted QR Decomposition (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`permutedColumns`	Not applicable	Pointer to the numeric table with the $1imesp$ matrix with the information for the permutation: • If the *i*-th element is zero, the *i*-th column of the input matrix is a free column and may be permuted with any other free column during the computation. • If the *i*-th element is non-zero, the *i*-th column of the input matrix is moved to the beginning of XP before the computation and remains in its place during the computation. **NOTE** By default, this parameter is an object of the `HomogenNumericTable` class, filled by zeros. However, you can define this parameter as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

Algorithm Output

Pivoted QR decomposition calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Pivoted QR Decomposition (Batch Processing)

Result ID	Result
`matrixQ`	Pointer to the numeric table with the $nimesp$ matrix Q_1.

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`matrixR`	Pointer to the numeric table with the $pimesp$ upper triangular matrix R_1. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.
`permutationMatrix`	Pointer to the numeric table with the $1imesp$ matrix such that $\mathrm{permutationMatrix}(i) = k$ if the column *k* of the full matrix *X* is permuted into the position *i* in *XP*. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix` class, `CSRNumericTable` class, and `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

Examples

C++ (CPU)

Batch Processing:

- pivoted_qr_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- PivotedQRDenseBatch.java

Python*

Batch Processing:

- pivoted_qr_batch.py

Outlier Detection

Outlier detection methods aim to identify observation points that are abnormally distant from other observation points. In oneDAL, the following outlier detection methods are implemented:

- Multivariate Outlier Detection

- Multivariate BACON Outlier Detection
- Univariate Outlier Detection

Multivariate Outlier Detection

In multivariate outlier detection methods, the observation point is the entire feature vector.

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to identify the vectors that do not belong to the underlying distribution (see [Ben2005] for exact definitions of an outlier).

The multivariate outlier detection method takes into account dependencies between features. This method can be parametric, assumes a known underlying distribution for the data set, and defines an outlier region such that if an observation belongs to the region, it is marked as an outlier. Definition of the outlier region is connected to the assumed underlying data distribution.

The following is an example of an outlier region for multivariate outlier detection:

$$\text{Outlier}(\alpha_n, M_n, \Sigma_n) = \{x : \sqrt{(x - M_n) \sum_n -1(x - M_n)} > g(n, \alpha_n)\}$$

where M_n and Sigma_n are (robust) estimates of the vector of means and variance-covariance matrix computed for a given data set, α_n is the confidence coefficient, and $g(n, \alpha_n)$ defines the limit of the region.

Batch Processing

Algorithm Input

The multivariate outlier detection algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Multivariate Outlier Detection (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the data for outlier detection. The input can be an object of any class derived from the `NumericTable` class.
`location`	Pointer to the $1imesp$ numeric table with the vector of means. The input can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.
`scatter`	Pointer to the $pimesp$ numeric table that contains the variance-covariance matrix. The input can be an object of any class derived from `NumericTable` except `PackedTriangularMatrix`.
`threshold`	Pointer to the $1imes1$ numeric table with the non-negative number that defines the outlier region. The input can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

If you do not provide at least one of the `location`, `scatter`, `threshold` inputs, the library will initialize all of them with the following default values:

Default Values for Algorithm Input of Multivariate Outlier Detection (Batch Processing)

`location`	A set of **0.0**
`scatter`	A numeric table with diagonal elements equal to **1.0** and non-diagonal elements equal to **0.0**
`threshold`	**3.0**

Algorithm Parameters

The multivariate outlier detection algorithm has the following parameters:

Algorithm Parameters for Multivariate Outlier Detection (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method.

Algorithm Output

The multivariate outlier detection algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Multivariate Outlier Detection (Batch Processing)

Result ID	Result
`weights`	Pointer to the $nimes1$ numeric table of zeros and ones. Zero in the *i*-th position indicates that the *i*-th feature vector is an outlier. **NOTE** By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- out_detect_mult_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- OutDetectMultDenseBatch.java

Python*

Batch Processing:

- multivariate_outlier_batch.py

Performance Considerations

To get the best overall performance of multivariate outlier detection:

- If input data is homogeneous, provide input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.
- For the default outlier detection method (`defaultDense`), you can benefit from splitting the input data set into blocks for parallel processing.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/ PerformanceIndex. Notice revision #20201201

Multivariate BACON Outlier Detection

In multivariate outlier detection methods, the observation point is the entire feature vector.

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to identify the vectors that do not belong to the underlying distribution using the BACON method (see [Billor2000]).

In the iterative method, each iteration involves several steps:

1. Identify an initial basic subset of $m > p$ feature vectors that can be assumed as not containing outliers. The constant *m* is set to $5p$. The library supports two approaches to selecting the initial subset:

- Based on distances from the medians $||x_i - \text{med}||$, where:
  - **med** is the vector of coordinate-wise medians
  - $||.||$ is the vector norm
  - $i = 1, \ldots, n$
- Based on the Mahalanobis distance $d_i(\text{mean}, S) = \sqrt{(x_i - \text{mean})^T s^{-1}(x_i - \text{mean})}$, where:
  - **mean** and *S* are the mean and the covariance matrix, respectively, of *n* feature vectors
  - $i = 1, \ldots, n$

Each method chooses *m* feature vectors with the smallest values of distances.

2. Compute the discrepancies using the Mahalanobis distance above, where mean and S are the mean and the covariance matrix, respectively, computed for the feature vectors contained in the basic subset.

3. Set the new basic subset to all feature vectors with the discrepancy less than $c_{npr}X^2_{p,\frac{\alpha}{n}}$, where:

- $chi^2_{p,\alpha}$ is the $(1 - \alpha)$ percentile of the Chi-square distribution with *p* degrees of freedom
- $c_{npr} = c_{hr} + c_{np}$, where:

- *r* is the size of the current basic subset
- $c_{hr} = \max\{0, \frac{h-r}{h+r}\}$, where $h = [\frac{n+p+1}{2}]$ and $[]$ is the integer part of a number
- $c_{np} = 1 + \frac{p+1}{n-p} + \frac{2}{n-1-3p}$

4. Iterate steps 2 and 3 until the size of the basic subset no longer changes.
5. Nominate the feature vectors that are not part of the final basic subset as outliers.

Batch Processing

Algorithm Input

The multivariate BACON outlier detection algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Multivariate BACON Outlier Detection (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the data for outlier detection. **NOTE** The input can be an object of any class derived from the `NumericTable` class.

Algorithm Parameters

The multivariate BACON outlier detection algorithm has the following parameters:

Algorithm Parameters for Multivariate BACON Outlier Detection (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`initializationMethod`	`baconMedian`	The initialization method, can be: • `baconMedian` - median-based method • `defaultDense` - Mahalanobis distance-based method
`alpha`	**0.05**	One-tailed probability that defines the $(1 - \alpha)$ quantile of the χ^2 distribution with *p* degrees of freedom. Recommended value: $\frac{\alpha}{n}$, where *n* is the number of observations.
`toleranceToConverge`	**0.005**	The stopping criterion. The algorithm is terminated if the size of the basic subset is changed by less than the threshold.

Algorithm Output

The multivariate BACON outlier detection algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Multivariate BACON Outlier Detection (Batch Processing)

Result ID	Result
`weights`	Pointer to the $nimes1$ numeric table of zeros and ones. Zero in the *i*-th position indicates that the *i*-th feature vector is an outlier. **NOTE** By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except the `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- out_detect_bacon_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- OutDetectBaconDenseBatch.java

Python*

Batch Processing:

- bacon_outlier_batch.py

Univariate Outlier Detection

A univariate outlier is an occurrence of an abnormal value within a single observation point.

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to identify the vectors that do not belong to the underlying distribution (see [Ben2005] for exact definitions of an outlier).

The algorithm for univariate outlier detection considers each feature independently. The univariate outlier detection method can be parametric, assumes a known underlying distribution for the data set, and defines an outlier region such that if an observation belongs to the region, it is marked as an outlier. Definition of the outlier region is connected to the assumed underlying data distribution.

The following is an example of an outlier region for the univariate outlier detection:

$$\text{Outlier}(\alpha_n, m_n, \sigma_n) = \{x : \frac{|x - m_n|}{\sigma_n} > g(n, \alpha_n)\}$$

where m_n and σ_n are (robust) estimates of the mean and standard deviation computed for a given data set, α_n is the confidence coefficient, and $g(n, \alpha_n)$ defines the limits of the region and should be adjusted to the number of observations.

Batch Processing

Algorithm Input

The univariate outlier detection algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Univariate Outlier Detection (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table with the data for outlier detection. **NOTE** The input can be an object of any class derived from the `NumericTable` class.
`location`	Pointer to the $1imesp$ numeric table with the vector of means. **NOTE** The input can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.
`scatter`	Pointer to the $1imesp$ numeric table with the vector of standard deviations. **NOTE** The input can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.
`threshold`	Pointer to the $1imesp$ numeric table with non-negative numbers that define the outlier region. **NOTE** The input can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

If you do not provide at least one of the `location`, `scatter`, `threshold` inputs, the library will initialize all of them with the following default values:

Default Values for Algorithm Input of Univariate Outlier Detection (Batch Processing)

`location`	A set of **0.0**
`scatter`	A set of **1.0**
`threshold`	A set of **3.0**

Algorithm Parameters

The univariate outlier detection algorithm has the following parameters:

Algorithm Parameters for Univariate Outlier Detection (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The univariate outlier detection algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Univariate Outlier Detection (Batch Processing)

Result ID	Result
`weights`	Pointer to the $nimesp$ numeric table of zeros and ones. Zero in the position (i, j) indicates an outlier in the *i*-th observation of the *j*-th feature. **NOTE** By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- out_detect_uni_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- OutDetectUniDenseBatch.java

Python*

Batch Processing:

- univariate_outlier_batch.py

Distance Matrix

Useful measures of similarity between feature vectors.

- Correlation Distance Matrix

- Cosine Distance Matrix

Correlation Distance Matrix

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to compute the symmetric $nimesn$ matrix $D_{\text{cor}} = (d_{ij})$ of distances between feature vectors, where

$$d_{ij} = 1 - \frac{\sum_{k=1}^{p} (x_{ik} - \overline{x_i})(x_{jk} - \overline{x_j})}{\sqrt{\sum_{k=1}^{p} (x_{ik} - \overline{x_i})^2} \sqrt{\sum_{k=1}^{p} (x_{jk} - \overline{x_j})^2}}$$

$$\overline{x_i} = \frac{1}{p} \sum_{k=1}^{p} x_{ik}$$

$$\overline{x_j} = \frac{1}{p} \sum_{k=1}^{p} x_{jk}$$

$$i = \overline{1, n}$$

$$j = \overline{1, n}$$

Batch Processing

Algorithm Input

The correlation distance matrix algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Correlation Distance Matrix (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table for which the distance is computed. The input can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The correlation distance matrix algorithm has the following parameters:

Algorithm Parameters for Correlation Distance Matrix (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The correlation distance matrix algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Correlation Distance Matrix (Batch Processing)

Result ID	Result
`correlationDistance`	Pointer to the numeric table that represents the $nimesn$ symmetric distance matrix D_{cor}. By default, the result is an object of the `PackedSymmetricMatrix` class with the `lowerPackedSymmetricMatrix` layout. However, you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- cor_dist_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- CorDistDenseBatch.java

Python*

Batch Processing:

- correlation_distance_batch.py

Performance Considerations

To get the best overall performance when computing the correlation distance matrix:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Cosine Distance Matrix

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots x_n = (x_{n1}, \ldots, x_{np})$ of dimension Lmath:**p**, the problem is to compute the symmetric $nimesn$ matrix $D_{\text{cos}} = (d_{ij})$ of distances between feature vectors, where

$$d_{ij} = 1 - \frac{\sum_{k=1}^{p} x_{ik} x_{jk}}{\sqrt{\sum_{k=1}^{p} x_{ik}^2} \sqrt{\sum_{k=1}^{p} x_{jk}^2}}$$

$$i = \overline{1, n}$$
$$j = \overline{1, n}$$

Batch Processing

Algorithm Input

The cosine distance matrix algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Cosine Distance Matrix (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table for which the distance is computed. The input can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The cosine distance matrix algorithm has the following parameters:

Algorithm Parameters for Cosine Distance Matrix (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The cosine distance matrix algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Cosine Distance Matrix (Batch Processing)

Result ID	Result
`cosineDistance`	Pointer to the numeric table that represents the $nimesn$ symmetric distance matrix $D_{\cos}$. By default, the result is an object of the `PackedSymmetricMatrix` class with the `lowerPackedSymmetricMatrix` layout. However, you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- cos_dist_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- CosDistDenseBatch.java

Python*

Batch Processing:

- cosine_distance_batch.py

Performance Considerations

To get the best overall performance when computing the cosine distance matrix:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Distributions

Random number distribution generators are used to generate random numbers with different types of the discrete and continuous distributions. The numbers are generated by transforming uniformly distributed variates in accordance with the required cumulative distribution function (CDF).

In oneDAL, distribution represents an algorithm interface that runs in-place initialization of memory according to the required CDF.

- Uniform Distribution
- Normal Distribution
- Bernoulli Distribution

Algorithm Input

Distribution algorithms accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Distributions

Input ID	Input
`tableToFill`	Pointer to the numeric table of size $nimesp$. **NOTE** This input can be an object of any class derived from `NumericTable` except `CSRNumericTable`, `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `MergedNumericTable` when it holds one of the above table types.

Algorithm Parameters

Distribution algorithms have the following common parameter:

Algorithm Parameters for Distributions

Parameter	Default Value	Description
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number engine.

Algorithm Output

Distribution algorithms calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Distributions

Result ID	Result
`randomNumbers`	Pointer to the $nimesp$ numeric table with algorithm results. In oneDAL, distribution algorithms are in-place, which means that the algorithm does not allocate memory for the distribution result, but returns pointer to the filled input.

Uniform Distribution

Generates random numbers uniformly distributed on the interval $[a, b)$.

Details

Uniform random number generator fills the input $nimesp$ numeric table with values that are uniformly distributed on the interval $[a, b)$, where $a, b \in \setminus$ and $a <> b$.

The probability density is given by:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b) \\ 0, & x \notin [a, b) \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x < b \\ 1, & x \geq b \end{cases}$$

Batch Processing

Algorithm Parameters

Uniform distribution algorithm has the following parameters in addition to the common parameters specified in Distributions:

Algorithm Parameters for Uniform Distribution (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.

Parameter	Default Value	Description
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`a`	**0.0**	The left bound *a*.
`b`	**1.0**	The right bound *b*.

Examples

Python*

Batch Processing:

- distributions_uniform_batch.py

Normal Distribution

Generates normally distributed random numbers.

Details

Normal (Gaussian) random number generator fills the input n x p numeric table with Gaussian random numbers with mean ɑ and standard deviation σ, where ɑ, σ∈R and σ > 0. The probability density function is given by:

$$f_{\alpha,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-a)^2}{2\sigma^2}), -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$F_{\alpha,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y-a)^2}{2\sigma^2}) dy, -\infty < x < +\infty$$

Batch Processing

Algorithm Parameters

Normal distribution algorithm has the following parameters in addition to the common parameters specified in Distributions:

Algorithm Parameters for Normal Distribution (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm. The only method supported so far is the Inverse Cumulative Distribution Function (ICDF) method.
`a`	**0**	The mean α
`sigma`	**1**	The standard deviation σ

Examples

Python*

Batch Processing:

- distributions_normal_batch.py

Bernoulli Distribution

Generates Bernoulli distributed random numbers.

Details

Bernoulli random number generator fills the $nimesp$ numeric table with Bernoulli distributed values with the *p* probability of success on a single trial, where $p \in R$, $0 \leq p \leq 1$.

A variate is called Bernoulli distributed if after a trial it is equal to **1** with the probability of success *p* and to **0** with the probability $1 - p$. The probability distribution is given by:

$$p\{x = 1\} = p$$
$$p\{x = 0\} = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in \mathbb{R} \\ 1, & x \geq 1 \end{cases}$$

Batch Processing

Algorithm Parameters

Bernoulli distribution algorithm has the following parameters in addition to the common parameters specified in Distributions:

Algorithm Parameters for Bernoulli Distribution (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`p`	Not applicable	Success probability of a trial, required parameter.

Examples

Python*

Batch Processing:

- distributions_bernoulli_batch.py

Performance Considerations

To get the best overall performance when using the Bernoulli distribution random number generator, provide the 32-bit signed integer homogeneous numeric table constructed with enabled equal features.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Engines

Random number engines are used for uniformly distributed random numbers generation by using a seed - the initial value that allows to select a particular random number sequence. Initialization is an engine-specific procedure.

Algorithm Input

Engines accept the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Engines

Input ID	Input
`tableToFill`	Pointer to the numeric table of size $nimesp$. This input can be an object of any class derived from `NumericTable` except `CSRNumericTable`, `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `MergedNumericTable` when it holds one of the above table types.

Algorithm Output

Engines calculate the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Engines

Result ID	Result
`randomNumbers`	Pointer to the $nimesp$ numeric table with generated random floating-point values of single or double precision. In oneDAL, engines are in-place, which means that the algorithm does not allocate memory for the distribution result, but returns pointer to the filled input.

NOTE In the current version of the library, engines are used for random number generation only as a parameter of another algorithm.

Parallel Random Number Generation

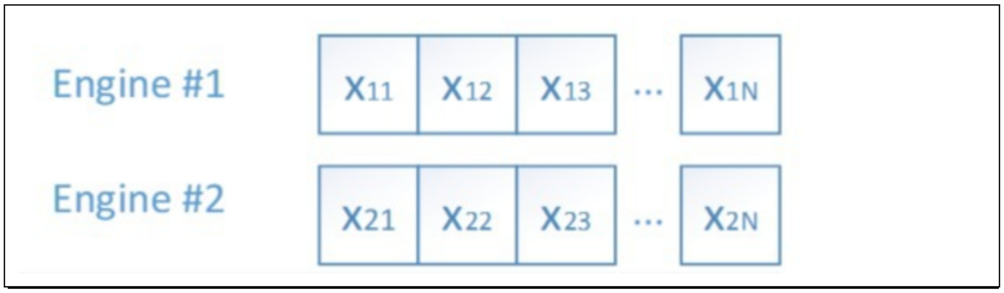
The following methods that support generation of sequences of random numbers in parallel are supported in library:

Family

Engines follow the same algorithmic scheme with different algorithmic parameters. The set of the parameters guarantee independence of random number sequences produced by the engines.

The example below demonstrates the idea for the case when 2 engines from the same family are used to generate 2 random sequences:

Family method of random sequence generation

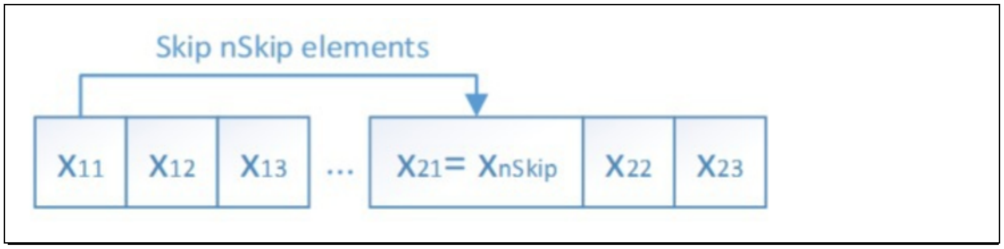


SkipAhead

This method skips `nskip` elements of the original random sequence. This method allows to produce `nThreads` non-overlapping subsequences.

The example below demonstrates the idea for the case when 2 subsequences are used from the random sequence:

SkipAhead method of random sequence generation

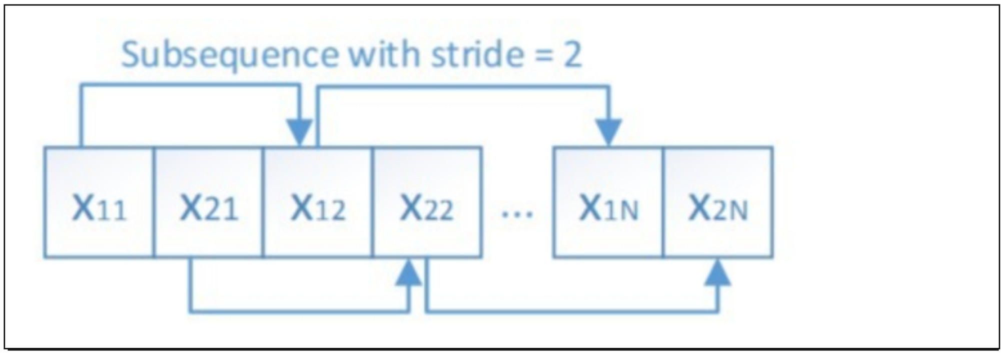


LeapFrog

This method generates random numbers with a stride of `nThreads`. `threadIdx` is an index of the current thread.

The example below demonstrates the idea for the case when 2 subsequences are used from the random sequence:

LeapFrog method of random sequence generation



These methods are represented with member functions of classes that represent functionality described in the Engines section. See API References for details.

NOTE Support of these methods is engine-specific.

- mt19937
- mcg59
- mt2203

mt19937

Mersenne Twister engine is a random number engine based on Mersenne Twister algorithm. More specifically, it is a Mersenne Twister pseudorandom number generator with period $2^{19937} - 1$[Matsumoto98].

Subsequence selection methods support

skipAhead (nskip)	Supported
leapfrog (threadIdx, nThreads)	Not supported

Batch Processing

Mersenne Twister engine needs the initial condition (`seed`) for state initialization. The seed can be either an integer scalar or a vector of *p* integer elements, the inputs to the respective engine constructors.

Algorithm Parameters

Mersenne Twister engine has the following parameters:

Algorithm Parameters for mt19937 engine (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method; the only method supported by the algorithm.
`seed`	• **777** for a scalar seed • NA for a vector seed	Initial condition for state initialization, scalar or vector: • Scalar, value of `size_t` type • Vector, pointer to `HomogenNumericTable` of size $1imesp$

mcg59

The engine is based on the 59-bit multiplicative congruential generator.

Subsequence selection methods support

skipAhead (nskip)	Supported
leapfrog (threadIdx, nThreads)	Supported

Batch Processing

MCG59 engine needs the initial condition (`seed`) for state initialization. The seed can be either an integer scalar or a vector of *p* integer elements, the inputs to the respective engine constructors.

Algorithm Parameters

MCG59 engine has the following parameters:

Algorithm Parameters for mcg58 engine (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method; the only method supported by the algorithm.
`seed`	• **777** for a scalar seed • NA for a vector seed	Initial condition for state initialization, scalar or vector: • Scalar, value of `size_t` type • Vector, pointer to `HomogenNumericTable` of size $1imesp$

mt2203

The engine is based on a set of 6024 Mersenne Twister pseudorandom number generators with period 22203.

MT2203 generators are intended for use in large scale Monte Carlo simulations performed on multi-processor computer systems [Matsumoto2000].

Subsequence selection methods support

skipAhead (nskip)	Not supported
leapfrog (threadIdx, nThreads)	Not supported

Batch Processing

Mersenne Twister engine needs the initial condition (`seed`) for state initialization. The seed can be either an integer scalar or a vector of *p* integer elements, the inputs to the respective engine constructors.

Algorithm Parameters

MT2203 engine has the following parameters:

Algorithm Parameters for mt2203 engine (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method; the only method supported by the algorithm.
`seed`	• **777** for a scalar seed • NA for a vector seed	Initial condition for state initialization, scalar or vector: • Scalar, value of `size_t` type • Vector, pointer to `HomogenNumericTable` of size $1imesp$

Moments of Low Order

Moments are basic quantitative measures of data set characteristics such as location and dispersion. oneDAL computes the following low order characteristics:

- minimums/maximums
- sums
- means
- sums of squares
- sums of squared differences from the means
- second order raw moments
- variances
- standard deviations
- variations

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to compute the following sample characteristics for each feature in the data set:

Moments of Low Order

Statistic	Definition
Minimum	$min(j) = \min_i \{x_{ij}\}$
Maximum	$max(j) = \max_i \{x_{ij}\}$
Sum	$s(j) = \sum_i x_{ij}$
Sum of squares	$s_2(j) = \sum_i x_{ij}^2$
Means	$m(j) = \frac{s(j)}{n}$
Second order raw moment	$a_2(j) = \frac{s_2(j)}{n}$
Sum of squared difference from the means	$\mathrm{SDM}(j) = \sum_i (x_{ij} - m(j))^2$
Variance	$k_2(j) = \frac{\mathrm{SDM}(j)}{n-1}$
Standard deviation	$\mathrm{stdev}(j) = \sqrt{k_2(j)}$
Variation coefficient	$V(j) = \frac{\mathrm{stdev}(j)}{m(j)}$

Computation

The following computation modes are available:

- Batch Processing

- Online Processing
- Distributed Processing

Examples

C++ (CPU)

Batch Processing:

- low_order_moms_dense_batch.cpp
- low_order_moms_csr_batch.cpp

Online Processing:

- low_order_moms_dense_online.cpp
- low_order_moms_csr_online.cpp

Distributed Processing:

- low_order_moms_dense_distr.cpp
- low_order_moms_csr_distr.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- LowOrderMomsDenseBatch.java
- LowOrderMomsCSRBatch.java

Online Processing:

- LowOrderMomsDenseOnline.java
- LowOrderMomsCSROnline.java

Distributed Processing:

- LowOrderMomsDenseDistr.java
- LowOrderMomsCSRDistr.java

Python* with DPC++ support

Batch Processing:

- low_order_moms_dense_batch.py

Online Processing:

- low_order_moms_streaming.py

Python*

Batch Processing:

- low_order_moms_dense_batch.py

Online Processing:

- low_order_moms_streaming.py

Distributed Processing:

- low_order_moms_spmd.py

Batch Processing

Algorithm Input

The low order moments algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Low Order Moments (Batch Processing)

Input ID	Input
`data`	Pointer to the numeric table of size $nimesp$ to compute moments for. While the input for `defaultDense`, `singlePassDense`, or `sumDense` method can be an object of any class derived from `NumericTable`, the input for `fastCSR`, `singlePassCSR`, or `sumCSR` method can only be an object of the `CSRNumericTable` class.

Algorithm Parameters

The low order moments algorithm has the following parameters:

Algorithm Parameters for Low Order Moments (Batch Processing)

Parameter	Default Valude	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for computation of low order moments: For CPU: • `defaultDense` - default performance-oriented method • `singlePassDense` - implementation of the single-pass algorithm proposed by D.H.D. West • `sumDense` - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined • `fastCSR` - performance-oriented method for CSR numeric tables • `singlePassCSR` - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables • `sumCSR` - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined For GPU: • `defaultDense` - default performance-oriented method
`estimatesToCompute`	`estimatesAll`	Estimates to be computed by the algorithm: • `estimatesAll` - all supported moments • `estimatesMinMax` - minimum and maximum • `estimatesMeanVariance` - mean and variance

Algorithm Output

The low order moments algorithm calculates the results described in the following table. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

NOTE Each result is a pointer to the $1imesp$ numeric table that contains characteristics for each feature in the data set. By default, the tables are objects of the `HomogenNumericTable` class, but you can define each table as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Algorithm Output for Low Order Moments (Batch Processing)

Result ID	Characteristic
`minimum`	Minimums
`maximum`	Maximums
`sum`	Sums
`sumSquares`	Sums of squares
`sumSquaresCentered`	Sums of squared differences from the means
`mean`	Estimates for the means
`secondOrderRawMoment`	Estimates for the second order raw moments
`variance`	Estimates for the variances
`standardDeviation`	Estimates for the standard deviations
`variation`	Estimates for the variations

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Online Processing

Online processing computation mode assumes that data arrives in blocks $i = 1, 2, 3, \ldots \text{nblocks}$.

Computation of low order moments in the online processing mode follows the general computation schema for online processing described in Algorithms.

Algorithm Input

The low order moments algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Low Order Moments (Online Processing)

Input ID	Input
`data`	Pointer to the numeric table of size $n_i \times p$ that represents the current data block.

Input ID	Input
	While the input for defaultDense, singlePassDense, or sumDense method can be an object of any class derived from NumericTable, the input for fastCSR, singlePassCSR, or sumCSR method can only be an object of the CSRNumericTable class.

Algorithm Parameters

The low order moments algorithm has the following parameters:

Algorithm Parameters for Low Order Moments (Online Processing)

Paramete r	Default Valude	Description
algorith mFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultD ense	Available methods for computation of low order moments: defaultDense — default performance-oriented method singlePassDense — implementation of the single-pass algorithm proposed by D.H.D. West sumDense — implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined fastCSR — performance-oriented method for CSR numeric tables singlePassCSR — implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables sumCSR — implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
initiali zationPr ocedure	Not applicable	The procedure for setting initial parameters of the algorithm in the online processing mode. By default, the algorithm does the following initialization: • Sets nObservations, partialSum, and partialSumSquares to zero. • Sets partialMinimum and partialMaximum to the first row of the input table.
estimate sToCompu te	estimate sAll	Estimates to be computed by the algorithm: • estimatesAll - all supported moments • estimatesMinMax - minimum and maximum • estimatesMeanVariance - mean and variance

Partial Results

The low order moments algorithm in the online processing mode calculates partial results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Partial Results for Low Order Moments (Online Processing)

Result ID	Result
nObservations	Pointer to the $1imes1$ numeric table that contains the number of rows processed so far. By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except CSRNumericTable.

Partial characteristics computed so far, each in a $1imesp$ numeric table. By default, each table is an object of the HomogenNumericTable class, but you can define the tables as objects of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Partial Characteristics for Low Order Moments (Online Processing)

Result ID	Result
partialMinimum	Partial minimums
partialMaximum	Partial maximums
partialSum	Partial sums
partialSumSquares	Partial sums of squares
partialSumSquaresCentered	Partial sums of squared differences from the means

Algorithm Output

The low order moments algorithm calculates the results described in the following table. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

> **NOTE** Each result is a pointer to the $1imesp$ numeric table that contains characteristics for each feature in the data set. By default, the tables are objects of the HomogenNumericTable class, but you can define each table as an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Algorithm Output for Low Order Moments (Online Processing)

Result ID	Characteristic
minimum	Minimums
maximum	Maximums

Result ID	Characteristic
sum	Sums
sumSquares	Sums of squares
sumSquaresCentered	Sums of squared differences from the means
mean	Estimates for the means
secondOrderRawMoment	Estimates for the second order raw moments
variance	Estimates for the variances
standardDeviation	Estimates for the standard deviations
variation	Estimates for the variations

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Distributed Processing

This mode assumes that the data set is split into nblocks blocks across computation nodes.

Algorithm Parameters

The low order moments algorithm in the distributed processing mode has the following parameters:

Algorithm Parameters for Low Order Moments (Distributed Processing)

Parameter	Default Valude	Description
computeStep	Not applicable	The parameter required to initialize the algorithm. Can be: • step1Local - the first step, performed on local nodes • step2Master - the second step, performed on a master node
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available methods for computation of low order moments: defaultDense: default performance-oriented method singlePassDense: implementation of the single-pass algorithm proposed by D.H.D. West

Parameter	Default Valude	Description	
		sumDense	implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined
		fastCSR	performance-oriented method for CSR numeric tables
		singlePassCSR	implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables
		sumCSR	implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
`estimatesToCompute`	`estimatesAll`	Estimates to be computed by the algorithm: • `estimatesAll` - all supported moments • `estimatesMinMax` - minimum and maximum • `estimatesMeanVariance` - mean and variance	

Computation of low order moments follows the general schema described in Algorithms:

Step 1 - on Local Nodes

In this step, the low order moments algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Low Order Moments (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the numeric table of size $n_i \times p$ that represents the *i*-th data block on the local node. While the input for `defaultDense`, `singlePassDense`, or `sumDense` method can be an object of any class derived from `NumericTable`, the input for `fastCSR`, `singlePassCSR`, or `sumCSR` method can only be an object of the `CSRNumericTable` class.

In this step, the low order moments algorithm calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Low Order Moments (Distributed Processing, Step 1)

Result ID	Result
`nObservations`	Pointer to the $1imes1$ numeric table that contains the number of observations processed so far on the local node. By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `CSRNumericTable`.

Partial characteristics computed so far on the local node, each in a 1imesp numeric table. By default, each table is an object of the `HomogenNumericTable` class, but you can define the tables as objects of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Partial Characteristics for Low Order Moments (Distributed Processing, Step 1)

Result ID	Result
`partialMinimum`	Partial minimums
`partialMaximum`	Partial maximums
`partialSum`	Partial sums
`partialSumSquares`	Partial sums of squares
`partialSumSquaresCentered`	Partial sums of squared differences from the means

Step 2 - on Master Node

In this step, the low order moments algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Low Order Moments (Distributed Processing, Step 2)

Input ID	Input
`partialResults`	A collection that contains numeric tables with partial results computed in Step 1 on local nodes (six numeric tables from each local node). These numeric tables can be objects of any class derived from the `NumericTable` class except `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

In this step, the low order moments algorithm calculates the results described in the following table. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

NOTE Each result is a pointer to the 1imesp numeric table that contains characteristics for each feature in the data set. By default, the tables are objects of the `HomogenNumericTable` class, but you can define each table as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Algorithm Output for Low Order Moments (Distributed Processing, Step 2)

Result ID	Characteristic
`minimum`	Minimums
`maximum`	Maximums
`sum`	Sums

Result ID	Characteristic
`sumSquares`	Sums of squares
`sumSquaresCentered`	Sums of squared differences from the means
`mean`	Estimates for the means
`secondOrderRawMoment`	Estimates for the second order raw moments
`variance`	Estimates for the variances
`standardDeviation`	Estimates for the standard deviations
`variation`	Estimates for the variations

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Quantile

Quantile is an algorithm to analyze the distribution of observations. Quantiles are the values that divide the distribution so that a given portion of observations is below the quantile.

Details

Given a set *X* of *p* features $x_1 = (x_{11}, \ldots, x_{1p}), \ldots x_n = (x_{n1}, \ldots, x_{np})$ and the quantile orders $\beta = \beta_1, \ldots, \beta_m$, the problem is to compute z_{ik} that meets the following conditions:

$$P\{\xi_i \leq z_{ik}\} \geq \beta_k$$
$$P\{\xi_i > z_{ik}\} \leq 1 - \beta_k$$

In the equations above:

- $x_i = (x_{1i}, \ldots, x_{ni})$ are observations of a random variable ξ_i that represents the *i*-th feature
- *P* is the probability measure
- $i = 1, \ldots, p$
- $k = 1, \ldots, m$

Batch Processing

Algorithm Input

The quantile algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Quantile (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table that contains the input data set. This table can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The quantile algorithm has the following parameters:

Algorithm Parameters for Quantile (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`quantileOrders`	**0.5**	The $1 \times m$ numeric table with quantile orders.

Algorithm Output

The quantile algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Quantile (Batch Processing)

Result ID	Result
`quantiles`	Pointer to the $p \times m$ numeric table with the quantiles. By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- quantiles_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- QuantilesDenseBatch.java

Python*

Batch Processing:

- quantiles_batch.py

Quality Metrics

In oneDAL, a quality metric is a numerical characteristic or a set of connected numerical characteristics that represents the qualitative aspect of the result returned by an algorithm: a computed statistical estimate, model, or result of decision making.

A common set of quality metrics can be defined for some training and prediction algorithms.

A typical workflow with quality metric set is the following:

1. Create a quality metric set object to compute quality metrics.
    - Set specific parameters for the algorithms.
    - Use the `useDefaultMetrics` flag to specify whether the default or user-defined quality metrics should be computed.
2. Get an input collection object using `QualityMetricsId` of a specific algorithm.
3. Set data to the input collection using the algorithm's `InputId`.
4. Perform computation.
5. Get the resulting collection of quality metrics using the algorithm's `ResultId`.

> **NOTE** For values of `InputId`, `Parameters`, `QualityMetricsId`, `ResultId`, refer to the description of a specific algorithm.

Quality metrics are optional. They are computed when the computation is explicitly requested.

- Working with the Default Metric Set
    - Quality Metrics for Binary Classification Algorithms
    - Quality Metrics for Multi-class Classification Algorithms
    - Quality Metrics for Linear Regression
    - Quality Metrics for Principal Components Analysis
- Working with User-defined Quality Metrics

Working with the Default Metric Set

For your convenience, oneDAL provides a set of quality metrics for some algorithms.

- Quality Metrics for Binary Classification Algorithms
- Quality Metrics for Multi-class Classification Algorithms
- Quality Metrics for Linear Regression
- Quality Metrics for Principal Components Analysis

Quality Metrics for Binary Classification Algorithms

For two classes $C1$ and $C2$, given a vector $X = (x_1, \dots, x_n)$ of class labels computed at the prediction stage of the classification algorithm and a vector $Y = (y_1, \dots, y_n)$ of expected class labels, the problem is to evaluate the classifier by computing the confusion matrix and connected quality metrics: precision, recall, and so on.

`QualityMetricsId` for binary classification is `confusionMatrix`.

Details

Further definitions use the following notations:

Notations for Quality Metrics for Binary Classification Algorithms

tp	true positive	the number of correctly recognized observations for class C_1
tn	true negative	the number of correctly recognized observations that do not belong to the class C_1
fp	false positive	the number of observations that were incorrectly assigned to the class C_1
fn	false negative	the number of observations that were not recognized as belonging to the class C_1

The library uses the following quality metrics for binary classifiers:

Definitions of Quality Metrics for Binary Classification Algorithms

Quality Metric	Definition
Accuracy	$\frac{\mathrm{tp}+\mathrm{tn}}{\mathrm{tp}+\mathrm{fn}+\mathrm{fp}+\mathrm{tn}}$
Precision	$\frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fp}}$
Recall	$\frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fn}}$
F-score	$\frac{(\beta^2+1)\mathrm{tp}}{(\beta^2+1)\mathrm{tp}+\beta^2\mathrm{fn}+\mathrm{fp}}$
Specificity	$\frac{\mathrm{tn}}{\mathrm{fp}+\mathrm{tn}}$
Area under curve (AUC)	$\frac{1}{2}\left(\frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fn}}+\frac{\mathrm{tn}}{\mathrm{tn}+\mathrm{fp}}\right)$

For more details of these metrics, including the evaluation focus, refer to [Sokolova09].

The confusion matrix is defined as follows:

Confusion Matrix for Binary Classification Algorithms

	Classified as Class C_1	Classified as Class C_2
Actual Class C_1	**tp**	**fn**
Actual Class C_2	**fp**	**tn**

Batch Processing

Algorithm Input

The quality metric algorithm for binary classifiers accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Quality Metrics for Binary Classification (Batch Processing)

Input ID	Input
predictedLabels	Pointer to the $nimes1$ numeric table that contains labels computed at the prediction stage of the classification algorithm. This input can be an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.
groundTruthLabels	Pointer to the $nimes1$ numeric table that contains expected labels. This input can be an object of any class derived from NumericTable except PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.

Algorithm Parameters

The quality metric algorithm has the following parameters:

Algorithm Parameters for Quality Metrics for Binary Classification (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.
beta	**1**	The β parameter of the F-score quality metric provided by the library.

Algorithm Output

The quality metric algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Quality Metrics for Binary Classification (Batch Processing)

Result ID	Result
confusionMatrix	Pointer to the 2×2 numeric table with the confusion matrix. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
binaryMetrics	Pointer to the 1×6 numeric table that contains quality metrics, which you can access by an appropriate Binary Metrics ID: • accuracy - accuracy • precision - precision • recall - recall • fscore - F-score • specificity - specificity • AUC - area under the curve

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- svm_two_class_metrics_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- SVMTwoClassMetricsDenseBatch.java

Quality Metrics for Multi-class Classification Algorithms

For *l* classes $C_1, \ldots, C_l$, given a vector $X = (x_1, \ldots, x_n)$ of class labels computed at the prediction stage of the classification algorithm and a vector $Y = (y_1, \ldots, y_n)$ of expected class labels, the problem is to evaluate the classifier by computing the confusion matrix and connected quality metrics: precision, error rate, and so on.

`QualityMetricsId` for multi-class classification is `confusionMatrix`.

Details

Further definitions use the following notations:

Notations for Quality Metrics for Multi-class Classification Algorithms

tp_i	true positive	the number of correctly recognized observations for class C_1
tn_i	true negative	the number of correctly recognized observations that do not belong to the class C_1
fp_i	false positive	the number of observations that were incorrectly assigned to the class C_1
$ERRORprocessingmath$	false negative	the number of observations that were not recognized as belonging to the class C_1

The library uses the following quality metrics for multi-class classifiers:

Definitions of Quality Metrics for Multi-class Classification Algorithms

Quality Metric	Definition
Average accuracy	$\frac{\sum_{i=1}^{l} \frac{\mathrm{tp}_i+\mathrm{tn}_i}{\mathrm{tp}_i+\mathrm{fn}_i+\mathrm{fp}_i+\mathrm{tn}_i}}{l}$
Error rate	$\frac{\sum_{i=1}^{l} \frac{\mathrm{fp}_i+\mathrm{fn}_i}{\mathrm{tp}_i+\mathrm{fn}_i+\mathrm{fp}_i+\mathrm{tn}_i}}{l}$
Micro precision ($\mathrm{Precision}_\mu$)	$\frac{\sum_{i=1}^{l} \mathrm{tp}_i}{\sum_{i=1}^{l} (\mathrm{tp}_i+\mathrm{fp}_i)}$
Micro recall (Recall_μ)	$\frac{\sum_{i=1}^{l} \mathrm{tp}_i}{\sum_{i=1}^{l} (\mathrm{tp}_i+\mathrm{fn}_i)}$
Micro F-score (F-score_μ)	$\frac{(\beta^2+1)(\mathrm{Precision}_\mu \times \mathrm{Recall}_\mu)}{\beta^2 \times \mathrm{Precision}_\mu + \mathrm{Recall}_\mu}$
Macro precision ($\mathrm{Precision}_M$)	$\frac{\sum_{i=1}^{l} \frac{\mathrm{tp}_i}{\mathrm{tp}_i+\mathrm{fp}_i}}{l}$
Macro recall (Recall_M)	$\frac{\sum_{i=1}^{l} \frac{\mathrm{tp}_i}{\mathrm{tp}_i+\mathrm{fn}_i}}{l}$
Macro F-score (F-score_M)	$\frac{(\beta^2+1)(\mathrm{Precision}_M \times \mathrm{Recall}_M)}{\beta^2 \times \mathrm{Precision}_M + \mathrm{Recall}_M}$

For more details of these metrics, including the evaluation focus, refer to [Sokolova09].

The following is the confusion matrix:

Confusion Matrix for Multi-class Classification Algorithms

	Classified as Class C_1	...	Classified as Class C_i	...	Classified as Class C_l
Actual Class C_1	c_{11}	...	c_{1i}	...	c_{1l}
...	...	...	...	...	...
Actual Class C_i	c_{i1}	...	c_{ii}	...	c_{il}
...	...	...	...	...	...
Actual Class C_l	c_{l1}	...	c_{li}	...	c_{ll}

The positives and negatives are defined through elements of the confusion matrix as follows:

$$\mathrm{tp}_i = c_{ii}$$

$$\mathrm{fp}_i = \sum_{n=1}^{l} c_{ni} - \mathrm{tp}_i$$

$$\mathrm{fn}_i = \sum_{n=1}^{l} c_{in} - \mathrm{tp}_i$$

$$\mathrm{tn}_i = \sum_{n=1}^{l} \sum_{k=1}^{l} c_{nk} - \mathrm{tp}_i - \mathrm{fp}_i - \mathrm{fn}_i$$

Batch Processing

Algorithm Input

The quality metric algorithm for multi-class classifiers accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Quality Metrics for Multi-class Classification Algorithms (Batch Processing)

Input ID	Input
`predictedLabels`	Pointer to the $nimes1$ numeric table that contains labels computed at the prediction stage of the classification algorithm. This input can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.
`groundTruthLabels`	Pointer to the $nimes1$ numeric table that contains expected labels. This input can be an object of any class derived from NumericTable except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Algorithm Parameters

The quality metric algorithm has the following parameters:

Algorithm Parameters for Quality Metrics for Multi-class Classification Algorithms (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`nClasses`	**0**	The number of classes (*l*).
`useDefaultMetrics`	`true`	A flag that defines a need to compute the default metrics provided by the library.
`beta`	**1**	The β parameter of the F-score quality metric provided by the library.

Algorithm Output

The quality metric algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Quality Metrics for Multi-class Classification Algorithms (Batch Processing)

Result ID	Result
confusionMatrix	Pointer to the $\mathrm{nClasses} \times \mathrm{nClasses}$ numeric table with the confusion matrix. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from NumericTable except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
multiClassMetrics	Pointer to the 1×8 numeric table that contains quality metrics, which you can access by an appropriate Multi-class Metrics ID: • `averageAccuracy` - average accuracy • `errorRate` - error rate • `microPrecision` - micro precision • `microRecall` - micro recall • `microFscore` - micro F-score • `macroPrecision` - macro precision • `macroRecall` - macro recall • `macroFscore` - macro F-score **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from NumericTable except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- svm_multi_class_metrics_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- SVMMultiClassMetricsDenseBatch.java

Quality Metrics for Linear Regression

Given a data set $X = (x_i)$ that contains vectors of input variables $x_i = (x_{i1}, \ldots, x_{ip})$, respective responses $z_i = (z_{i1}, \ldots, z_{ik})$ computed at the prediction stage of the linear regression model defined by its coefficients β_{ht}, $h = 1, \ldots, k$, $t = 1, \ldots, p$, and expected responses $y_i = (y_{i1}, \ldots, y_{ik})$, $i = 1, \ldots, n$, the problem is to evaluate the linear regression model by computing the root mean square error, variance-covariance matrix of beta coefficients, various statistics functions, and so on. See Linear Regression for additional details and notations.

For linear regressions, the library computes statistics listed in tables below for testing insignificance of beta coefficients and one of the following values of `QualityMetricsId`:

- `singleBeta` for a single coefficient
- `groupOfBetas` for a group of coefficients

For more details, see [Hastie2009].

Details

The statistics are computed given the following assumptions about the data distribution:

- Responses y_{ij}, $i = 1, \ldots, n$, are independent and have a constant variance σ_j^2, $j = 1, \ldots, k$
- Conditional expectation of responses $y_{.j}$, $j = 1, \ldots, k$, is linear in input variables $x_. = (x_{.1}, \ldots, x_{.p})$
- Deviations of y_{ij}, $i = 1, \ldots, n$, around the mean of expected responses ERM_j, $j = 1, \ldots, k$, are additive and Gaussian.

Testing Insignificance of a Single Beta

The library uses the following quality metrics:

Quality Metrics for Testing Insignificance of a Single Beta

Quality Metric	Definition
Root Mean Square (RMS) Error	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_{ij} - x_{ij})^2}$, $j = 1, \ldots, k$
Vector of variances $\sigma^2 = (\sigma_1^2, \ldots, \sigma_k^2)$	$\sigma_j^2 = \frac{1}{n-p-1}\sum_{i=1}^{n}(y_{ij} - x_{ij})^2$, $j = 1, \ldots, k$
A set of variance-covariance matrices $C = C_1, \ldots, C_k$ for vectors of betas β_{jt}, $j = 1, \ldots, k$	$C_j = (X^T X)^{-1}\sigma_j^2$, $j = 1, \ldots, k$
Z-score statistics used in testing of insignificance of a single coefficient β_{jt}	$\mathrm{zscore}_{jt} = \frac{\beta_{jt}}{\sigma_j \sqrt{v_t}}$, $j = 1, \ldots, k$, σ_j is the *j*-th element of the vector of variance σ^2 and $ERRORprocessingmath$ is the *t*-th diagonal element of the matrix $(X^T X)^{-1}$
Confidence interval for β_{jt}	$(\beta_{jt} - \mathrm{pc}_{1-\alpha}\sqrt{v_t}, \beta_{jt} + \mathrm{pc}_{1-\alpha}\sqrt{v_t})$, $j = 1, \ldots, k$, $\mathrm{pc}_{1-\alpha}$ is the $(1-\alpha)$ percentile of the Gaussian distribution, σ_j is the *j*-th element of the vector of variance σ^2, $ERRORprocessingmath$ is the *t*-th diagonal element of the matrix $(X^T X)^{-1}$

Testing Insignificance of a Group of Betas

The library uses the following quality metrics:

Quality Metrics for Testing Insignificance of a Group of Betas

Quality Metric	Definition
Mean of expected responses, $\mathrm{ERM} = (\mathrm{ERM}_1, \ldots, \mathrm{ERM}_k)$	$\mathrm{ERM}_j = \frac{1}{n}\sum_{i=1}^{n} y_{ij}$, $j = 1, \ldots, k$
Variance of expected responses, $\mathrm{ERV} = (\mathrm{ERV}_1, \ldots, \mathrm{ERV}_k)$	$\mathrm{ERV}_j = \frac{1}{n-1}\sum_{i=1}^{n}(y_{ij} - \mathrm{ERM}_j)^2$, $j = 1, \ldots, k$
Regression Sum of Squares $\mathrm{RegSS} = (\mathrm{RegSS}_1, \ldots, \mathrm{RegSS}_k)$	$\mathrm{RegSS}_j = \frac{1}{n}\sum_{i=1}^{n}(z_{ij} - \mathrm{ERM}_j)^2$, $j = 1, \ldots, k$
Sum of Squares of Residuals $\mathrm{ResSS} = (\mathrm{ResSS}_1, \ldots, \mathrm{ResSS}_k)$	$\mathrm{ResSS}_j = \sum_{i=1}^{n}(y_{ij} - z_{ij})^2$, $j = 1, \ldots, k$
Total Sum of Squares $\mathrm{TSS} = (\mathrm{TSS}_1, \ldots, \mathrm{TSS}_k)$	$\mathrm{TTS}_j = \sum_{i=1}^{n}(y_{ij} - \mathrm{ERM}_j)^2$, $j = 1, \ldots, k$
Determination Coefficient $R^2 = (R_1^2, \ldots, R_k^2)$	$R_j^2 = \frac{\mathrm{RegSS}_j}{\mathrm{TTS}_j}$, $j = 1, \ldots, k$
F-statistics used in testing insignificance of a group of betas $F = (F_1, \ldots, F_k)$	$F_j = \frac{(\mathrm{ResSS}_{0j} - \mathrm{ResSS}_j)/(p - p_0)}{\mathrm{ResSS}_j/(n-p-1)}$, $j = 1, \ldots, k$, where ResSS_j are computed for a model with $p+1$ betas and ResSS_{0j} are computed for a reduced model with $p_0 + 1$ betas ($p - p_0$ betas are set to zero)

Batch Processing

- Testing Insignificance of a Single Beta
- Testing Insignificance of a Group of Betas

Testing Insignificance of a Single Beta

Algorithm Input

The quality metric algorithm for linear regression accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Testing Insignificance of a Single Beta in Linear Regression (Batch Processing)

Input ID	Input
`expectedResponses`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables) used for training the linear regression model. This table can be an object of any class derived from `NumericTable`.

Input ID	Input
`model`	Pointer to the model computed at the training stage of the linear regression algorithm. The model can only be an object of the `linear_regression::Model` class.
`predictedResponses`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables) computed at the prediction stage of the linear regression algorithm. This table can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The quality metric algorithm for linear regression has the following parameters:

Algorithm Parameters for Testing Insignificance of a Single Beta in Linear Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`alpha`	**0.05**	Significance level used in the computation of confidence intervals for coefficients of the linear regression model.
`accuracyThreshold`	**0.001**	Values below this threshold are considered equal to it.

Algorithm Output

The quality metric algorithm for linear regression calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Testing Insignificance of a Single Beta in Linear Regression (Batch Processing)

Result ID	Result
`rms`	Pointer to the $1imesk$ numeric table that contains root mean square errors computed for each response (dependent variable) **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`variance`	Pointer to the $1imesk$ numeric table that contains variances $\sigma_j^2, j = 1, \ldots, k$ computed for each response (dependent variable).

Result ID	Result
	NOTE By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`betaCovariances`	Pointer to the DataCollection object that contains *k* numeric tables, each with the $m \times m$ variance-covariance matrix for betas of the j-th response (dependent variable), where m is the number of betas in the model (m is equal to p when interceptFlag is set to false at the training stage of the linear regression algorithm; otherwise, m is equal to p + 1). The collection can contain objects of any class derived from `NumericTable`.
`zScore`	Pointer to the $k \times m$ numeric table that contains the Z-score statistics used in the testing of insignificance of individual linear regression coefficients, where *m* is the number of betas in the model (*m* is equal to *p* when `interceptFlag` is set to `false` at the training stage of the linear regression algorithm; otherwise, *m* is equal to $p + 1$). **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`confidenceIntervals`	Pointer to the $k \times 2 \times m$ numeric table that contains limits of the confidence intervals for linear regression coefficients: • $\text{confidenceIntervals}[t][2 * j]$ is the left limit of the confidence interval computed for the *j*-th beta of the *t*-th response (dependent variable) • $\text{confidenceIntervals}[t][2 * j + 1]$ is the right limit of the confidence interval computed for the *j*-th beta of the *t*-th response (dependent variable), where *m* is the number of betas in the model (*m* is equal to *p* when `interceptFlag` is set to `false` at the training stage of the linear regression algorithm; otherwise, *m* is equal to $p + 1$). **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`inverseOfXtX`	Pointer to the $m \times m$ numeric table that contains the $(X^T X)^{-1}$ matrix, where *m* is the number of betas in the model (*m* is equal to *p* when `interceptFlag` is set to `false` at the training stage of the linear regression algorithm; otherwise, *m* is equal to $p + 1$).

Testing Insignificance of a Group of Betas

Algorithm Input

The quality metric algorithm for linear regression accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Testing Insignificance of a Group of Betas in Linear Regression (Batch Processing)

Input ID	Input
`expectedResponses`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables) used for training the linear regression model. This table can be an object of any class derived from `NumericTable`.
`predictedResponses`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables) computed at the prediction stage of the linear regression algorithm. This table can be an object of any class derived from `NumericTable`.
`predictedReducedModelResponses`	Pointer to the $nimesk$ numeric table with responses (*k* dependent variables) computed at the prediction stage of the linear regression algorithm using the reduced linear regression model, where $p - p_0$ out of *p* beta coefficients are set to zero. This table can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The quality metric algorithm for linear regression has the following parameters:

Algorithm Parameters for Testing Insignificance of a Group of Betas in Linear Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`numBeta`	**0**	Number of beta coefficients used for prediction.
`numBetaReducedModel`	**0**	Number of beta coefficients (p_0) used for prediction with the reduced linear regression model, where $p - p_0$ out of *p* beta coefficients are set to zero.

Algorithm Output

The quality metric algorithm for linear regression calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Testing Insignificance of a Group of Betas in Linear Regression (Batch Processing)

Result ID	Result
`expectedMeans`	Pointer to the $1imesk$ numeric table that contains the mean of expected responses computed for each dependent variable.
`expectedVariance`	Pointer to the $1imesk$ numeric table that contains the variance of expected responses computed for each dependent variable.

Result ID	Result
regSS	Pointer to the $1 \times k$ numeric table that contains the regression sum of squares computed for each dependent variable.
resSS	Pointer to the $1 \times k$ numeric table that contains the sum of squares of residuals computed for each dependent variable.
tSS	Pointer to the $1 \times k$ numeric table that contains the total sum of squares computed for each dependent variable.
determinationCoeff	Pointer to the $1 \times k$ numeric table that contains the determination coefficient computed for each dependent variable.
fStatistics	Pointer to the $1 \times k$ numeric table that contains the F-statistics computed for each dependent variable.

NOTE By default, these results are objects of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- lin_reg_metrics_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- LinRegMetricsDenseBatch.java

Quality Metrics for Principal Components Analysis

Given the results of the PCA algorithm, data set $E = (e_i)$, $i = \overline{1, p}$ of eigenvalues in decreasing order, full number of principal components *p* and reduced number of components $p_r \leq p$, the problem is to evaluate the explained variances radio and noise variance.

`QualityMetricsId` for the PCA algorithm is `explainedVarianceMetrics`.

Details

The metrics are computed given the input data meets the following requirements:

- At least the largest eigenvalue e_0 is non-zero. Returns an error otherwise.
- The number of eigenvalues *p* must be equal to the number of features provided. Returns an error if *p* is less than the number of features.

The PCA algorithm receives input argument eigenvalues e_k, $k = \overline{1,p}$. It represents the following quality metrics:

- Explained variance ratio
- Noise variance

The library uses the following quality metrics:

Quality Metrics for Principal Components Analysis

Quality Metric	Definition
Explained variance	e_k, $k = \overline{1,p}$
Explained variance ratios	$r_k = \frac{e_k}{\sum_{i=1}^{p} e_i}$, $k = \overline{1,p}$
Noise variance	$v_{\text{noise}} = \begin{cases} 0, & p_r = p; \\ \frac{1}{p-p_r} \sum_{i=p_r+1}^{p} e_i, & p_r < p \end{cases}$

> **NOTE** Quality metrics for PCA are correctly calculated only if the eigenvalues vector obtained from the PCA algorithm has not been reduced. That is, the nComponents parameter of the PCA algorithm must be zero or equal to the number of features. The formulas rely on a full set of the principal components. If the set is reduced, the result is considered incorrect.

Batch Processing

Algorithm Input

The Quality Metrics for PCA algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Quality Metrics for Principal Components Analysis (Batch Processing)

Input ID	Input
`eigenvalues`	*p* eigenvalues (explained variances), numeric table of size $1limesp$. You can define it as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Algorithm Parameters

The quality metric algorithm has the following parameters:

Algorithm Parameters for Quality Metrics for Principal Components Analysis (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`nComponents`	**0**	The number of principal components $p_r \leq p$ to compute metrics for. If it is zero, the algorithm will compute the result for *p*.

Parameter	Default Value	Description
`nFeatures`	**0**	The number of features in the data set used as input in PCA algorithm. If it is zero, the algorithm will compute the result for p. **NOTE** if $\mathrm{nFeatures} \neq p$, the algorithm will return non-relevant results.

Algorithm Output

The quality metric for PCA algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Algorithm Output for Quality Metrics for Principal Components Analysis (Batch Processing)

Result ID	Result
`explainedVariances`	Pointer to the $1 \times p_r$ numeric table that contains a reduced eigenvalues array.
`explainedVariancesRatios`	Pointer to the $1 \times p_r$ numeric table that contains an array of reduced explained variances ratios.
`noiseVariance`	Pointer to the $1imes1$ numeric table that contains noise variance.

> **NOTE** By default, each numeric table specified by the collection elements is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, except for `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and CSRNumericTable.

Examples

C++ (CPU)

Batch Processing:

- pca_metrics_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- PCAMetricsDenseBatch.java

Working with User-defined Quality Metrics

In addition to or instead of the metrics available in the library, you can use your own quality metrics. To do this:

1. Add your own implementation of the quality metrics algorithm and define Input and Result classes for that algorithm.
2. Register this new algorithm in the `inputAlgorithms` collection of the quality metric set. Also register the input objects for the new algorithm in the `inputData` collection of the quality metric set.

Use the unique key when registering the new algorithm and its input, and use the same key to obtain the computed results.

Sorting

In oneDAL sorting is an algorithm to sort the observations by each feature (column) in the ascending order.

The result of the sorting algorithm applied to the matrix $X = (x_{ij})_{n \times p}$ is the matrix $Y = (y_{ij})_{n \times p}$ where the *j*-th column $(Y)_j = (y_{ij})$, $i = 1, \ldots, n$, is the column $(X)_j = (x_{ij})$, $i = 1, \ldots, n$, sorted in the ascending order.

Batch Processing

Algorithm Input

The sorting algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Sorting (Batch Processing)

Input ID	Input
`data`	Pointer to the $nimesp$ numeric table that contains the input data set. This table can be an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Algorithm Parameters

The sorting algorithm has the following parameters:

Algorithm Parameters for Sorting (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The radix method for sorting a data set, the only method supported by the algorithm.

Algorithm Output

The sorting algorithm function calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Sorting (Batch Processing)

Result ID	Result
`sortedData`	Pointer to the $nimesp$ numeric table that stores the results of sorting.

NOTE If the number of feature vectors is greater than or equal to 2^{31}, the library uses the quick sort method instead of radix sort.

Examples

C++ (CPU)

Batch Processing:

- sorting_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- SortingDenseBatch.java

Python*

Batch Processing:

- sorting_batch.py

Normalization

Normalization is a set of algorithms intended to transform data before feeding it to some classes of algorithms, for example, classifiers [James2013]. Normalization may improve computation accuracy and efficiency. Different rules can be used to normalize data. In oneDAL, two techniques to normalize data are implemented: z-score and min-max.

- Z-score
- Min-max

Z-score

Z-score normalization is an algorithm that produces data with each feature (column) having zero mean and unit variance.

Details

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to compute the matrix $Y = (y_{ij})$ of dimension $nimesp$ as following:

$$y_{ij} = \frac{x_{ij} - m_j}{\Delta}$$

where:

- m_j is the mean of *j*-th component of set $(X)_j$, where $j = \overline{1, p}$
- value of Δ depends omn a computation mode

oneDAL provides two modes for computing the result matrix. You can enable the mode by setting the flag `doScale` to a certain position (for details, see Algorithm Parameters). The mode may include:

- **Centering only.** In this case, $\Delta = 1$ and no scaling is performed. After normalization, the mean of *j*-th component of result set $(Y)_j$ will be zero.
- **Centering and scaling.** In this case, $\Delta = \sigma_j$, where σ_j is the standard deviation of *j*-th component of set $(X)_j$. After normalization, the mean of *j*-th component of result set $(Y)_j$ will be zero and its variance will get a value of one.

> **NOTE** Some algorithms require normalization parameters (mean and variance) as an input. The implementation of Z-score algorithm in oneDAL does not return these values by default. Enable this option by setting the resultsToCompute flag. For details, see Algorithm Parameters.

Batch Processing

Algorithm Input

Z-score normalization algorithm accepts an input as described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Z-score (Batch Processing)

Input ID	Input
`data`	Pointer to the numeric table of size $nimesp$. **NOTE** This table can be an object of any class derived from `NumericTable`.

Algorithm Parameters

Z-score normalization algorithm has the following parameters. Some of them are required only for specific values of the computation method parameter `method`:

Algorithm Parameters for Z-score (Batch Processing)

Parameter	method	Default Value	Description
`algorithmFPType`	`defaultDense` or `sumDense`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	Not applicable	`defaultDense`	Available computation methods: defaultDense — a performance-oriented method. Mean and variance are computed by low order moments algorithm. For details, see Batch Processing for Moments of Low Order. sumDense — a method that uses the basic statistics associated with the numeric table of pre-computed sums. Returns an error if pre-computed sums are not defined.

Parameter	method	Default Value	Description
`moments`	`defaultDense`	**SharedPtr<low_order_moments::Batch<algorithmFPType, low_order_moments::defaultDense> >**	Pointer to the low order moments algorithm that computes means and standard deviations to be used for Z-score normalization with the `defaultDense` method.
`doScale`	`defaultDense` or `sumDense`	`true`	If true, the algorithm applies both centering and scaling. Otherwise, the algorithm provides only centering.
`resultsToCompute`	`defaultDense` or `sumDense`	Not applicable	*Optional*. Pointer to the data collection containing the following key-value pairs for Z-score: • `mean` - means • `variance` - variances Provide one of these values to request a single characteristic or use bitwise OR to request a combination of them.

Algorithm Output

Z-score normalization algorithm calculates the result as described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Z-score (Batch Processing)

Result ID	Result
`normalizedData`	Pointer to the $nimesp$ numeric table that stores the result of normalization. **NOTE** By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`means`	*Optional*. Pointer to the $1imesp$ numeric table that contains mean values for each feature. If the function result is not requested through the `resultsToCompute` parameter, the numeric table contains a `NULL` pointer.
`variances`	*Optional*.

Result ID	Result
	Pointer to the $1imesp$ numeric table that contains variance values for each feature. If the function result is not requested through the `resultsToCompute` parameter, the numeric table contains a `NULL` pointer. -

NOTE By default, each numeric table specified by the collection elements is an object of the `HomogenNumericTable` class. You can also define the result as an object of any class derived from `NumericTable`, except for `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- zscore_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- ZScoreDenseBatch.java

Python*

Batch Processing:

- normalization_zscore_batch.py

Min-max

Min-max normalization is an algorithm to linearly scale the observations by each feature (column) into the range $[a, b]$.

Problem Statement

Given a set *X* of *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of dimension *p*, the problem is to compute the matrix $Y = (y_{ij})_{n \times p}$ where the *j*-th column $(Y)_j = (y_{ij})_{i=1,\ldots,n}$ is obtained as a result of normalizing the column $(X)_j = (x_{ij})_{i=1,\ldots,n}$ of the original matrix as:

$$y_{ij} = a + \frac{x_{ij} - \min(j)}{\max(j) - \min(j)}(b - a),$$

where:

$$\min(j) = \min_{i=1,\ldots,n} x_{ij},$$
$$\max(j) = \max_{i=1,\ldots,n} x_{ij},$$

a and *b* are the parameters of the algorithm.

Batch Processing

Algorithm Input

The min-max normalization algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Min-max (Batch Processing)

Input ID	Input
`data`	Pointer to the numeric table of size $nimesp$. **NOTE** This table can be an object of any class derived from `NumericTable`.

Algorithm Parameters

The min-max normalization algorithm has the following parameters:

Algorithm Parameters for Min-max (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`lowerBound`	**0.0**	The lower bound of the range to which the normalization scales values of the features.
`upperBound`	**1.0**	The upper bound of the range to which the normalization scales values of the features.
`moments`	**SharedPtr<low_order_moments::Batch<algorithmFPType, low_order_moments::defaultDense> >**	Pointer to the low order moments algorithm that computes minimums and maximums to be used for min-max normalization with the defaultDense method. For more details, see Batch Processing for Moments of Low Order.

Algorithm Output

The min-max normalization algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see `Algorithms`.

Algorithm Output for Min-max (Batch Processing)

Result ID	Result
normalizedData	Pointer to the $nimesp$ numeric table that stores the result of normalization. **NOTE** By default, the result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- minmax_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- MinMaxDenseBatch.java

Python*

Batch Processing:

- normalization_minmax_batch.py

Optimization Solvers

An optimization solver is an algorithm to solve an optimization problem, that is, to find the maximum or minimum of an objective function in the presence of constraints on its variables. In oneDAL the optimization solver represents the interface of algorithms that search for the argument θ_* that minimizes the function $K(\theta)$:

$$\theta_* = \text{argmin}_{\theta \in \Theta} K(\theta)$$

- Objective Function
  - Computation
  - Sum of Functions
  - Mean Squared Error Algorithm
  - Objective Function with Precomputed Characteristics Algorithm
  - Logistic Loss
  - Cross-entropy Loss
- Iterative Solver
  - Computation
  - Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm
  - Stochastic Gradient Descent Algorithm
  - Adaptive Subgradient Method
  - Coordinate Descent Algorithm
  - Stochastic Average Gradient Accelerated Method

Objective Function

In oneDAL, the objective function represents an interface of objective functions $K(\theta) = F(\theta) + M(\theta)$, where $F(\theta)$ is a smooth and $M(\theta)$ is a non-smooth functions, that accepts input argument $\theta \in R^p$ and returns:

- The value of objective function, $y = K(\theta)$
- The value of $M(\theta)$, $y_{ns} = M(\theta)$
- The gradient of $F(\theta)$:

$$g(\theta) = \nabla F(\theta) = \{\frac{\partial F}{\partial \theta_1}, \dots, \frac{\partial F}{\partial \theta_p}\}$$

- The Hessian of $F(\theta)$:

$$ERRORprocessingmath$$

- The objective function specific projection of proximal operator (see [MSE, Log-Loss, Cross-Entropy] for details):

$$\text{prox}_\eta^M(x) = \text{argmin}_{u \in R^p}(M(u) + \frac{1}{2\eta}|u - x|_2^2)$$

$$x \in R^p$$

- The objective function specific Lipschwitz constant, $\text{constantOfLipschitz} \leq |\nabla| F(\theta)$.

Objective functions

- Computation
- Sum of Functions
- Mean Squared Error Algorithm
- Objective Function with Precomputed Characteristics Algorithm
- Logistic Loss
- Cross-entropy Loss

NOTE On GPU, only Logistic Loss and Cross-entropy Loss are supported, Mean Squared Error Algorithm is not supported.

Computation

Input

The objective function accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Objective Function Computaion

Input ID	Input
`argument`	A numeric table of size $p \times 1$ with the input argument of the objective function.

Parameters

The objective function has the following parameters:

Parameters for Objective Function Computaion

<table>
<tr><th>Parameter</th><th>Default value</th><th>Description</th></tr>
<tr><td><code>resultsToCompute</code></td><td><code>gradient</code></td><td>The 64-bit integer flag that specifies which characteristics of the objective function to compute.
Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:
<table>
<tr><td>value</td><td>Value of the objective function</td></tr>
<tr><td>nonSmoothTermValue</td><td>Value of non-smooth term of the objective function</td></tr>
<tr><td>gradient</td><td>Gradient of the smooth term of the objective function</td></tr>
<tr><td>hessian</td><td>Hessian of smooth term of the objective function</td></tr>
<tr><td>proximalProjection</td><td>Projection of proximal operator for non-smooth term of the objective function</td></tr>
<tr><td>lipschitzConstant</td><td>Lipschitz constant of the smooth term of the objective function</td></tr>
<tr><td>gradientOverCertainFeature</td><td>Certain component of gradient vector</td></tr>
<tr><td>hessianOverCertainFeature</td><td>Certain component of hessian diagonal</td></tr>
<tr><td>proximalProjectionOfCertainFeature</td><td>Certain component of proximal projection</td></tr>
</table>
<b>NOTE</b> On GPU, <code>resultsToCompute</code> only computes <code>value</code>, <code>gradient</code>, and <code>hessian</code>.</td></tr>
</table>

Output

The objective function calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Objective Function Computaion

Result ID	Result
`valueIdx`	A numeric table of size $1imes1$ with the value of the objective function in the given argument.
`nonSmoothTermValueIdx`	A numeric table of size $1imes1$ with the value of the non-smooth term of the objective function in the given argument.
`gradientIdx`	A numeric table of size $p \times 1$ with the gradient of the smooth term of the objective function in the given argument.

`hessianIdx`	A numeric table of size $pimesp$ with the Hessian of the smooth term of the objective function in the given argument.
`proximalProjectionIdx`	A numeric table of size $p \times 1$ with the projection of proximal operator for non-smooth term of the objective function in the given argument.
`lipschitzConstantIdx`	A numeric table of size $1imes1$ with Lipschitz constant of the smooth term of the objective function.
`gradientOverCertainFeatureIdx`	A numeric table of size $1imes1$ with certain component of gradient vector.
`hessianOverCertainFeatureIdx`	A numeric table of size $1imes1$ with certain component of hessian diagonal.
`proximalProjectionOverCertainFeatureIdx`	A numeric table of size $1imes1$ with certain component of proximal projection.

NOTE

- If the function result is not requested through the resultsToCompute parameter, the respective element of the result contains a NULL pointer.
- By default, each numeric table specified by the collection elements is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable, except for PackedSymmetricMatrix, PackedTriangularMatrix, and CSRNumericTable.
- Hessian matrix is computed for the objective function $F(\theta) \in C^2$. For the objective functions $F(\theta) \in C^p$ with :math`p < 2` the library will stop computations and report the status on non-availability of the computation of the Hessian.
- If Lipschitz constant constantOfLipschitz is not estimated explicitly, pointer to result numeric table is required to be set to nullptr.

Sum of Functions

The sum of functions $F(\theta)$ is a function that has the form of a sum:

$$F(\theta) = \sum_{i=1} nF_i(\theta), \theta \in \mathbb{R}^p$$

For given set of the indices $I = \{i_1, i_2, \ldots, i_m\}$, $1 \leq ik < n$, $k \in \{1, \ldots, m\}$, the value and the gradient of the sum of functions in the argument θ has the format:

$$F_I(\theta) = \sum_{i \in I} F_i(\theta)$$

$$\nabla_I F_I(\theta) = \sum_{i \in I} \nabla F_i(\theta)$$

The set of the indices I is called a batch of indices.

Computation

Algorithm Input

The sum of functions algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Sum of Functions Computaion

Input ID	Input
`argument`	A numeric table of size $p \times 1$ with the input argument of the objective function.

Algorithm Parameters

The sum of functions algorithm has the following parameters:

Algorithm Parameters for Sum of Functions Computaion

<table>
<tr><th>Parameter</th><th>Default Value</th><th>Description</th></tr>
<tr><td><code>resultsToCompute</code></td><td><code>gradient</code></td><td>The 64-bit integer flag that specifies which characteristics of the objective function to compute.
Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:
<table>
<tr><td>value</td><td>Value of the objective function</td></tr>
<tr><td>nonSmoothTermValue</td><td>Value of non-smooth term of the objective function</td></tr>
<tr><td>gradient</td><td>Gradient of the smooth term of the objective function</td></tr>
<tr><td>hessian</td><td>Hessian of smooth term of the objective function</td></tr>
<tr><td>proximalProjection</td><td>Projection of proximal operator for non-smooth term of the objective function</td></tr>
<tr><td>lipschitzConstant</td><td>Lipschitz constant of the smooth term of the objective function</td></tr>
<tr><td>gradientOverCertainFeature</td><td>Certain component of gradient vector</td></tr>
<tr><td>hessianOverCertainFeature</td><td>Certain component of hessian diagonal</td></tr>
<tr><td>proximalProjectionOfCertainFeature</td><td>Certain component of proximal projection</td></tr>
</table>
</td></tr>
</table>

Algorithm Output

For the output of the sum of functions algorithm, see Output for objective functions.

Mean Squared Error Algorithm

NOTE Mean Squared Error Algorithm is not supported on GPU.

Details

Given $x = (x_{i1}, \ldots, x_{ip}) \in R^p$, a set of feature vectors $i \in \{1, \ldots, n\}$, and a set of respective responses y_i, the mean squared error (MSE) objective function $F(\theta; x, y)$ is a function that has the format:

$$F(\theta; x, y) = \sum_{i=1}^{n} F_i(\theta; x, y) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - h(\theta, x_i))^2$$

$$M(\theta) = 0$$

$$\text{prox}_{\gamma}^{M}(\theta_j) = \theta_j, j = 1, \ldots, p$$

In oneDAL implementation of the MSE, the $h(\theta, y_i)$ is represented as:

$$h(\theta, y_i) = \theta_0 + \sum_{j=1}^{p} \theta_j x_{ij}$$

For a given set of the indices $I = \{i_1, i_2, \ldots, i_m\}$, $1 \le i_r < n$, $l \in \{1, \ldots, m\}$, $|I| = m$, the value and the gradient of the sum of functions in the argument *x* respectively have the format:

$$F_I(\theta; x, y) = \frac{1}{2m} \sum_{i_k \in I} (y_{i_k} - h(\theta, x_{i_k}))^2$$

$$\nabla F_I(\theta; x, y) = \left\{ \frac{\partial F_I}{\partial \theta_0}, \ldots, \frac{\partial F_I}{\partial \theta_p} \right\}$$

where

$$\frac{\partial F_I}{\partial \theta_0} = \frac{1}{m} \sum_{i_k \in I} (y_{i_k} - h(\theta, x_{i_k}))$$

$$\frac{\partial F_I}{\partial \theta_j} = \frac{1}{m} \sum_{i_k \in I} (y_{i_k} - h(\theta, x_{i_k})) x_{i_k j}, j = 1, \ldots, p$$

$$lipschitzConstant = \max_{i=1,\ldots,n} \|x_i\|_2$$

Computation

Algorithm Input

The mean squared error algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for MSE Computaion

Input ID	Input
`argument`	A numeric table of size $(p + 1) \times 1$ with the input argument θ of the objective function.

`data`	A numeric table of size $nimesp$ with the data x_{ij}.
`dependentVariables`	A numeric table of size $nimes1$ with dependent variables y_i.

Optional Algorithm Input

The mean squared error algorithm accepts the optional input described below. Pass the Optional `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Optional Algorithm Input for MSE Computaion

Input ID	Input
`weights`	Optional input. Pointer to the $1imesn$ numeric table with weights of samples. The input can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix` and `PackedSymmetricMatrix`. By default, all weights are equal to **1**.
`gramMatrix`	Optional input. Pointer to the :mathL`p times p` numeric table with pre-computed Gram matrix. The input can be an object of any class derived from `NumericTable` except for `PackedTriangularMatrix` and `PackedSymmetricMatrix`. By default, the table is set to empty numeric table.

Algorithm Parameters

The mean squared error algorithm has the following parameters. Some of them are required only for specific values of the computation method parameter method:

Algorithm Parameters for MSE Computaion

Parameter	Default value	Description
`penaltyL1`	**0**	The numeric table of size $1 \times \mathrm{nDependentVariables}$ with L1 regularized coefficients.
`penaltyL2`	**0**	The numeric table of size $1 \times \mathrm{nDependentVariables}$ with L2 regularized coefficients.
`interceptFlag`	`true`	Flag to indicate whether or not to compute the intercept.
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method.
`numberOfTerms`	Not applicable	The number of terms in the objective function.
batchIndices	Not applicable	The numeric table of size $1 \times m$, where *m* is the batch size, with a batch of indices to be used to compute the function results. If no indices are provided, the implementation uses all the terms in the computation.

<table>
<tr><td></td><td></td><td><b>NOTE</b> This parameter can be an object of any class derived from <code>NumericTable</code> except for <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code>.</td></tr>
<tr><td><code>resultsToCompute</code></td><td><code>gradient</code></td><td>The 64-bit integer flag that specifies which characteristics of the objective function to compute.
Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:
<table>
<tr><td>value</td><td>Value of the objective function</td></tr>
<tr><td>nonSmoothTermValue</td><td>Value of non-smooth term of the objective function</td></tr>
<tr><td>gradient</td><td>Gradient of the smooth term of the objective function</td></tr>
<tr><td>hessian</td><td>Hessian of smooth term of the objective function</td></tr>
<tr><td>proximalProjection</td><td>Projection of proximal operator for non-smooth term of the objective function</td></tr>
<tr><td>lipschitzConstant</td><td>Lipschitz constant of the smooth term of the objective function</td></tr>
</table></td></tr>
</table>

Algorithm Output

For the output of the mean squared error algorithm, see Output for objective functions.

Examples

C++ (CPU)

- mse_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

- MSEDenseBatch.java

Objective Function with Precomputed Characteristics Algorithm

Objective function with precomputed characteristics gives an ability to provide the results of the objective function precomputed with the user-defined algorithm.

Set an earlier computed value and/or gradient and/or Hessian by allocating the result object and setting the characteristics of this result object. After that provide the modified result object to the algorithm for its further use with the iterative solver.

For more details on iterative solvers, refer to Iterative Solver.

Logistic Loss

Logistic loss is an objective function being minimized in the process of logistic regression training when a dependent variable takes only one of two values, **0** and **1**.

Details

Given *n* feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors, a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in \{0, 1\}$ describes the class to which the feature vector x_i belongs, the logistic loss objective function $K(\theta, X, y)$ has the following format $K(\theta, X, y) = F(\theta, X, y) + M(\theta)$, where

- $F(\theta, X, y)$ is defined as

$$F(\theta, X, y) = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \ln\left(\frac{1}{1+e^{-(\theta_0+\sum_{j=1}^{p}\theta_j x_{ij})}}\right) + (1-y_i)\ln\left(\frac{1}{1+e^{-(\theta_0+\sum_{j=1}^{p}\theta_j x_{ij})}}\right)\right) + \lambda_2 \sum_{j=1}^{p}$$

  with $\sigma(x, \theta) = \frac{1}{1+e^{-f(z,\theta)}}$, $f(z, \theta) = \theta_0 + \sum_{k=1}^{p}\theta_k z_k$, $\lambda_1 \geq 0$, $\lambda_2 \geq 0$
- $M(\theta) = \lambda_1 \sum_{j=1}^{p} |\theta_j|$

For a given set of the indices $I = \{i_1, i_2, \ldots, i_m\}$, $1 \leq i_r \leq n$, $r \in \{1, \ldots, m\}$:

- The value of the sum of functions has the format:

$$F_I(\theta, X, y) = -\frac{1}{m}\sum_{i \in I}(y_i \ln \sigma(x_i, \theta) + (1-y_i)\ln(1-\sigma(x_i,\theta))) + \lambda_2 \sum_{k=1}^{p}\theta_k^2$$

- The gradient of the sum of functions has the format:

$$\nabla F_I(\theta, x, y) = \left\{\frac{\partial F_I}{\partial \theta_0}, \ldots, \frac{\partial F_I}{\partial \theta_p}\right\},$$

  where

$$\frac{\partial F_I}{\partial \theta_0} = \frac{1}{m}\sum_{i \in I}(\sigma(x_i, \theta) - y_i) + 2\lambda_2\theta_0, \frac{\partial F_I}{\partial \theta_p} = \frac{1}{m}\sum_{i \in I}(\sigma(x_i, \theta) - y_i)x_{ij} + 2\lambda_2\theta_j, j = 1, \ldots, p$$

$$\text{prox}_\gamma^M(\theta_j) = \begin{cases} \theta_J - \lambda_1\gamma, & \theta_j > \lambda_1\gamma \\ 0, & |\theta_j| \leq \lambda_1\gamma \\ \theta_j + \lambda_1\gamma, & \theta_j < -\lambda_1\gamma \end{cases}$$

$$lipschitzConstant = \max_{i=1,\ldots,n} \|x_i\|_2 + \frac{\lambda_2}{n}$$

For more details, see [Hastie2009].

Computation

Algorithm Input

The logistic loss algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Logitic Loss Computaion

Input ID	Input

argument	A numeric table of size $(p+1) \times 1$ with the input argument θ of the objective function. **NOTE** The sizes of the argument, gradient, and hessian numeric tables do not depend on `interceptFlag`. When `interceptFlag` is set to `false`, the computation of θ_0 value is skipped, but the sizes of the tables should remain the same.
data	A numeric table of size $nimesp$ with the data x_ij. **NOTE** This parameter can be an object of any class derived from `NumericTable`.
dependentVariables	A numeric table of size $nimes1$ with dependent variables y_i. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix` , `PackedSymmetricMatrix` , and `CSRNumericTable`.

Algorithm Parameters

The logistic loss algorithm has the following parameters. Some of them are required only for specific values of the computation method's parameter `method`:

Algorithm Parameters for Logitic Loss Computaion

Parameter	Default value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
method	defaultDense	Performance-oriented computation method.
numberOfTerms	Not applicable	The number of terms in the objective function.
batchIndices	Not applicable	The numeric table of size $1 \times m$, where *m* is the batch size, with a batch of indices to be used to compute the function results. If no indices are provided, the implementation uses all the terms in the computation. **NOTE** This parameter can be an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `PackedSymmetricMatrix` .
resultsToCompute	gradient	The 64-bit integer flag that specifies which characteristics of the objective function to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: value — Value of the objective function

		nonSmoothTermValue	Value of non-smooth term of the objective function
		gradient	Gradient of the smooth term of the objective function
		hessian	Hessian of smooth term of the objective function
		proximalProjection	Projection of proximal operator for non-smooth term of the objective function
		lipschitzConstant	Lipschitz constant of the smooth term of the objective function
`interceptFlag`	`true`	A flag that indicates a need to compute θ_{0j}.	
`penaltyL1`	**0**	L1 regularization coefficient	
`penaltyL2`	**0**	L2 regularization coefficient	

Algorithm Output

For the output of the logistic loss algorithm, see Output for objective functions.

Examples

C++ (CPU)

- sgd_log_loss_dense_batch.cpp

Cross-entropy Loss

Cross-entropy loss is an objective function minimized in the process of logistic regression training when a dependent variable takes more than two values.

Details

Given *n* feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors, a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in \{0, T-1\}$ describes the class, to which the feature vector x_i belongs, where *T* is the number of classes, optimization solver optimizes cross-entropy loss objective function by argument θ, it is a matrix of size $T \times (p+1)$. The cross entropy loss objective function $K(\theta, X, y)$ has the following format $K(\theta, X, y) = F(\theta) + M(\theta)$ where

- $F(\theta) = -\frac{1}{n}\sum_{i=1}^{n} \log p_{y_i}(x_i, \theta) + \lambda_2 \sum_{t=0}^{T-1} \sum_{j=1}^{p} \theta_{tj}^2$, with $p_t(z, \theta) = \frac{e^{f_t(z,\theta)}}{\sum_{i=0}^{K-1} e^{f_i(z,\theta)}}$ and $f_t(z, \theta) = \theta_{t0} + \sum_{j=1}^{p} \theta_{tj} z_j$, $\lambda_1 \geq 0$, $\lambda_2 \geq 0$
- $M(\theta) = \lambda_1 \sum_{t=0}^{T-1} \sum_{j=1}^{p} |\theta_{tj}|$

For a given set of indices $I = \{i_1, i_2, \ldots, i_m\}$, $1 \leq i_r \leq n$, $r \in \{1, \ldots, m\}$, the value and the gradient of the sum of functions in the argument X respectively have the format:

$$F_I(\theta, X, y) = -\frac{1}{m} \sum_{i \in I} (\log p_{y_i}(x_i, \theta) + \lambda_2 \sum_{t=0}^{T-1} \sum_{j=1}^{p} \theta_{ij}^2)$$

$$\nabla F_I(\theta, x, y) = \left(\frac{\partial F_I}{\partial \theta_{00}}, \ldots, \frac{\partial F_I}{\partial \theta_{T-1p}} \right)^T$$

where

$$\frac{\partial F_I}{\partial \theta_{tj}} = \begin{cases} \frac{1}{m} \sum_{i \in I} g_t(\theta, x_i, y_i) + L_{tj}(\theta), & j = 0 \\ \frac{1}{m} \sum_{i \in I} g_t(\theta, x_i, y_i) x_{ij} + L_{tj}(\theta), & j = 0 \end{cases}$$

$$g_t(\theta, x, y) = \begin{cases} p_k(x, \theta) - 1, & y = t \\ p_t(x, \theta), & y \neq t \end{cases}$$

$$L_{tj}(\theta) = 2\lambda_2 \theta_{tj}$$

$$t \in [0, T-1]$$

$$j \in [0, p]$$

Hessian matrix is a symmetric matrix of size $S \times S$, where $S = T \times (p+1)$

$$\begin{bmatrix} \frac{\partial^2 F_I}{\partial \theta_{00} \partial \theta_{00}} & \cdots & \frac{\partial^2 F_I}{\partial \theta_{00} \partial \theta_{T-1p}} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 F_I}{\partial \theta_{T-1p} \partial \theta_{00}} & \cdots & \frac{\partial^2 F_I}{\partial \theta_{T-1p} \partial \theta_{T-1p}} \end{bmatrix}$$

$$\frac{\partial^2 F_I}{\partial \theta_{tj} \partial \theta_{pq}} = \begin{cases} \frac{1}{m} \sum_{i \in I} g_{tp}(\theta, x_i, y_i) + 2\lambda_2, & j = 0, q = 0 \\ \frac{1}{m} \sum_{i \in I} g_{tp}(\theta, x_i, y_i) x_{ij}, & j > 0, q = 0 \\ \frac{1}{m} \sum_{i \in I} g_{tp}(\theta, x_i, y_i) x_{iq}, & j = 0, q > 0 \\ \frac{1}{m} \sum_{i \in I} g_{tp}(\theta, x_i, y_i) x_{ij} x_{iq}, & j > 0, q > 0, j \neq q \\ \frac{1}{m} \sum_{i \in I} g_{tp}(\theta, x_i, y_i) x_{ij} x_{iq} + 2\lambda_2, & j > 0, q > 0, j = q \\ \end{cases}$$

$$g_{tp}(\theta, x, y) = \begin{cases} p_p(x, \theta)(1 - p_t(x, \theta)), & p = t \\ -p_t(x, \theta) p_p(x, \theta), & p \neq t \end{cases}$$

$$t, p \in [0, T-1]$$

$$j, q \in [0, p]$$

$$\text{prox}_\gamma^M(\theta_j) = \begin{cases} \theta_J - \lambda_1 \gamma, & \theta_j > \lambda_1 \gamma \\ 0, & |\theta_j| \leq \lambda_1 \gamma \\ \theta_j + \lambda_1 \gamma, & \theta_j < -\lambda_1 \gamma \end{cases}$$, where γ is the learning rate

$$lipschitzConstant = \max_{i=1,\ldots,n} \|x_i\|_2 + \frac{\lambda_2}{n}$$

For more details, see [Hastie2009].

Computation

Algorithm Input

The cross entropy loss algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Cross-entropy Loss Computaion

Input ID	Input
`argument`	A numeric table of size $(p+1) \times \mathrm{nClasses}$ with the input argument θ of the objective function. **NOTE** The sizes of the argument, gradient, and hessian numeric tables do not depend on `interceptFlag`. When `interceptFlag` is set to `false`, the computation of θ_0 value is skipped, but the sizes of the tables should remain the same.
`data`	A numeric table of size $nimesp$ with the data $x_i j$. **NOTE** This parameter can be an object of any class derived from `NumericTable`.
`dependentVariables`	A numeric table of size $nimes1$ with dependent variables y_i. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix` , `PackedSymmetricMatrix` , and `CSRNumericTable`.

Algorithm Parameters

The cross entropy loss algorithm has the following parameters. Some of them are required only for specific values of the computation method's parameter `method`:

Algorithm Parameters for Cross-entropy Loss Computaion

Parameter	Default value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method.
`numberOfTerms`	Not applicable	The number of terms in the objective function.

<table>
<tr><td><code>batchIndices</code></td><td>Not applicable</td><td>The numeric table of size $1 \times m$, where <i>m</i> is the batch size, with a batch of indices to be used to compute the function results. If no indices are provided, the implementation uses all the terms in the computation.

<b>NOTE</b> This parameter can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code> .</td></tr>
<tr><td><code>resultsToCompute</code></td><td><code>gradient</code></td><td>The 64-bit integer flag that specifies which characteristics of the objective function to compute.

Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:
<table>
<tr><td>value</td><td>Value of the objective function</td></tr>
<tr><td>nonSmoothTermValue</td><td>Value of non-smooth term of the objective function</td></tr>
<tr><td>gradient</td><td>Gradient of the smooth term of the objective function</td></tr>
<tr><td>hessian</td><td>Hessian of smooth term of the objective function</td></tr>
<tr><td>proximalProjection</td><td>Projection of proximal operator for non-smooth term of the objective function</td></tr>
<tr><td>lipschitzConstant</td><td>Lipschitz constant of the smooth term of the objective function</td></tr>
<tr><td>gradientOverCertainFeature</td><td>Certain component of gradient vector</td></tr>
<tr><td>hessianOverCertainFeature</td><td>Certain component of hessian diagonal</td></tr>
<tr><td>proximalProjectionOfCertainFeature</td><td>Certain component of proximal projection</td></tr>
</table></td></tr>
<tr><td><code>interceptFlag</code></td><td><code>true</code></td><td>A flag that indicates a need to compute θ_{0j}.</td></tr>
<tr><td><code>penaltyL1</code></td><td><b>0</b></td><td>L1 regularization coefficient</td></tr>
<tr><td><code>penaltyL2</code></td><td><b>0</b></td><td>L2 regularization coefficient</td></tr>
<tr><td><code>nClasses</code></td><td>Not applicable</td><td>The number of classes (different values of dependent variable)</td></tr>
</table>

Algorithm Output

For the output of the cross entropy loss algorithm, see Output for objective functions.

Examples

C++ (CPU)

- lbfgs_cr_entr_loss_dense_batch.cpp

Python*

- lbfgs_cr_entr_loss_batch.py

Iterative Solver

The iterative solver provides an iterative method to minimize an objective function that can be represented as a sum of functions in composite form

$$\theta_* = \mathrm{argmin}_{\theta \in R^p} K(\theta) = \mathrm{argmin}_{\theta \in R^p} F(\theta) + M(\theta)$$

where:

- $F(\theta) = \sum_{i=1}^{n} F_i(\theta)$, $\theta \in R^p$, where $F_i(\theta) : R^p \to R$ is a convex, continuously differentiable $F_i(\theta) \in C^{l \geq 1}$ (smooth) functions, $i = 1, \ldots, n$
- $M(\theta) : R^p \to R$ is a convex, non-differentiable (non-smooth) function

The Algorithmic Framework of an Iterative Solver

All solvers presented in the library follow a common algorithmic framework. Let S_t be a set of intrinsic parameters of the iterative solver for updating the argument of the objective function. This set is the algorithm-specific and can be empty. The solver determines the choice of S_0.

To do the computations, iterate *t* from **1** until $\mathrm{nIterations}$:

1. Choose a set of indices without replacement $I = \{i_1, \ldots, i_b\}$, $1 \leq i_j \leq n$, $j = 1, \ldots, b$, where *b* is the batch size.
2. Compute the gradient $g(\theta_{t-1}) = \nabla F_I(\theta_{t-1})$ where $F_I(\theta_{t-1}) = \sum_{i \in I} F_i(\theta_{t-1})$
3. Convergence check:

   Stop if $\frac{|U|_d}{\max(1, ||\theta_{t-1}||_d)} < \epsilon$ where *U* is an algorithm-specific vector (argument or gradient) and d is an algorithm-specific power of Lebesgue space
4. Compute θ_t using the algorithm-specific transformation *T* that updates the function’s argument:

   $$\theta_t = T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1})$$
5. Update $S_t : S_t = U(S_{t-1})$ where *U* is an algorithm-specific update of the set of intrinsic parameters.

The result of the solver is the argument $\theta_.$ and a set of parameters $S_.$ after the exit from the loop.

> **NOTE** You can resume the computations to get a more precise estimate of the objective function minimum. To do this, pass to the algorithm the results $\theta_.$ and $S_.$ of the previous run of the optimization solver. By default, the solver does not return the set of intrinsic parameters. If you need it, set the `optionalResultRequired` flag for the algorithm.

Iterative solvers

- Computation
- Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm
- Stochastic Gradient Descent Algorithm
- Adaptive Subgradient Method
- Coordinate Descent Algorithm
- Stochastic Average Gradient Accelerated Method

Computation

Algorithm Input

The iterative solver algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Algorithm Input for Iterative Solver Computaion

Input ID	Input
`inputArgument`	A numeric table of size $p \times 1$ with the value of start argument θ_0.
`optionalArgument`	Object of the `OptionalArgument` class that contains a set of algorithm-specific intrinsic parameters. For a detailed definition of the set, see the problem statement above and the description of a specific algorithm.

Algorithm Parameters

The iterative solver algorithm has the following parameters:

Algorithm Parameters for Iterative Solver Computaion

Parameter	Default Value	Description
`function`	Not applicable	Objective function represented as a sum of functions.
`nIterations`	**100**	Maximum number of iterations of the algorithm.
`accuracyThreshold`	$1.0 - e5$	Accuracy of the algorithm. The algorithm terminates when this accuracy is achieved.
`optionalResultRequired`	`false`	Indicates whether the set of the intrinsic parameters should be returned by the solver.

Algorithm Output

The iterative solver algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Algorithm Output for Iterative Solver Computaion

Result ID	Result
`minimum`	A numeric table of size $p \times 1$ with argument θ_*. By default, the result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable, except for PackedTriangularMatrix and PackedSymmetricMatrix.

Result ID	Result
nIterations	A numeric table of size $1imes1$ with a 32-bit integer number of iterations done by the algorithm. By default, the result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable, except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
optionalResult	Object of the OptionalArgument class that contains a set of algorithm-specific intrinsic parameters. For a detailed definition of the set, see the problem statement above and the description of a specific algorithm.

Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm

The limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm [Byrd2015] follows the algorithmic framework of an iterative solver with the algorithm-specific transformation *T* and set of intrinsic parameters S_t defined for the memory parameter *m*, frequency of curvature estimates calculation *L*, and step-length sequence $\alpha_t > 0$, algorithm-specific vector *U* and power *d* of Lebesgue space defined as follows:

Transformation

$$T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1})$$

$$\theta_t = \begin{cases} \theta_{t-1} - \alpha^t g(\theta_{t-1}), & t \leq 2 \\ \theta_{t-1} - \alpha^t H g(\theta_{t-1}), & \text{otherwise} \end{cases}$$

where *H* is an approximation of the inverse Hessian matrix computed from m correction pairs by the Hessian Update Algorithm.

Convergence check: $U = g\left(\theta_{t-1}\right),\ d = 2$

Intrinsic Parameters

For the LBFGS algorithm, the set of intrinsic parameters S_t includes the following:

- Correction pairs (s_j, y_j)
- Correction index k in the buffer that stores correction pairs
- Index of last iteration t of the main loop from the previous run
- Average value of arguments for the previous L iterations $\overline{\theta_{k-1}}$
- Average value of arguments for the last L iterations $\overline{\theta_k}$

Below is the definition and update flow of the intrinsic parameters (s_j, y_j). The index is set and remains zero for the first 2L-1 iterations of the main loop. Starting with iteration $2L$, the algorithm executes the following steps for each of L iterations of the main loop:

1. $k := k + 1$
2. Choose a set of indices without replacement: $I_H = \{i_1, i_2, \ldots, i_{b_H}\}$, $1 \leq i_l < n$, $l \in \{1, \ldots, b_H\}$, $|I_H| = b_H = \mathrm{correctionPairBatchSize}$.
3. Compute the sub-sampled Hessian

$$\nabla^2 F\left(\overline{\theta_k}\right) = \frac{1}{b_H} \sum_{i \in I_H} \nabla^2 F_i\left(\overline{\theta_k}\right)$$

at the point $\overline{\theta_k} = \frac{1}{L} \sum_{i=Lk}^{L(k+1)} \theta_i$ for the objective function using Hessians of its terms

$$ERRORprocessingmath$$

4. Compute the correction pairs (s_k, y_k):

   $s_k = \overline{\theta_k} - \overline{\theta_{k-1}}$

   $y_k = \nabla^2 F\left(\overline{\theta_k}\right) s_k$

NOTE

- The set S_k of intrinsic parameters is updated once per *L* iterations of the major loop and remains unchanged between iterations with the numbers that are multiples of *L*
- A cyclic buffer stores correction pairs. The algorithm fills the buffer with pairs one-by-one. Once the buffer is full, it returns to the beginning and overwrites the previous correction pairs.

Hessian Update Algorithm

This algorithm computes the approximation of the inverse Hessian matrix from the set of correction pairs [Byrd2015].

For a given set of correction pairs (s_j, y_j), $j = k - min(k, m) + 1, \ldots, k$:

1. Set $H = s_k^T y_k / y_k^T y_k$
2. Iterate *j* from $k - min(k, m) + 1$ until *k*:
   a. $\rho_j = 1 / y_j^T y_j$
   b. $H := \left(I - \rho_j s_j y_j^T\right) H \left(I - \rho_j y_j s_j^T\right) + \rho_j s_j s_j^T.$
3. Return *H*

Computation

The limited-memory BFGS algorithm is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see Computation.

Algorithm Input

In addition to the input of the iterative solver, the limited-memory BFGS algorithm accepts the following optional input:

Algorithm Input for Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Computaion

OptionalDataID	Input
`correctionPairs`	A numeric table of size $2m \times p$ where the rows represent correction pairs *s* and *y*. The row correctionPairs[j], $0 \leq j < m$, is a correction vector s_j, and the row correctionPairs[j], $m \leq j < 2m$, is a correction vector y_j.
`correctionIndices`	A numeric table of size 1×2 with 32-bit integer indexes. The first value is the index of correction pair *t*, the second value is the index of last iteration *k* from the previous run.
`averageArgumentLIterations`	A numeric table of size $2 \times p$, where row 0 represents average arguments for previous *L* iterations, and row 1 represents average arguments for last *L* iterations. These values are required to compute *s* correction vectors in the next step.

Algorithm Parameters

In addition to parameters of the iterative solver, the limited-memory BFGS algorithm has the following parameters:

Algorithm Parameters for Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Computaion

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method
`batchIndices`	`NULL`	The numeric table of size $nIterations \times batchSize$ with 32-bit integer indices of terms in the objective function to be used in step 2 of the limited-memory BFGS algorithm. If no indices are provided, the implementation generates random indices. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`batchSize`	**10**	The number of observations to compute the stochastic gradient. The implementation of the algorithm ignores this parameter if the batchIndices numeric table is provided. If BatchSize equals the number of terms in the objective function, no random sampling is performed and all terms are used to calculate the gradient.

Parameter	Default Value	Description
`correctionPairBatchSize`	**100**	The number of observations to compute the sub-sampled Hessian for correction pairs computation. The implementation of the algorithm ignores this parameter if the correctionPairIndices numeric table is provided. If `correctionPairBatchSize` equals the number of terms in the objective function, no random sampling is performed and all terms are used to calculate the Hessian matrix.
`correctionPairIndices`	`NULL`	The numeric table of size $(nIterations/L) \times correctionPairBatchSize$ with 32-bit integer indices to be used instead of random values. If no indices are provided, the implementation generates random indices. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`. **NOTE** If the algorithm runs with no optional input data, $(nIterations/L - 1)$ rows of the table are used. Otherwise, it can use one more row, $(nIterations/L)$ in total.
m	**10**	The memory parameter. The maximum number of correction pairs that define the approximation of the Hessian matrix.
L	**10**	The number of iterations between calculations of the curvature estimates.
`stepLengthSequence`	A numeric table of size $1imes1$ that contains the default step length equal to **1**.	The numeric table of size $1 \times nIterations$ or $1imes1$. The contents of the table depend on its size: • $size = 1 \times nIterations$: values of the step-length sequence α^k for $k = 1, \ldots, nIterations$. • $size = 1 \times 1$: the value of step length at each iteration $\alpha^1 = \ldots = \alpha^{nIterations}$

Parameter	Default Value	Description
		..note: This parameter can be an object of any class derived from ``NumericTable``, except for ``PackedTriangularMatrix``, ``PackedSymmetricMatrix``, and ``CSRNumericTable``. The recommended data type for storing the step-length sequence is the floating-point type, either float or double, that the algorithm uses in intermediate computations.
engine	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally for random choosing terms from the objective function.

Algorithm Output

In addition to the output of the iterative solver, the limited-memory BFGS algorithm calculates the following optional results:

Algorithm Output for Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Computaion

OptionalDataID	Output
correctionPairs	A numeric table of size $2m \times p$ where the rows represent correction pairs *s* and *y*. The row correctionPairs[j], $0 \leq j < m$, is a correction vector s_j, and the row correctionPairs[j], $m \leq j < 2m$, is a correction vector y_j.
correctionIndices	A numeric table of size 1×2 with 32-bit integer indexes. The first value is the index of correction pair *t*, the second value is the index of last iteration *k* from the previous run.
averageArgumentLIterations	A numeric table of size $2 \times p$, where row 0 represents average arguments for previous *L* iterations, and row 1 represents average arguments for last *L* iterations. These values are required to compute *s* correction vectors in the next step.

Examples

C++ (CPU)

Batch Processing:

- lbfgs_dense_batch.cpp
- lbfgs_opt_res_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- LBFGSDenseBatch.java
- LBFGSOptResDenseBatch.java

Python*

Batch Processing:

- lbfgs_cr_entr_loss_batch.py
- lbfgs_mse_batch.py

Stochastic Gradient Descent Algorithm

The stochastic gradient descent (SGD) algorithm is a special case of an iterative solver. See Iterative Solver for more details.

Computation methods

The following computation methods are available in oneDAL for the stochastic gradient descent algorithm:

- Mini-batch method
- Default method (a special case of mini-batch used by default)
- Momentum method

Mini-batch method

The mini-batch method (miniBatch) of the stochastic gradient descent algorithm [Mu2014] follows the algorithmic framework of an iterative solver with an empty set of intrinsic parameters of the algorithm S_t, algorithm-specific transformation *T* defined for the learning rate sequence $\{\eta_t\}_{t=1,\ldots,\text{nIterations}}$, conservative sequence $\{\gamma_t\}_{t=1,\ldots,\text{nIterations}}$ and the number of iterations in the internal loop *L*, algorithm-specific vector *U* and power *d* of Lebesgue space defined as follows:

$$T\left(\theta_{t-1}, g\left(\theta_{t-1}\right), S_{t-1}\right)$$

For *l* from **1** until *L*:

1. Update the function argument: $\theta_t := \theta_t - \eta_t\left(g\left(\theta_t\right) + \gamma_t\left(\theta_t - \theta_{t-1}\right)\right)$
2. Compute the gradient: $g\left(\theta_t\right) = \nabla F_I\left(\theta_t\right)$

Convergence check: $U = g\left(\theta_{t-1}\right),\ d = 2$

Default method

The default method (defaultDense) is a particular case of the mini-batch method with the batch size $b = 1$, $L = 1$, and conservative sequence $\gamma_t \equiv 0$.

Momentum method

The momentum method (momentum) of the stochastic gradient descent algorithm [Rumelhart86] follows the algorithmic framework of an iterative solver with the set of intrinsic parameters S_t, algorithm-specific transformation *T* defined for the learning rate sequence $\{\eta_t\}_{t=1,\ldots,\text{nIterations}}$ and momentum parameter $\mu in[0,1]$, and algorithm-specific vector *U* and power *d* of Lebesgue space defined as follows:

$$T\left(\theta_{t-1}, g\left(\theta_{t-1}\right), S_{t-1}\right)$$

1. $v_t = \mu \cdot v_{t-1} + \eta_t \cdot g\left(\theta_{t-1}\right)$
2. $\theta_t = \theta_{t-1} - v_t$

For the momentum method of the SGD algorithm, the set of intrinsic parameters S_t only contains the last update vector v_t.

Convergence check: $U = g\left(\theta_{t-1}\right),\ d = 2$

Computation

The stochastic gradient descent algorithm is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see Computation.

Algorithm Parameters

In addition to parameters of the iterative solver, the stochastic gradient descent algorithm has the following parameters. Some of them are required only for specific values of the computation method parameter method:

Algorithm Parameters for Stochastic Gradient Descent Algorithm Computaion

Parameter	method	Default Value	Description
`algorithmFPType`	`defaultDense`, `miniBatch`, `momentum`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	Not applicable	`defaultDense`	Available computation methods: For CPU: • `defaultDense` • `miniBatch` • `momentum` For GPU: • `miniBatch`
`batchIndices`	`defaultDense`, `miniBatch`, `momentum`	Not applicable	The numeric table with 32-bit integer indices of terms in the objective function. The method parameter determines the size of the numeric table: • `defaultDense`: `nIterations` x 1 • `miniBatch` and `momentum`: `nIterations` x `batchSize` If no indices are provided, the implementation generates random indices.
`batchSize`	`miniBatch`,`` momentum``	**128**	The number of batch indices to compute the stochastic gradient. If `batchSize` equals the number of terms in the objective function, no random sampling is performed, and all terms are used to calculate the gradient. The algorithm ignores this parameter if the batchIndices parameter is provided. For the `defaultDense` value of method, one term is used to compute the gradient on each iteration.

Parameter	method	Default Value	Description
`conservativeSequence`	`miniBatch`	A numeric table of size $1imes1$ that contains the default conservative coefficient equal to 1.	The numeric table of size $1 \times \mathrm{nIterations}$ or $1imes1$. The contents of the table depend on its size: • size = $1 \times \mathrm{nIterations}$: values of the conservative coefficient sequence γ^k for $k = 1, \dots, \mathrm{nIterations}$. • size = $1imes1$ the value of conservative coefficient at each iteration $\gamma^1 = \dots = \gamma^{\mathrm{nIterations}}$.
`innerNIterations`	`miniBatch`	**5**	The number of inner iterations for the miniBatch method.
`learningRateSequence`	`defaultDense, miniBatch, momentum`	A numeric table of size $1imes1$ that contains the default step length equal to 1.	The numeric table of size $1 \times \mathrm{nIterations}$ or $1imes1$. The contents of the table depend on its size: • size = $1 \times \mathrm{nIterations}$: values of the learning rate sequence η^k for $k = 1, \dots, \mathrm{nIterations}$. • size = $1imes1$: the value of learning rate at each iteration $\eta^1 = \dots = \eta^{\mathrm{nIterations}}$.
`momentum`	`momentum`	**0.9**	The momentum value.
`engine`	`defaultDense, miniBatch, momentum`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally for generation of 32-bit integer indices of terms in the objective function.

Examples

C++ (CPU)

Batch Processing:

- sgd_dense_batch.cpp
- sgd_mini_dense_batch.cpp
- sgd_moment_dense_batch.cpp
- sgd_moment_opt_res_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- SGDDenseBatch.java
- SGDMiniDenseBatch.java

- SGDMomentDenseBatch.java
- SGDMomentOptResDenseBatch.java

Python*

Batch Processing:

- sgd_logistic_loss_batch.py
- sgd_mse_batch.py

Adaptive Subgradient Method

The adaptive subgradient method (AdaGrad) [Duchi2011] follows the algorithmic framework of an iterative solver with the algorithm-specific transformation *T*, set of intrinsic parameters S_t defined for the learning rate η, and algorithm-specific vector *U* and power *d* of Lebesgue space defined as follows:

$$S_t = G_t$$

$$G_t = (G_{t,i})_{i=1,\ldots,p}$$

$$G_0 \equiv 0$$

$T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1})$:

1. $G_{t,i} = G_{t-1,i} + g_i^2(\theta_{t-1})$, where $g_i(\theta_{t-1})$ is the *i*-th coordinate of the gradient $g(\theta_{t-1})$
2. $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \varepsilon}} g(\theta_{t-1})$, where

$$\frac{\eta}{\sqrt{G_t + \varepsilon}} g(\theta_{t-1}) = \{\frac{\eta}{\sqrt{G_{t,1} + \varepsilon}} g_1(\theta_{t-1}), \ldots, \frac{\eta}{\sqrt{G_{t,1} + \varepsilon}} g_p(\theta_{t-1})\}$$

Convergence check: $U = g(\theta_{t-1}), d = 2$

Computation

The adaptive subgradient (AdaGrad) method is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see Computation for Iterative Solver.

Algorithm Input

In addition to the input of the iterative solver, the AdaGrad method accepts the following optional input:

Algorithm Input for Adaptive Subgradient Method Computaion

OptionalDataID	Input
`gradientSquareSum`	A numeric table of size $p \times 1$ with the values of G_t. Each value is an accumulated sum of squares of coordinate values of a corresponding gradient.

Algorithm Parameters

In addition to parameters of the iterative solver, the AdaGrad method has the following parameters:

Algorithm Parameters for Adaptive Subgradient Method Computaion

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.

Parameter	Default Value	Description
`method`	`defaultDense`	Default performance-oriented computation method.
`batchIndices`	`NULL`	A numeric table of size $\mathrm{nIterations} \times \mathrm{batchSize}$ for the `defaultDense` method that represents 32-bit integer indices of terms in the objective function. If no indices are provided, the algorithm generates random indices.
`batchSize`	**128**	The number of batch indices to compute the stochastic gradient. If `batchSize` equals the number of terms in the objective function, no random sampling is performed, and all terms are used to calculate the gradient. The algorithm ignores this parameter if the `batchIndices` parameter is provided.
`learningRate`	A numeric table of size $1imes1$ that contains the default step length equal to **0.01**.	A numeric table of size $1imes1$ that contains the value of learning rate η. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`degenerateCasesThreshold`	$1e-08$	Value ε needed to avoid degenerate cases when computing square roots.
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally for generation of 32-bit integer indices of terms in the objective function.

Algorithm Output

In addition to the output of the iterative solver, the AdaGrad method calculates the following optional result:

Algorithm Output for Adaptive Subgradient Method Computaion

OptionalDataID	Output
`gradientSquareSum`	A numeric table of size $p \times 1$ with the values of G_t. Each value is an accumulated sum of squares of coordinate values of a corresponding gradient.

Examples

C++ (CPU)

- adagrad_dense_batch.cpp
- adagrad_opt_res_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

- AdagradDenseBatch.java
- AdagradOptResDenseBatch.java

Python*

- adagrad_mse_batch.py

Coordinate Descent Algorithm

The Coordinate Descent algorithm follows the algorithmic framework of iterative solver with one exception: the default method (`defaultDense`) of Coordinate Descent algorithm is a case of the iterative solver method with the batch equal to the number of observations in the training data set.

Details

The aet of intrinsic parameters S_t is empty. Algorithmic-specific transformation *T*, algorithm-specific vector *U*, and power *d* of Lebesgue space[Adams2003] are defined as follows:

$$T(\theta_{t-1}, F'(\theta_{t-1}), S_{t-1}, M(\theta_{t-1}))$$

1. Define the index *j* to update the component of a coefficient as a remainder in the division of the number of current iteration (*t*) by the number of features in the training data set (*p*): $j = \mathrm{mod}(t, p)$

   Alternatively, if `selection` parameter was set to `random`, generate *j* randomly.
2. If `stepLengthSequence` was not provided by the user, compute the learning rate: $\eta = (F''(\theta_{t-1}))_{jj}$ (the diagonal element of the Hessian matrix)
3. Update the *j*-th component of vector θ:

$$(\theta_t)_j = \mathrm{prox}_{\frac{1}{\eta}}^{M}\left((\theta_{t-1})_j - \frac{1}{\max(\eta, \mathrm{eps})}(F'(\theta_{t-1}))_j\right)$$

Note: for example, if a non-smooth term $M = \lambda \sum_{i=1}^{p} |\theta_t|$, where *p* is the number of features in the training data set, the objective function should compute prox operator as follows:

$$\mathrm{prox}_{\frac{1}{\eta}}^{M}((\theta_{t-1})_j) = \begin{cases} (\theta_{t-1})_j - \lambda\frac{1}{\eta}, & (\theta_{t-1})_j > \lambda\frac{1}{\eta} \\ 0, & |(\theta_{t-1})_j| \leq \lambda\frac{1}{\eta} \\ (\theta_{t-1})_j + \lambda\frac{1}{\eta}, & (\theta_{t-1})_j < -\lambda\frac{1}{\eta} \end{cases}$$

Convergence check is performed each *p* iterations:

- $U = \theta_t - \theta_{t-\mathrm{nFeatures}}, d = \infty$
- For $x \in R^p$, the infinity norm ($d = \infty$) is defined as follows:

$$|x|_\infty = \max_{i \in [0,p]}(|x_i|)$$

Computation

Coordinate Descent algorithm is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see Iterative Solver > Computation.

Algorithm Parameters

In addition to the input of a iterative solver, Coordinate Descent algorithm accepts the following parameters:

Algorithm Parameters for Coordinate Descent Computaion

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented method.
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally during each iteration to choose a random component of the minimum result vector to be updated.
`positive`	`false`	A boolean value. When set to `true`, it forces the coefficients to be positive.
`selection`	`cyclic`	Value that specifies the strategy of certain coordinate selection on each iteration. Except for default `cyclic` value, Coordinate Descent also supports: • `random` – on each iteration the index of coordinate is selected randomly by the engine.
`skipTheFirstComponents`	`false`	A boolean value. When set to `true`, Coordinate Descent algorithm will skip the first component from optimization.

Examples

C++ (CPU)

- cd_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

- CDDenseBatch.java

Stochastic Average Gradient Accelerated Method

The Stochastic Average Gradient Accelerated (SAGA) [Defazio2014] follows the algorithmic framework of an iterative solver with one exception.

The default method (`defaultDense`) of SAGA algorithm is a particular case of the iterative solver method with the batch size $b = 1$.

Details

Algorithmic-specific transformation *T*, the set of intrinsic parameters S_t defined for the learning rate η, and algorithm-specific vector *U* and power *d* of Lebesgue space are defined as follows:

$$S_t = \{G^t\}$$
$$G^t = (G_i^t)_{i=1,\ldots,n}$$
$$G^0 \equiv (G_i^0)_{i=1,\ldots,n} \equiv F_i'(\theta_0)_{i=1,\ldots,n}$$

S_t is a matrix of the gradients of smooth terms at point θ_t, where

- *t* is defined by the number of iterations the solver runs
- G_i^t stores the gradient of $f_i(\theta_t)$

$T(\theta_{t-1}, F_j'(\theta_{t-1}), S_{t-1}, M(\theta_{t-1}))$:

1. $W_t = \theta_{t-1} - \eta_j \left[F_j'(\theta_{t-1}) - G_j^{t-1} + \frac{1}{n}\sum_{i=1}^{n} G_i^{t-1}\right]$
2. $\theta_t = \text{prox}_\eta^M(W_t)$

Update of the set of intrinsic parameters S_t:

$$G_j^{t-1} = F_j'(\theta_{t-1})$$

> **NOTE** The algorithm enables automatic step-length selection if learning rate η was not provided by the user. Automatic step-length will be computed as $\eta = \frac{1}{L}$, where *L* is the Lipschitz constant returned by objective function. If the objective function returns `nullptr` to numeric table with `lipschitzConstant` Result ID, the library will use default step size **0.01**.

Convergence checks:

- $U = \theta_t - \theta_{t-1}, d = \infty$
- $|x|_\infty = \max_{i \in [0,p]}(|x^i|)$, $x \in R^p$

Computation

The stochastic average gradient (SAGA) algorithm is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see Iterative Solver > Computation.

Algorithm Input

In addition to the input of the iterative solver, the SAGA optimization solver has the following optional input:

Algorithm Input for Stochastic Average Gradient Accelerated Method Computaion

OptionalDataID	Default Value	Description
`gradientTable`	Not applicable	A numeric table of size $nimesp$ which represents G_0 matrix that contains gradients of $F_i(\theta)$, $1, \ldots, n$ at the initial point $\theta_0 \in R^p$. This input is optional: if the user does not provide the table of gradients for $F_i(\theta)$, $1, \ldots, n$, the library will compute it inside the SAGA algorithm.

		NOTE This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Algorithm Parameters

In addition to parameters of the iterative solver, the SAGA optimization solver has the following parameters:

Algorithm Parameters for Stochastic Average Gradient Accelerated Method Computaion

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented method.
`batchIndices`	**1**	A numeric table of size $\mathrm{nIterations} \times 1$ with 32-bit integer indices of terms in the objective function. If no indices are provided, the implementation generates random index on each iteration. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`learningRateSequence`	Not applicable	The numeric table of size $1 \times \mathrm{nIterations}$ or $1imes1$ that contains learning rate for each iterations is first case, otherwise constant step length will be used for all iterations. It is recommended to set diminishing learning rate sequence. If `learningRateSequence` is not provided, the learning rate will be computed automatically via `constantOfLipschitz` Result ID. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`engine`	**SharedPtr<engines::mt19937::Batch<>>**	Pointer to the random number generator engine that is used internally for generation of 32-bit integer index of term in the objective function.

Algorithm Output

In addition to the output of the iterative solver, the SAGA optimization solver calculates the following optional result:

Algorithm Output for Stochastic Average Gradient Accelerated Method Computaion

OptionalDataID	Default Value	Description
`gradientTable`	Not applicable	A numeric table of size $nimesp$ that represents matrix G_t updated after all iterations. This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Examples

C++ (CPU)

Batch Processing:

- saga_dense_batch.cpp
- saga_logistic_loss_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- SAGADenseBatch.java
- SAGALogisticLossDenseBatch.java

Python*

Batch Processing:

- saga_batch.py

Training and Prediction

Training and prediction algorithms in Intel® oneAPI Data Analytics Library (oneDAL) include a range of popular machine learning algorithms:

- Decision Forest
- Decision Trees
- Gradient Boosted Trees
- Stump
- Linear and Ridge Regressions
- LASSO and Elastic Net Regressions
- k-Nearest Neighbors (kNN) Classifier
- Implicit Alternating Least Squares
- Logistic Regression
- Naïve Bayes Classifier
- Support Vector Machine Classifier
- Multi-class Classifier
- Boosting

Unlike Analysis algorithms, which are intended to characterize the structure of data sets, machine learning algorithms model the data. Modeling operates in two major stages:

- **Training**, when the algorithm estimates model parameters based on a training data set.
- **Prediction or decision making**, when the algorithm uses the trained model to predict the outcome based on new data.

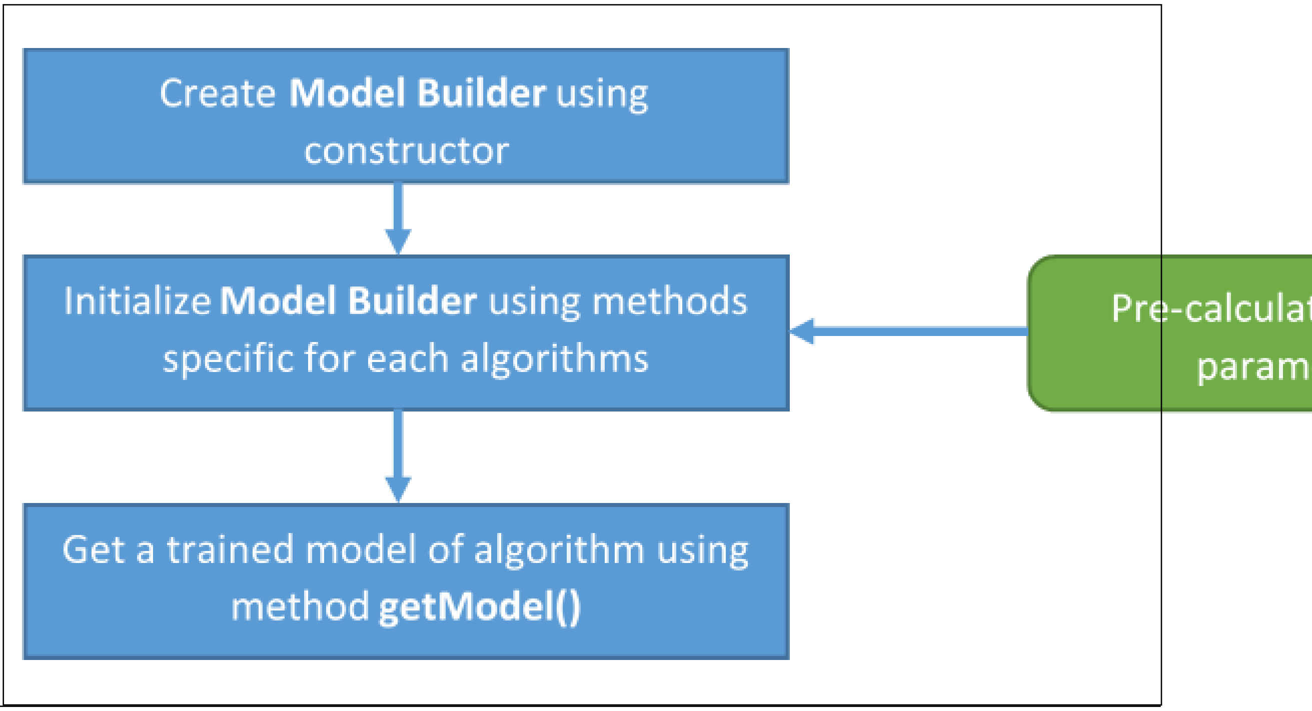
Training is typically a lot more computationally complex problem than prediction. Therefore, certain end-to-end analytics usage scenarios require that training and prediction phases are done on different devices, the training is done on more powerful devices, while prediction is done on smaller devices. Because smaller devices may have stricter memory footprint requirements, oneDAL separates Training, Prediction, and respective Model in three different class hierarchies to minimize the footprint.

Training Alternative

An alternative to training your model with algorithms implemented in oneDAL is to build a trained model from pre-calculated model parameters, for example, coefficients β for Linear Regression. This enables you to use oneDAL only to get predictions based on the model parameters computed elsewhere.

The Model Builder class provides an interface for adding all the necessary parameters and building a trained model ready for the prediction stage.

The following schema illustrates the use of Model Builder class:


Create **Model Builder** using constructor
Initialize **Model Builder** using methods specific for each algorithms
Get a trained model of algorithm using method **getModel()**
Pre-calculat
param

The Model Builder class is implemented for the following algorithms:

- Linear Regression
- Support Vector Machine Classifier
- Multi-class Classifier
- Logistic Regression
- Regression Gradient Boosted Trees
- Classification Gradient Boosted Trees
- Classification Decision Forest

Decision Forest

NOTE Decision Forest is also available with oneAPI interfaces:

- Decision Forest Classification and Regression (DF)

The library provides decision forest classification and regression algorithms based on an ensemble of tree-structured classifiers, which are known as decision trees. Decision forest is built using the general technique of bagging, a **b**ootstrap **agg**regation, and a random choice of features.

Decision Tree is a binary tree graph. Its internal (split) nodes represent a *decision function* used to select the child node at the prediction stage. Its leaf, or terminal, nodes represent the corresponding response values, which are the result of the prediction from the tree. For more details, see [Breiman84] and [Breiman2001].

- Decision Forest
- Regression Decision Forest
- Classification Decision Forest

Decision Forest

Details

Given n feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors and n responses $y = (y_1, \ldots, y_n)$, the problem is to build a decision forest classification or regression model.

Training Stage

Library uses the following algorithmic framework for the training stage. Let $S = (X, Y)$ be the set of observations. Given a positive integer parameters, such as the number of trees *B*, the bootstrap parameter $N = f * n$, where *f* is a fraction of observations used for a training of one tree, and the number of features per node *m*, the algorithm does the following for $b = 1, \ldots, B$:

- Selects randomly with replacement the set D_b of *N* vectors from the set *S*. The set D_b is called a *bootstrap* set.
- Trains a decision tree classifier T_b on D_b using parameter *m* for each tree.

Decision tree *T* is trained using the training set *D* of size *N*. Each node *t* in the tree corresponds to the subset D_t of the training set *D*, with the root node being *D* itself. Its internal nodes *t* represent a binary test (split) dividing their subset X_t in two subsets X_{t_L} and X_{t_R}, corresponding to their children t_L and t_R.

Inexact Histogram Computation Method

In inexact histogram method only a selected subset of splits is considered for computation of a best split. This subset is computed for each feature at the initialization stage of the algorithm. After the set of splits is computed, each value from initially provided data is substituted with the value of the corresponding bin. The bins are continuous intervals between the selected splits.

Split Criteria

The metric for measuring the best split is called *impurity*, *i(t)*. It generally reflects the homogeneity of responses within the subset D_t in the node *t*. For the detailed definition of *i(t)* metrics, see the description of a specific algorithm.

Let the *impurity decrease* in the node *t* be

$$\Delta i\left(t\right) = i\left(t\right) - \frac{|D_{t_L}|}{|D_t|} i\left(t_L\right) - \frac{|D_{t_R}|}{|D_t|} i\left(t_R\right).$$

Termination Criteria

The library supports the following termination criteria of decision forest training:

Minimal number of observations in a leaf node	Node *t* is not processed if $\|D_t\|$ is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.
Minimal number of observations in a split node	Node *t* is not processed if $\|D_t\|$ is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.
Minimum weighted fraction of the sum total of weights of all the input observations required to be at a leaf node	Node *t* is not processed if $\|D_t\|$ is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.
Maximal tree depth	Node *t* is not processed if its depth in the tree reached the predefined value.
Impurity threshold	Node *t* is not processed if its impurity is smaller than the predefined threshold.
Maximal number of leaf nodes	Grow trees with positive maximal number of leaf nodes in a best-first fashion. Best nodes are defined by relative reduction in impurity. If maximal number of leaf nodes equals zero, then this criterion does not limit the number of leaf nodes, and trees grow in a depth-first fashion.

Tree Building Strategies

Maximal number of leaf nodes defines the strategy of tree building: depth-first or best-first.

Depth-first Strategy

If maximal number of leaf nodes equals zero, a decision tree is built using depth-first strategy. In each terminal node *t*, the following recursive procedure is applied:

- Stop if the termination criteria are met.
- Choose randomly without replacement *m* feature indices $J_t \in \{0, 1, \ldots, p-1\}$.
- For each $j \in J_t$, find the best split $s_{j,t}$ that partitions subset D_t and maximizes impurity decrease $\Delta i(t)$.
- A node is a split if this split induces a decrease of the impurity greater than or equal to the predefined value. Get the best split s_t that maximizes impurity decrease Δi in all $s_{j,t}$ splits.
- Apply this procedure recursively to t_L and t_R.

Best-first Strategy

If maximal number of leaf nodes is positive, a decision tree is built using best-first strategy. In each terminal node *t*, the following steps are applied:

- Stop if the termination criteria are met.
- Choose randomly without replacement *m* feature indices $J_t \in \{0, 1, \ldots, p-1\}$.
- For each $j \in J_t$, find the best split $s_{j,t}$ that partitions subset D_t and maximizes impurity decrease $\Delta i(t)$.

- A node is a split if this split induces a decrease of the impurity greater than or equal to the predefined value and the number of split nodes is less or equal to $\mathrm{maxLeafNodes}-1$. Get the best split s_t that maximizes impurity decrease Δi in all $s_{j,t}$ splits.
- Put a node into a sorted array, where sort criterion is the improvement in impurity $\Delta i(t)|D_t|$. The node with maximal improvement is the first in the array. For a leaf node, the improvement in impurity is zero.
- Apply this procedure to t_L and t_R and grow a tree one by one getting the first element from the array until the array is empty.

Random Numbers Generation

To create a *bootstrap* set and choose feature indices in the performant way, the training algorithm requires the source of random numbers, capable to produce sequences of random numbers in parallel.

Initialization of the engine in the decision forest is based on the scheme below:

The state of the engine is updated once the training of the decision forest model is completed. The library provides support to retrieve the instance of the engine with updated state that can be used in other computations. The update of the state is engine-specific and depends on the parallelization technique used as defined earlier:

- Family: the updated state is the set of states that represent individual engines in the family.
- Leapfrog: the updated state is the state of the sequence with the rightmost position on the sequence. The example below demonstrates the idea for case of 2 subsequences (‘x’ and ‘o’) of the random number sequence:
- SkipAhead: the updated state is the state of the independent sequence with the rightmost position on the sequence. The example below demonstrates the idea for case of 2 subsequences (‘x’ and ‘o’) of the random number sequence:

Prediction Stage

Given decision forest classifier and vectors $x1, \ldots, x_r$, the problem is to calculate the responses for those vectors. To solve the problem for each given query vector x_i , the algorithm finds the leaf node in a tree in the forest that gives the response by that tree. The response of the forest is based on an aggregation of responses from all trees in the forest. For the detailed definition, see the description of a specific algorithm.

Additional Characteristics Calculated by the Decision Forest

Decision forests can produce additional characteristics, such as an estimate of generalization error and an importance measure (relative decisive power) of each of p features (variables).

Out-of-bag Error

The estimate of the generalization error based on the training data can be obtained and calculated as follows:

- For each tree T_b in the forest, trained on the bootstrap set D_b, the set $\overline{D_b} = S \setminus D_b$ is called the out-of-bag (OOB) set.
- Predict the data from $\overline{D_b}$ set by T_b.
- For each vector x_i in the dataset X, predict its response $\hat{y_i}$ by the trees that contain x_i in their OOB set.
- Aggregate the out-of-bag predictions in all trees and calculate the OOB error of the decision forest.
- If OOB error value per each observation is required, then calculate the prediction error for x_i.

For the detailed definition, see the description of a specific algorithm.

Variable Importance

There are two main types of variable importance measures:

- *Mean Decrease Impurity* importance (MDI).

  Importance of the *j*-th variable for predicting *Y* is the sum of weighted impurity decreases $p(t)\Delta i(s_t, t)$ for all nodes *t* that use x_j, averaged over all *B* trees in the forest:

  $$MDI\left(j\right) = \frac{1}{B}\sum_{b=1}^{B}\sum_{t\in T_b : v\left(s_t\right)=j} p\left(t\right)\Delta i\left(s_t, t\right),$$

  where $p\left(t\right) = \frac{|D_t|}{|D|}$ is the fraction of observations reaching node *t* in the tree T_b, and $v(s_t)$ is the index of the variable used in split s_t .
- *Mean Decrease Accuracy* (MDA).

  Importance of the *j*-th variable for predicting *Y* is the average increase in the OOB error over all trees in the forest when the values of the *j*-th variable are randomly permuted in the OOB set. For that reason, this latter measure is also known as *permutation importance*.

  In more details, the library calculates MDA importance as follows:

  - Let $\pi(X, j)$ be the set of feature vectors where the *j*-th variable is randomly permuted over all vectors in the set.
  - Let E_b be the OOB error calculated for $T_b :$ on its out-of-bag dataset $\overline{D_b}$.
  - Let $E_{b,j}$ be the OOB error calculated for $T_b :$ using $\pi\left(\overline{X_b}, j\right)$, and its out-of-bag dataset $\overline{D_b}$ is permuted on the *j*-th variable. Then
    - $\delta_{b,j} = E_b - E_{b,j}$ is the OOB error increase for the tree T_b.
    - $RawMDA\left(j\right) = \frac{1}{B}\sum_{b=1}^{B}\delta_{b,j}$ is MDA importance.
    - $ScaledMDA\left(j\right) = \frac{Raw\ MDA(x_j)}{\frac{\sigma_j}{\sqrt{B}}}$, where σ_j^2 is the variance of $D_{b,j}$

Batch Processing

Decision forest classification and regression follows the general workflow described in Classification Usage Model.

Training

At the training stage, decision forest regression has the following parameters:

Training Parameters for Decision Forest (Batch Processing)

Parameter	Default Value	Description
`nTrees`	**100**	The number of trees in the forest.
`observationsPerTreeFraction`	**1**	Fraction of the training set S used to form the bootstrap set for a single tree training, $0 < \text{observationsPerTreeFraction} \leq 1$. The observations are sampled randomly with replacement.

Parameter	Default Value	Description
featuresPerNode	**0**	The number of features tried as possible splits per node. If the parameter is set to **0**, the library uses the square root of the number of features, $\sqrt{p}$, for classification and $\frac{p}{3}$ features for regression.
maxTreeDepth	**0**	Maximal tree depth. Default is **0** (unlimited).
DEPRECATED: seed	**777**	The seed for random number generator, which is used to choose the bootstrap set, split features in every split node in a tree, and generate permutation required in computations of MDA variable importance. **NOTE** This parameter is deprecated and will be removed in future releases. Use engine instead.
engine	**SharePtr< engines:: mt2203:: Batch>()**	Pointer to the random number generator engine. The random numbers produced by this engine are used to choose the bootstrap set, split features in every split node in a tree, and generate permutation required in computations of MDA variable importance.
impurityThreshold	**0**	The threshold value used as stopping criteria: if the impurity value in the node is smaller than the threshold, the node is not split anymore.
varImportance	none	The variable importance computation mode. Possible values: • none – variable importance is not calculated • MDI - Mean Decrease of Impurity, also known as the Gini importance or Mean Decrease Gini • MDA_Raw - Mean Decrease of Accuracy (permutation importance) • MDA_Scaled - the MDA_Raw value scaled by its standard deviation
resultsToCompute	**0**	The 64-bit integer flag that specifies which extra characteristics of the decision forest to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: • computeOutOfBagError • computeOutOfBagErrorPerObservation
bootstrap	true	If true, the training set for a tree is a bootstrap of the whole training set. If false, the whole training set is used to build trees.

Parameter	Default Value	Description
`minObservationsInLeafNode`	**1** for classification, **5** for regression	Minimum number of observations in the leaf node.
`minObservationsInSplitNode`	**2**	Minimum number of samples required to split an internal node; it can be any non-negative number.
`minWeightFractionInLeafNode`	**0.0**	Minimum weighted fraction of the sum total of weights of all the input observations required to be at a leaf node, from **0.0** to **0.5**. All observations have equal weights if the weights of the observations are not provided.
`minImpurityDecreaseInSplitNode`	**0.0**	Minimum amount of impurity decrease required to split a node; it can be any non-negative number.
`maxLeafNodes`	**0**	Grow trees with positive maximal number of leaf nodes in a best-first fashion. Best nodes are defined as relative reduction in impurity. If maximal number of leaf nodes equals zero, then this parameter does not limit the number of leaf nodes, and trees grow in a depth-first fashion.

Output

In addition to regression or classifier output, decision forest calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the result of your algorithm.

Training Output for Decision Forest (Batch Processing)

Result ID	Result
`outOfBagError`	A numeric table $1imes1$ containing out-of-bag error computed when the `computeOutOfBagErroroption` option is on. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`.
`variableImportance`	A numeric table $1imesp$ that contains variable importance values for each feature. If you set the `varImportance` parameter to none, the library returns a null pointer to the table. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `PackedSymmetricMatrix`.

Result ID	Result
outOfBagErrorPerObservation	A numeric table of size $1imesn$ that contains the computed out-of-bag error when the computeOutOfBagErrorPerObservation option is enabled. The value **-1** in the table indicates that no OOB value was computed because this observation was not in OOB set for any of the trees in the model (never left out during the bootstrap). **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable.
updatedEngine	Engine instance with state updated after computations.

Performance Considerations

To get the best performance of the decision forest variable importance computation, use the Mean Decrease Impurity (MDI) rather than the Mean Decrease Accuracy (MDA) method.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Regression Decision Forest

Decision forest regression is a special case of the Decision Forest model.

Details

Given:

- *n* feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of size *p*;
- their non-negative sample weights $w = (w_1, \ldots, w_n)$;
- the vector of responses $y = (y_1, \ldots, y_n)$

The problem is to build a decision forest regression model that minimizes the Mean-Square Error (MSE) between the predicted and true value.

Training Stage

Decision forest regression follows the algorithmic framework of decision forest training algorithm based on the mean-squared error (MSE) [Breiman84]. If sample weights are provided as input, the library uses a weighted version of the algorithm.

MSE is an impurity metric (*D* is a set of observations that reach the node), calculated as follows:

Decision Forest Regression: impurity calculations

Without sample weights	With sample weights
$I_{\text{MSE}}(D) = \frac{1}{W(D)} \sum_{i=1}^{W(D)} \left(y_i - \frac{1}{W(D)} \sum_{j=1}^{W(D)} y_j\right)^2$	$I_{\text{MSE}}(D) = \frac{1}{W(D)} \sum_{i \in D} w_i \left(y_i - \frac{1}{W(D)} \sum_{j \in D} w_j y_j\right)^2$
$W(S) = \sum_{s \in S} 1$, which is equivalent to the number of elements in *S*	$W(S) = \sum_{s \in S} w_s$

Prediction Stage

Given decision forest regression model and vectors $x_1, \ldots, x_r$, the problem is to calculate the responses for those vectors. To solve the problem for each given query vector x_i, the algorithm finds the leaf node in a tree in the forest that gives the response by that tree as the mean of dependent variables. The forest predicts the response as the mean of responses from trees.

Out-of-bag Error

Decision forest regression follows the algorithmic framework for calculating the decision forest out-of-bag (OOB) error, where aggregation of the out-of-bag predictions in all trees and calculation of the OOB error of the decision forest is done as follows:

- For each vector x_i in the dataset *X*, predict its response $\hat{y_i}$ as the mean of prediction from the trees that contain x_i in their OOB set:

  $\hat{y_i} = \frac{1}{|B_i|} \sum_{b=1}^{|B_i|} \hat{y_{ib}}$, where $B_i = \bigcup T_b : x_i \in \overline{D_b}$ and $\hat{y_{ib}}$ is the result of prediction x_i by T_b.
- Calculate the OOB error of the decision forest T as the Mean-Square Error (MSE):

$$OOB(T) = \frac{1}{|D^{'}|} \sum_{y_i \in D^{'}} \sum (y_i - \hat{y}_i)^2, \text{where } D^{'} = \bigcup_{b=1}^{B} \overline{D_b}$$

- If OOB error value per each observation is required, then calculate the prediction error for x_i:

$$OOB(x_i) = \ (y_i - \hat{y}_i)^2$$

Batch Processing

Decision forest regression follows the general workflow described in Decision Forest.

Training

For the description of the input and output, refer to Regression Usage Model.

In addition to the decision forest parameters described in Batch Processing, the training algorithm for decision forest regression has the following parameters:

Training Parameters for Decision Forest Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the decision forest regression. For CPU:

Parameter	Default Value	Description
		- defaultDense - default performance-oriented method - hist - inexact histogram computation method For GPU: - hist - inexact histogram computation method

Output

In addition to the output of regression described in Regression Usage Model, decision forest regression calculates the result of decision forest. For more details, refer to Batch Processing.

Prediction

For the description of the input and output, refer to Regression Usage Model.

In addition to the parameters of regression, decision forest regression has the following parameters at the prediction stage:

Prediction Parameters for Decision Forest Regression (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	The computation method used by the decision forest regression. The only prediction method supported so far is the default dense method.

Examples

oneAPI DPC++

Batch Processing:

- dpc_df_reg_hist_batch.cpp

oneAPI C++

Batch Processing:

- cpp_df_reg_dense_batch.cpp

C++ (CPU)

Batch Processing:

- df_reg_default_dense_batch.cpp
- df_reg_hist_dense_batch.cpp
- df_reg_traverse_model.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- DfRegDefaultDenseBatch.java
- DfRegHistDenseBatch.java

- DfRegTraverseModel.java

Python*

Batch Processing:

- decision_forest_regression_default_dense_batch.py
- decision_forest_regression_hist_batch.py
- decision_forest_regression_traverse_batch.py

Classification Decision Forest

Decision forest classifier is a special case of the Decision Forest model.

Details

Given:

- *n* feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of size *p*;
- their non-negative sample weights $w = (w_1, \ldots, w_n)$;
- the vector of class labels $y = (y_1, \ldots, y_n)$ that describes the class to which the feature vector x_i belongs, where $y_i \in \{0, 1, \ldots, C - 1\}$ and *C* is the number of classes.

The problem is to build a decision forest classifier.

Training Stage

Decision forest classifier follows the algorithmic framework of decision forest training with Gini impurity metrics as impurity metrics [Breiman84]. If sample weights are provided as input, the library uses a weighted version of the algorithm.

Gini index is an impurity metric, calculated as follows:

$$I_{Gini}(D) = 1 - \sum_{i=0}^{C-1} p_i^2$$

where

- *D* is a set of observations that reach the node;
- p_i is specified in the table below:

Decision Forest Classification: impurity calculations

Without sample weights	With sample weights
p_i is the observed fraction of observations that belong to class *i* in *D*	p_i is the observed weighted fraction of observations that belong to class *i* in *D*: $$p_i = \frac{\sum_{d \in \{d \in D \mid y_d = i\}} W_d}{\sum_{d \in D} W_d}$$

Prediction Stage

Given decision forest classifier and vectors $x_1, \ldots, x_r$, the problem is to calculate the labels for those vectors. To solve the problem for each given query vector x_i, the algorithm finds the leaf node in a tree in the forest that gives the classification response by that tree. The forest chooses the label y taking the majority of trees in the forest voting for that label.

Out-of-bag Error

Decision forest classifier follows the algorithmic framework for calculating the decision forest out-of-bag (OOB) error, where aggregation of the out-of-bag predictions in all trees and calculation of the OOB error of the decision forest is done as follows:

- For each vector x_i in the dataset *X*, predict its label $\hat{y}_i$ by having the majority of votes from the trees that contain x_i in their OOB set, and vote for that label.
- Calculate the OOB error of the decision forest *T* as the average of misclassifications:

$$OOB(T) = \frac{1}{|D'|} \sum_{y_i \in D'} I\{y_i \neq \hat{y}_i\}, \text{where } D' = \bigcup_{b=1}^{B} \overline{D_b}.$$

- If OOB error value per each observation is required, then calculate the prediction error for x_i: $OOB(x_i) = I\{y_i \neq \hat{y}_i\}$

Variable Importance

The library computes *Mean Decrease Impurity* (MDI) importance measure, also known as the *Gini importance* or *Mean Decrease Gini*, by using the Gini index as impurity metrics.

Usage of Training Alternative

To build a Decision Forest Classification model using methods of the Model Builder class of Decision Forest Classification, complete the following steps:

- Create a Decision Forest Classification model builder using a constructor with the required number of classes and trees.
- Create a decision tree and add nodes to it:
  - Use the `createTree` method with the required number of nodes in a tree and a label of the class for which the tree is created.
  - Use the `addSplitNode` and `addLeafNode` methods to add split and leaf nodes to the created tree. See the note below describing the decision tree structure.
  - After you add all nodes to the current tree, proceed to creating the next one in the same way.
- Use the `getModel` method to get the trained Decision Forest Classification model after all trees have been created.

NOTE Each tree consists of internal nodes (called non-leaf or split nodes) and external nodes (leaf nodes). Each split node denotes a feature test that is a Boolean expression, for example, f < `featureValue` or f = `featureValue`, where f is a feature and `featureValue` is a constant. The test type depends on the feature type: continuous, categorical, or ordinal. For more information on the test types, see Decision Tree.
The inducted decision tree is a binary tree, meaning that each non-leaf node has exactly two branches: true and false. Each split node contains `featureIndex`, the index of the feature used for the feature test in this node, and `featureValue`, the constant for the Boolean expression in the test. Each leaf node contains a `classLabel`, the predicted class for this leaf. For more information on decision trees, see Decision Tree.

Add nodes to the created tree in accordance with the pre-calculated structure of the tree. Check that the leaf nodes do not have children nodes and that the splits have exactly two children.

Examples

C++ (CPU)

- df_cls_dense_batch_model_builder.cpp
- df_cls_traversed_model_builder.cpp

Java*

NOTE There is no support for Java on GPU.

- DfClsDenseBatchModelBuilder.java
- DfClsTraversedModelBuilder.java

Python*

- df_cls_dense_batch_model_builder.py
- df_cls_traversed_model_builder.py

Batch Processing

Decision forest classification follows the general workflow described in Decision Forest and Classification Usage Model.

Training

In addition to the parameters of a classifier (see Classification Usage Model) and decision forest parameters described in Batch Processing, the training algorithm for decision forest classification has the following parameters:

Training Parameters for Decision Forest Classification (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the decision forest classification. For CPU: • `defaultDense` - default performance-oriented method • `hist` - inexact histogram computation method For GPU: • `hist` - inexact histogram computation method
`nClasses`	Not applicable	The number of classes. A required parameter.

Output

Decision forest classification calculates the result of regression and decision forest. For more details, refer to Batch Processing and Classification Usage Model.

Prediction

For the description of the input and output, refer to Classification Usage Model.

In addition to the parameters of a classifier, decision forest classification has the following parameters at the prediction stage:

Prediction Parameters for Decision Forest Classification (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the decision forest classification. The only prediction method supported so far is the default dense method.
`nClasses`	Not applicable	The number of classes. A required parameter.
`votingMethod`	`weighted`	A flag that specifies which method is used to compute probabilities and class labels: weighted • Probability for each class is computed as a sample mean of estimates across all trees, where each estimate is the normalized number of training samples for this class that were recorded in a particular leaf node for current input. • The algorithm returns the label for the class that gets the maximal value in a sample mean. unweighted • Probabilities are computed as normalized votes distribution across all trees of the forest. • The algorithm returns the label for the class that gets the majority of votes across all trees of the forest.

Examples

oneAPI DPC++

Batch Processing:

- dpc_df_cls_hist_batch.cpp

oneAPI C++

Batch Processing:

- cpp_df_cls_dense_batch.cpp

C++ (CPU)

Batch Processing:

- df_cls_default_dense_batch.cpp
- df_cls_hist_dense_batch.cpp
- df_cls_traverse_model.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- DfClsDefaultDenseBatch.java
- DfClsHistDenseBatch.java
- DfClsTraverseModel.java

Python*

Batch Processing:

- decision_forest_classification_default_dense_batch.py
- decision_forest_classification_hist_batch.py
- decision_forest_classification_traverse_batch.py

Decision Trees

- Decision Tree
- Regression Decision Tree
- Classification Decision Tree

Decision Tree

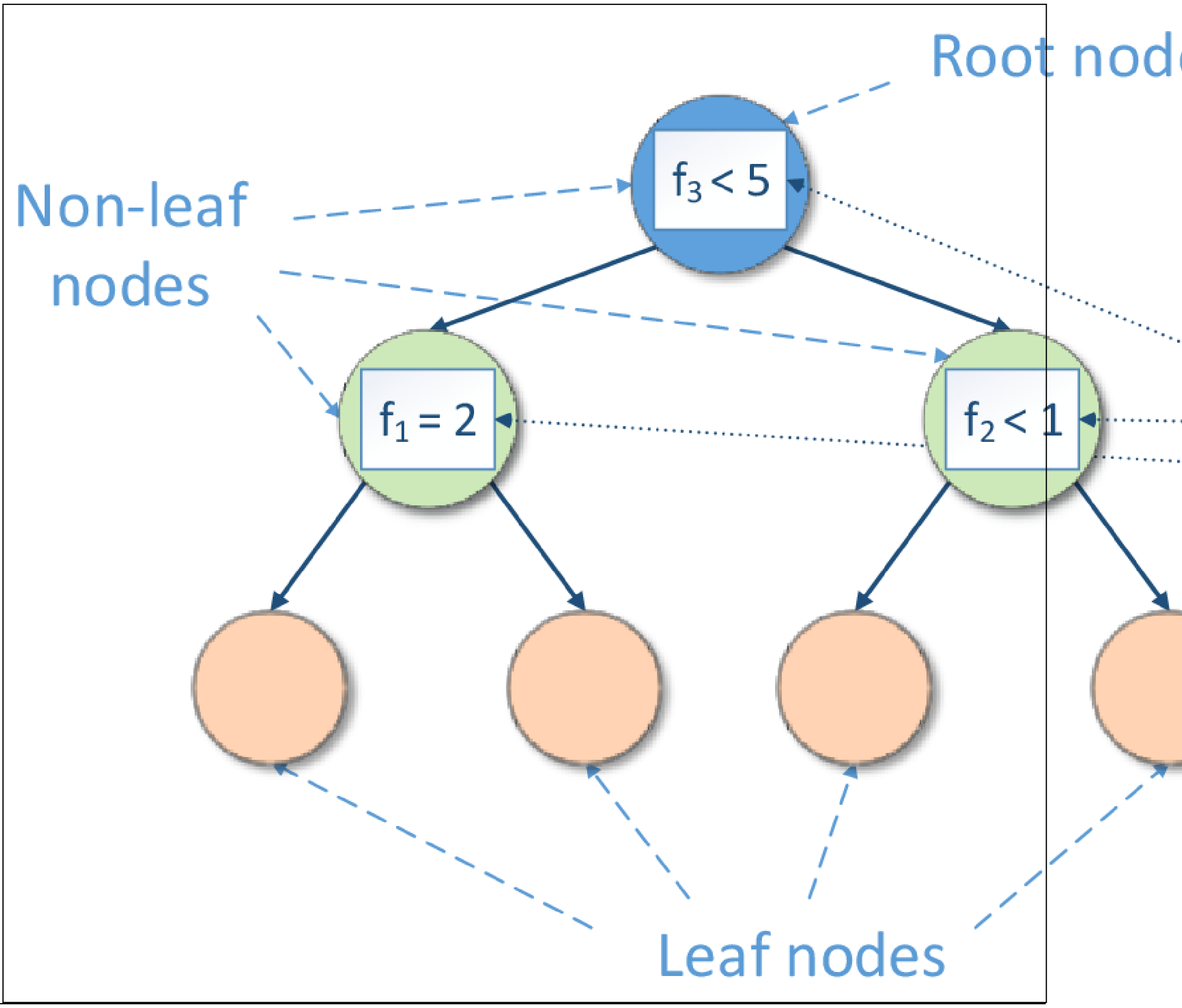
Decision trees partition the feature space into a set of hypercubes, and then fit a simple model in each hypercube. The simple model can be a prediction model, which ignores all predictors and predicts the majority (most frequent) class (or the mean of a dependent variable for regression), also known as 0-R or constant classifier.

Decision tree induction forms a tree-like graph structure as shown in the figure below, where:

- Each internal (non-leaf) node denotes a test on features
- Each branch descending from node corresponds to an outcome of the test

- Each external node (leaf) denotes the mentioned simple model

Decision Tree Structure



The test is a rule for partitioning of the feature space. The test depends on feature values. Each outcome of the test represents an appropriate hypercube associated with both the test and one of descending branches.

If the test is a Boolean expression (for example, $f < c$ or $f = c$, where *f* is a feature and *c* is a constant fitted during decision tree induction), the inducted decision tree is a binary tree, so its each non-leaf node has exactly two branches (‘true’ and ‘false’) according to the result of the Boolean expression.

Prediction is performed by starting at the root node of the tree, testing features by the test specified by this node, then moving down the tree branch corresponding to the outcome of the test for the given example. This process is then repeated for the subtree rooted at the new node. The final result is the prediction of the simple model at the leaf node.

Decision trees are often used in popular ensembles (e.g. boosting, bagging or decision forest).

Details

Given n feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p* and the vector of responses $y = y_1, \ldots, y_n$, the problem is to build a decision tree.

Split Criteria

The library provides the decision tree classification algorithm based on split criteria Gini index [Breiman84] and Information gain [Quinlan86], [Mitchell97]. See details in Classification Decision Tree.

The library also provides the decision tree regression algorithm based on the mean-squared error (MSE) [Breiman84]. See details in Regression Decision Tree.

Types of Tests

The library inducts decision trees with the following types of tests:

1. For continuous features, the test has a form of $f_j < constant$, where f_j is a feature, $j \in \{1, \ldots, p\}$.

   While enumerating all possible tests for each continuous feature, the *constant* can be any threshold as midway between sequential values for sorted unique values of given feature f_j that reach the node.
2. For categorical features, the test has a form of $f_j = constant$, where f_j is a feature, $j \in \{1, \ldots, p\}$.

   While enumerating all possible tests for each categorical feature, the *constant* can be any value of given feature f_j that reach the node.
3. For ordinal features, the test has a form of $f_j <> constant$ where f_j is a feature, $j \in \{1, \ldots, p\}$.

   While enumerating all possible tests for each ordinal feature, the *constant* can be any unique value except for the first one (in the ascending order) of given feature f_j that reach the node

Post-pruning

Optionally, the decision tree can be post-pruned using given *m* feature vectors $x_1^{pruning} = (x_{11}^{pruning}, \ldots, x_{1p}^{pruning}), \ldots, x_m^{pruning} = (x_{m1}^{pruning}, \ldots, x_{mp}^{pruning})$ of size *p*, a vector of class labels $y^{pruning} = (y_1^{pruning}, \ldots, y_m^{pruning})$ for classification or a vector of responses $y^{pruning} = (y_1^{pruning}, \ldots, y_m^{pruning})$ for regression. For more details about pruning, see [Quinlan87].

Pruned dataset can be some fraction of original training dataset (e.g. randomly chosen 30% of observations), but in this case those observations must be excluded from the training dataset.

Training Stage

The library uses the following algorithmic framework for the training stage.

The decision tree grows recursively from the root node, which corresponds to the entire training dataset. This process takes into account pre-pruning parameters: *maximum tree depth* and *minimum number of observations in the leaf node* . For each feature, each possible test is examined to be the best one according to the given split criterion. The best test is used to perform partition of the feature space into a set of hypercubes, and each hypercube represents appropriate part of the training dataset to accomplish the construction of each node at the next level in the decision tree.

After the decision tree is built, it can optionally be pruned by Reduced Error Pruning (REP) [Quinlan87] to avoid overfitting. REP assumes that there is a separate pruning dataset, each observation in which is used to get prediction by the original (unpruned) tree. For every non-leaf subtree, the change in mispredictions is examined over the pruning dataset that would occur if this subtree was replaced by the best possible leaf:

$$\Delta E = E_{leaf} - E_{subtree}$$

where

- $E_{subtree}$ is the number of errors (for classification) and the mean-squared error (MSE) (for regression) for a given subtree
- E_{leaf} is the number of errors (for classification) and the MSE (for regression) for the best possible leaf, which replaces the given subtree.

If the new tree gives an equal or fewer mispredictions ($\Delta E \leq 0$) and the subtree contains no subtree with the same property, the subtree is replaced by the leaf. The process continues until any further replacements increase mispredictions over the pruning dataset. The final tree is the most accurate subtree of the original tree with respect to the pruning dataset and is the smallest tree with that accuracy.

The training procedure contains the following steps:

1. Grow the decision tree (subtree):
   - If all observations contain the same class label (for classification) or same value of dependent variable (for regression), or pre-pruning parameters disallow further decision tree growing, construct a leaf node.
   - Otherwise
     - For each feature, sort given feature values and evaluate an appropriate split criterion for every possible test (see Split Criteria and Types of Tests for details).
     - Construct a node with a test corresponding to the best split criterion value.
     - Partition observations according to outcomes of the found test and recursively grow a decision subtree for each partition.
2. Post-prune the decision tree (see Post-pruning for details).

Prediction Stage

The library uses the following algorithmic framework for the prediction stage.

Given the decision tree and vectors $x_1, \ldots, x_r$, the problem is to calculate the responses for those vectors.

To solve the problem for each given vector x_i, the algorithm examines x_i by tests in split nodes to find the leaf node, which contains the prediction response.

Regression Decision Tree

Regression decision tree is a kind of decision trees described in Classification and Regression > Decision Tree.

Details

Given:

- n feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size p
- The vector of responses $y = (y_1, \ldots, y_n)$, where $y_i \in R$ describes the dependent variable for independent variables x_i.

The problem is to build a regression decision tree.

Split Criterion

The library provides the decision tree regression algorithm based on the mean-squared error (MSE) [Breiman84]:

$$ERROR processing math$$

Where

- $O(\tau)$ is the set of all possible outcomes of test τ
- D_v is the subset of *D*, for which outcome of τ is *v*, for example, $ERRORprocessingmath$.

The test used in the node is selected as $\underset{\tau}{\operatorname{argmax}} \Delta I_{MSE}(D,\ \tau)$. For binary decision tree with “true” and “false” branches,

$$ERRORprocessingmath$$

Training Stage

The regression decision tree follows the algorithmic framework of decision tree training described in Decision Tree.

Prediction Stage

The regression decision tree follows the algorithmic framework of decision tree prediction described in Decision Tree.

Given the regression decision tree and vectors $x1, \ldots, x_r$, the problem is to calculate the responses for those vectors.

Batch Processing

Decision tree regression follows the general workflow described in Regression Usage Model.

Training

At the training stage, decision tree regression has the following parameters:

Training Parameters for Decision Forest Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the decision tree regression. The only training method supported so far is the default dense method.
`pruning`	`reducedErrorPruning`	Method to perform post-pruning. Available options for the pruning parameter: • `reducedErrorPruning` - reduced error pruning. Provide dataForPruning and dependentVariablesForPruning inputs, if you use pruning. • `none` - do not prune.
`maxTreeDepth`	**0**	Maximum tree depth. Zero value means unlimited depth. Can be any non-negative number.
`minObservationsInLeafNodes`	**5**	Minimum number of observations in the leaf node. Can be any positive number.

Parameter	Default Value	Description
pruningFraction	**0.2**	Fraction of observations from training dataset to be used as observations for post-pruning via random sampling. The rest observations (with fraction $1 - pruningFraction$ to be used to build a decision tree). Can be any number in the interval (0, 1). If pruning is not used, all observations are used to build the decision tree regardless of this parameter value.
engine	**SharedPtr<engines::mt19937::Batch<> >()**	Pointer to the random number engine to be used for random sampling for reduced error post-pruning.

Prediction

At the prediction stage, decision tree regression has the following parameters:

Prediction Parameters for Decision Forest Regression (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	The computation method used by the decision tree regression. The only training method supported so far is the default dense method.

Examples

C++ (CPU)

Batch Processing:

- dt_reg_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- DtRegDenseBatch.java

Python*

Batch Processing:

- decision_tree_regression_batch.py
- decision_tree_regression_traverse_batch.py

Classification Decision Tree

Classification decision tree is a kind of a decision tree described in Decision Tree.

Details

Given:

- n feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p*
- The vector of class labels $y = (y_1, \ldots, y_n)$ that describes the class to which the feature vector x_i belongs, where $y_i \in \{0, 1, \ldots, C-1\}$ and C is the number of classes.

The problem is to build a decision tree classifier.

Split Criteria

The library provides the decision tree classification algorithm based on split criteria Gini index [Breiman84] and Information gain [Quinlan86], [Mitchell97]:

1. Gini index

$$I_{Gini}\left(D\right) = 1 - \sum_{i=0}^{C-1} p_i^2$$

where

- *D* is a set of observations that reach the node
- p_i is the observed fraction of observations with class *i* in *D*

To find the best test using Gini index, each possible test is examined using

$$ERRORprocessingmath$$

where

- $O(\tau)$ is the set of all possible outcomes of test τ
- D_v is the subset of *D*, for which outcome of τ is *v*, for example $ERRORprocessingmath$

The test to be used in the node is selected as $ERRORprocessingmath$. For binary decision tree with 'true' and 'false' branches, $ERRORprocessingmath$

2. Information gain

$$ERRORprocessingmath$$

where

- $O(\tau)$, *D*, D_v are defined above
- $I_{Entropy}\left(D\right) = -\sum_{i=0}^{C-1} p_i \mathrm{log} p_i$, with p_i defined above in Gini index.

Similarly to Gini index, the test to be used in the node is selected as $\underset{\tau}{\mathrm{argmax}} InfoGain\left(D, \tau\right)$.

For binary decision tree with 'true' and 'false' branches, $ERRORprocessingmath$

Training Stage

The classification decision tree follows the algorithmic framework of decision tree training described in Decision Tree.

Prediction Stage

The classification decision tree follows the algorithmic framework of decision tree prediction described in Decision Tree.

Given decision tree and vectors $x_i, \ldots, x_r$, the problem is to calculate the responses for those vectors.

Batch Processing

Decision tree classification follows the general workflow described in Classification Usage Model.

Training

In addition to common input for a classifier, decision trees can accept the following inputs that are used for post-pruning:

Training Input for Decision Tree Classification (Batch Processing)

Input ID	Input
dataForPruning	Pointer to the $m \times p$ numeric table with the pruning data set. This table can be an object of any class derived from NumericTable.
labelsForPruning	Pointer to the $m \times 1$ numeric table with class labels. This table can be an object of any class derived from NumericTable except PackedSymmetricMatrix and PackedTriangularMatrix.

At the training stage, decision tree classifier has the following parameters:

Training Parameters for Decision Tree Classification (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	The computation method used by the decision tree classification. The only training method supported so far is the default dense method.
nClasses	Not applicable	The number of classes. A required parameter.
splitCriterion	infoGain	Split criterion to choose the best test for split nodes. Available split criteria for decision trees: • gini - the Gini index • infoGain - the information gain
pruning	reducedErrorPruning	Method to perform post-pruning. Available options for the pruning parameter: • reducedErrorPruning - reduced error pruning. Provide dataForPruning and labelsForPruning inputs, if you use pruning. • none - do not prune.
maxTreeDepth	**0**	Maximum tree depth. Zero value means unlimited depth. Can be any non-negative number.
minObservationsInLeafNodes	**1**	Minimum number of observations in the leaf node. Can be any positive number.

Prediction

At the prediction stage, decision tree classifier has the following parameters:

Prediction Parameters for Decision Tree Classification (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the decision tree classification. The only training method supported so far is the default dense method.

Examples

C++ (CPU)

Batch Processing:

- dt_cls_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- DtClsDenseBatch.java

Python*

Batch Processing:

- decision_tree_classification_batch.py
- decision_tree_classification_traverse_batch.py

Gradient Boosted Trees

The library provides gradient boosted trees classification and regression algorithms based on an ensemble of regression (decision) trees trained using stochastic gradient boosting technique. *Regression tree* is a binary tree graph. Its internal (split) nodes represent a *decision function* used to select following (child) node at prediction stage. Its leaf (terminal) nodes represent the corresponding response values which are the result of prediction from the tree. For more details, see Decision Tree [Breiman84].

- Gradient Boosted Trees
- Regression Gradient Boosted Trees
- Classification Gradient Boosted Trees

Gradient Boosted Trees

Details

Given n feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of *np*-dimensional feature vectors and *n* responses $Y = \{y_1, \ldots, y_n\}$, the problem is to build a gradient boosted trees classification or regression model.

The tree ensemble model uses M additive functions to predict the output $\hat{y_i} = f(x) = \sum_{k=1}^{M} f_k(x_i), f_k \in F$ where $F = \ \left\{ f\ (x) = w_{q(x)}\ ,\ q : R^p \to T,\ w \in R^T \right\}$ is the space of regression trees, *T* is the number of leaves in the tree, *w* is a leaf weights vector, w_i is a score on *i*-th leaf. *q(x)* represents the structure of each tree that maps an observation to the corresponding leaf index.

Training procedure is an iterative functional gradient descent algorithm which minimizes objective function over function space by iteratively choosing a function (regression tree) that points in the negative gradient direction. The objective function is

$$L(f) = \sum_{i=1}^{n} l(y_i, f(x_i)) + \sum_{k=1}^{M} \Omega(f_k)$$

where *l(f)* is twice differentiable convex loss function and $\Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||$ is a regularization term that penalizes the complexity of the model defined by the number of leaves T and the L2 norm of the weights $||w||$ for each tree, γ and λ are regularization parameters.

Training Stage

Library uses the second-order approximation of objective function

$$L^{(k)}(f) \approx \sum_{i=1}^{n} (g_i f_k(x_i) + \ \frac{1}{2} h_i {f_k}^2(x_i)) + \Omega(f_k),$$

where $g_i = \ \frac{\partial l(y_i, {\hat{y_i}}^{(k-1)})}{\partial {\hat{y_i}}^{(k-1)}}$, $h_i = \ \frac{\partial^2 l(y_i, {\hat{y_i}}^{(k-1)})}{\partial^2 {\hat{y_i}}^{(k-1)}}$ and following algorithmic framework for the training stage.

Let $S = (X, Y)$ be the set of observations. Given the training parameters, such as the number of iterations *M*, loss function *l(f)*, regression tree training parameters, regularization parameters γ and λ, shrinkage (learning rate) parameter θ, the algorithm does the following:

- Find an initial guess ${\hat{y_i}}^{(0)}$, $i = 1, ..., n$
- For $k = 1, ..., M$:
  - Update g_i and h_i, $i = 1, ..., n$
  - Grow a regression tree $f_k \in F$ that minimizes the objective function $-\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$, where $G_j = \sum_{i \in I_j} g_j$, $H_j = \sum_{i \in I_j} h_j$, $I_j = \ \{i| \ (x_i) = j\}$, $j = 1, ..., T$.
  - Assign an optimal weight ${w_j}^* = \ \frac{G_j}{H_j + \lambda}$ to the leaf *j*, $j = 1, ..., T$.
  - Apply shrinkage parameter θ to the tree leafs and add the tree to the model
  - Update ${\hat{y_i}}^{(k)}$

The algorithm for growing the tree:

- Generate a bootstrap training set if required (stochastic gradient boosting) as follows: select randomly without replacement $N = f * n$ observations, where *f* is a fraction of observations used for training of one tree.

- Start from the tree with depth **0**.
- For each leaf node in the tree:
  - Choose a subset of feature for split finding if required (stochastic gradient boosting).
  - Find the best split that maximizes the gain:

$$\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+\ H_R+\lambda}-\gamma$$
$$-Stopwhenaterminationcriterionismet.$$

For more details, see [Chen2016].

The library supports the following termination criteria when growing the tree:

- **Minimal number of observations in a leaf node.** Node t is not processed if the subset of observations is smaller than the predefined value. Splits that produce nodes with the number of observations smaller than that value are not allowed.
- **Maximal tree depth.** Node t is not processed, if its depth in the tree reached the predefined value.
- **Minimal split loss.** Node t is not processed, if the best possible split is smaller than parameter γ.

Prediction Stage

Given a gradient boosted trees model and vectors $(x_1, \ldots, x_r)$, the problem is to calculate the responses for those vectors. To solve the problem for each given query vector x_i, the algorithm finds the leaf node in a tree in the ensemble which gives the response by that tree. Resulting response is based on an aggregation of responses from all trees in the ensemble. For detailed definition, see description of a specific algorithm.

Split Calculation Mode

The library supports two split calculation modes:

- exact - all possible split values are examined when searching for the best split for a feature.
- inexact - continuous features are bucketed into discrete bins and the possible splits are restricted by the buckets borders only.

Batch Processing

Gradient boosted trees classification and regression follows the general workflow described in Classification Usage Model and Regression Usage Model.

Training

For description of the input and output, refer to .

At the training stage, the gradient boosted trees batch algorithm has the following parameters:

Training Parameters for Gradient Boosted Trees (Batch Processing)

Parameter	Default Value	Description
`splitMethod`	`inexact`	Split computation mode. Possible values: • `inexact` - continuous features are bucketed into discrete bins and the buckets borders are examined only • `exact` - all possible splits for a given feature are examined

Parameter	Default Value	Description
maxIterations	**50**	Maximal number of iterations when training the model, defines maximal number of trees in the model.
maxTreeDepth	**6**	Maximal tree depth. If the parameter is set to **0** then the depth is unlimited.
shrinkage	**0.3**	Learning rate of the boosting procedure. Scales the contribution of each tree by a factor $(0, 1]$
minSplitLoss	**0**	Loss regularization parameter. Minimal loss reduction required to make a further partition on a leaf node of the tree. Range: $[0, \infty)$
lambda	**1**	L2 regularization parameter on weights. Range: $[0, \infty)$
observationsPerTreeFraction	**1**	Fraction of the training set S used for a single tree training, $0 < \text{observationsPerTreeFraction} \leq 1$. The observations are sampled randomly without replacement.
featuresPerNode	**0**	The number of features tried as the possible splits per node. If the parameter is set to **0**, all features are used.
minObservationsInLeafNode	**5**	Minimal number of observations in the leaf node.
memorySavingMode	false	If true then use memory saving (but slower) mode.
engine	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator.
maxBins	**256**	Used with inexact split method only. Maximal number of discrete bins to bucket continuous features. Increasing the number results in higher computation costs
minBinSize	**5**	Used with inexact split method only. Minimal number of observations in a bin.

Regression Gradient Boosted Trees

Gradient boosted trees regression is the special case of gradient boosted trees. For more details, see Gradient Boosted Trees.

Details

Given n feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of :math`n` *p*-dimensional feature vectors and a vector of dependent variables $y = (y_1, \ldots, y_n)$, the problem is to build a gradient boosted trees regression model that minimizes the loss function based on the predicted and true value.

Training Stage

Gradient boosted trees regression follows the algorithmic framework of gradient boosted trees training with following loss

$$L(y,\ f) = \frac{1}{2}(y - f(x))^2$$

Prediction Stage

Given the gradient boosted trees regression model and vectors $(x_1, \ldots, x_r)$, the problem is to calculate responses for those vectors. To solve the problem for each given feature vector x_i, the algorithm finds the leaf node in a tree in the ensemble, and the leaf node gives the tree response. The algorithm result is a sum of responses of all the trees.

Usage of Training Alternative

To build a Gradient Boosted Trees Regression model using methods of the Model Builder class of Gradient Boosted Tree Regression, complete the following steps:

- Create a Gradient Boosted Tree Regression model builder using a constructor with the required number of classes and trees.
- Create a decision tree and add nodes to it:
  - Use the `createTree` method with the required number of nodes in a tree and a label of the class for which the tree is created.
  - Use the `addSplitNode` and `addLeafNode` methods to add split and leaf nodes to the created tree. See the note below describing the decision tree structure.
  - After you add all nodes to the current tree, proceed to creating the next one in the same way.
- Use the `getModel` method to get the trained Gradient Boosted Trees Regression model after all trees have been created.

> **NOTE** Each tree consists of internal nodes (called non-leaf or split nodes) and external nodes (leaf nodes). Each split node denotes a feature test that is a Boolean expression, for example, f < `featureValue` or f = `featureValue`, where f is a feature and `featureValue` is a constant. The test type depends on the feature type: continuous, categorical, or ordinal. For more information on the test types, see Decision Tree.
> The inducted decision tree is a binary tree, meaning that each non-leaf node has exactly two branches: true and false. Each split node contains `featureIndex`, the index of the feature used for the feature test in this node, and `featureValue`, the constant for the Boolean expression in the test. Each leaf node contains a `classLabel`, the predicted class for this leaf. For more information on decision trees, see Decision Tree.
>
> Add nodes to the created tree in accordance with the pre-calculated structure of the tree. Check that the leaf nodes do not have children nodes and that the splits have exactly two children.

Examples

C++ (CPU)

- gbt_reg_traversed_model_builder.cpp

Java*

> **NOTE** There is no support for Java on GPU.

- GbtRegTraversedModelBuilder.java

Python*

- gbt_reg_traversed_model_builder.py

Batch Processing

Gradient boosted trees regression follows the general workflow described in Gradient Boosted Trees and Regression Usage Model.

Training

In addition to parameters of the gradient boosted trees described in Batch Processing, the gradient boosted trees regression training algorithm has the following parameters:

Training Parameters for Gradient Boosted Trees Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the gradient boosted trees regression. The only training method supported so far is the default dense method.
`loss`	`squared`	Loss function type.

Prediction

In addition to the common regression parameters, the gradient boosted trees regression has the following parameters at the prediction stage:

Prediction Parameters for Gradient Boosted Trees Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the gradient boosted trees regression. The only training method supported so far is the default dense method.
`numIterations`	**0**	An integer parameter that indicates how many trained iterations of the model should be used in prediction. The default value **0** denotes no limit. All the trained trees should be used.

Examples

C++ (CPU)

Batch Processing:

- gbt_reg_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- GbtRegDenseBatch.java

Python* with DPC++ support

Batch Processing:

- gradient_boosted_regression_batch.py

Python*

Batch Processing:

- gradient_boosted_regression_batch.py
- gradient_boosted_regression_traverse_batch.py

Classification Gradient Boosted Trees

Gradient boosted trees classification is the special case of gradient boosted trees. For more details, see Gradient Boosted Trees.

Details

Given n feature vectors $X = \{x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})\}$ of n p-dimensional feature vectors and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in \{0, 1, \ldots, C - 1\}$ and C is the number of classes, which describes the class to which the feature vector x_i belongs, the problem is to build a gradient boosted trees classifier.

Training Stage

Gradient boosted trees classification follows the algorithmic framework of gradient boosted trees training. For a classification problem with K classes, K regression trees are constructed on each iteration, one for each output class. The loss function is cross-entropy (multinomial deviance):

$$L(y, f) = -\sum_{k=1}^{K} (I(y = k) \ln p_k(x))$$

where $p_k(x) = \frac{e^{f_k(x)}}{\sum_{i=1}^{K} e^{f_i(x)}}$

Binary classification is a special case when single regression tree is trained on each iteration. The loss function is

$$L(y, f) = -(y \cdot \ln \sigma(f) + (1 - y) \ln(1 - \sigma(f)))$$

where $\sigma(f) = \frac{1}{1 + e^{-f}}$

Prediction Stage

Given the gradient boosted trees classifier model and vectors $(x_1, \ldots, x_r)$, the problem is to calculate labels for those vectors. To solve the problem for each given feature vector x_i, the algorithm finds the leaf node in a tree in the ensemble, and the leaf node gives the tree response. The algorithm computes a sum of responses of all the trees for each class and chooses the label y corresponding to the class with the maximal response value (highest class probability).

Usage of Training Alternative

To build a Gradient Boosted Trees Classification model using methods of the Model Builder class of Gradient Boosted Tree Classification, complete the following steps:

- Create a Gradient Boosted Tree Classification model builder using a constructor with the required number of classes and trees.
- Create a decision tree and add nodes to it:
  - Use the `createTree` method with the required number of nodes in a tree and a label of the class for which the tree is created.
  - Use the `addSplitNode` and addLeafNode methods to add split and leaf nodes to the created tree. See the note below describing the decision tree structure.
  - After you add all nodes to the current tree, proceed to creating the next one in the same way.
- Use the `getModel` method to get the trained Gradient Boosted Trees Classification model after all trees have been created.

> **NOTE** Each tree consists of internal nodes (called non-leaf or split nodes) and external nodes (leaf nodes). Each split node denotes a feature test that is a Boolean expression, for example, f < `featureValue` or f = `featureValue`, where f is a feature and `featureValue` is a constant. The test type depends on the feature type: continuous, categorical, or ordinal. For more information on the test types, see Decision Tree.
> The inducted decision tree is a binary tree, meaning that each non-leaf node has exactly two branches: true and false. Each split node contains featureIndex, the index of the feature used for the feature test in this node, and `featureValue`, the constant for the Boolean expression in the test. Each leaf node contains a classLabel, the predicted class for this leaf. For more information on decision trees, see Decision Tree.
>
> Add nodes to the created tree in accordance with the pre-calculated structure of the tree. Check that the leaf nodes do not have children nodes and that the splits have exactly two children.

Examples

C++ (CPU)

- gbt_cls_traversed_model_builder.cpp

Java*

> **NOTE** There is no support for Java on GPU.

- GbtClsTraversedModelBuilder.java

Python*

- gbt_cls_traversed_model_builder.py

Batch Processing

Gradient boosted trees classification follows the general workflow described in Gradient Boosted Trees and Classification Usage Model

Training

In addition to parameters of the gradient boosted trees described in Batch Processing, the gradient boosted trees classification training algorithm has the following parameters:

Training Parameters for Gradient Boosted Trees Classification (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the gradient boosted trees regression. The only training method supported so far is the default dense method.
`nClasses`	Not applicable	The number of classes. A required parameter.
`loss`	`crossEntropy`	Loss function type.

Prediction

In addition to the parameters of a classifier, the gradient boosted trees classifier has the following parameters at the prediction stage:

Prediction Parameters for Gradient Boosted Trees Classification (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the gradient boosted trees regression. The only training method supported so far is the default dense method.
`nClasses`	Not applicable	The number of classes. A required parameter.
`numIterations`	**0**	An integer parameter that indicates how many trained iterations of the model should be used in prediction. The default value **0** denotes no limit. All the trained trees should be used.

Examples

C++ (CPU)

Batch Processing:

- gbt_cls_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- GbtClsDenseBatch.java

Python*

Batch Processing:

- gradient_boosted_classification_batch.py
- gradient_boosted_classification_traverse_batch.py

Stump

- Classification Stump
  - Batch Processing
  - Examples
- Regression Stump
  - Batch Processing
  - Examples

Classification Stump

A Classification Decision Stump is a model that consists of a one-level decision tree where the root is connected to terminal nodes (leaves) [Friedman2017]. The library only supports stumps with two leaves. Two methods of split criterion are available: gini and information gain. See Classification Decision Tree for details.

Batch Processing

A classification stump follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Classification Usage Model.

At the training stage, a classification decision stump has the following parameters:

Training Parameters for Classification Stump (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`splitCriterion`	`decision_tree::classification::gini`	Split criteria for classification stump. Two split criterion are available: • `decision_tree::classification::gini` • `decision_tree::classification::infoGain` See Classification Decision Tree chapter for details.

Parameter	Default Value	Description
`varImportance`	`none`	**NOTE** Variable importance computation is not supported for current version of the library.
`nClasses`	**2**	The number of classes.

Prediction

For a description of the input and output, refer to Classification Usage Model.

At the prediction stage, a classification stump has the following parameters:

Training Parameters for Classification Stump (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`nClasses`	**2**	The number of classes.
`resultsToEvaluate`	`classifier::computeClassLabels`	The form of computed result: • `classifier::computeClassLabels` – the result contains the `NumericTable` of size $nimes1$ with predicted labels • `classifier::computeClassProbabilities` – the result contains the `NumericTable` of size $n \times \mathrm{nClasses}$ with probabilities to belong to each class

Examples

C++ (CPU)

Batch Processing:

- stump_cls_gini_dense_batch.cpp
- stump_cls_infogain_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- StumpClsGiniDenseBatch.java
- StumpClsInfogainDenseBatch.java

Python*

Batch Processing:

- stump_classification_batch.py

Regression Stump

A Regression Decision Stump is a model that consists of a one-level decision tree where the root is connected to terminal nodes (leaves) [Friedman2017]. The library only supports stumps with two leaves based on regression decision trees. The one method of split criteria is available: mse. See Regression Decision Tree for details.

Batch Processing

A regression stump follows the general workflow described in Regression Usage Model.

Training

For a description of the input and output, refer to Regression Usage Model.

At the training stage, a regression decision stump has the following parameters:

Training Parameters for Regression Stump (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`varImportance`	`none`	**NOTE** Variable importance computation is not supported for current version of the library.

Prediction

For a description of the input and output, refer to Regression Usage Model.

At the prediction stage, a regression stump has the following parameters:

Prediction Parameters for Regression Stump (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.

Examples

C++ (CPU)

Batch Processing:

stump_reg_mse_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

StumpRegMseDenseBatch.java

Python*

Batch Processing:

- stump_regression_batch.py

Linear and Ridge Regressions

To learn the details of Linear and Ridge regressions, see the following chapters:

- Linear Regression
- Ridge Regression

The following chapter covers the details of the computation process:

- Linear and Ridge Regressions Computation
  - Batch Processing
  - Online Processing
  - Distributed Processing
  - Examples

Linear Regression

Linear regression is a method for modeling the relationship between a dependent variable (which may be a vector) and one or more explanatory variables by fitting linear equations to observed data.

Details

Let $(x_1, \ldots, x_p)$ be a vector of input variables and $y = (y_1, \ldots, y_k)$ be the response. For each $j = 1, \ldots, k$, the linear regression model has the format [Hastie2009]:

$$y_j = \beta_{0j} + \beta_{1j}x_1 + \ldots + \beta_{pj}x_p$$

Here x_i, $i = 1, \ldots, p$, are referred to as independent variables, and y_j are referred to as dependent variables or responses.

Linear regression is called:

- **Simple Linear Regression** (if there is only one explanatory variable)
- **Multiple Linear Regression** (if the number of explanatory variables $p > 1$)

Training Stage

Let $(x_{11}, \ldots, x_{1p}, y_1, \ldots, x_{n1}, \ldots, x_{np}, y_n)$ be a set of training data, $n \gg p$. The matrix *X* of size $nimesp$ contains observations x_{ij}, $i = 1, \ldots, n$, $j = 1, \ldots, p$ of independent variables.

To estimate the coefficients $(\beta_{0j}, \ldots, \beta_{pj})$ one these methods can be used:

- Normal Equation system
- QR matrix decomposition

Prediction Stage

Linear regression based prediction is done for input vector $(x_1, \ldots, x_p)$ using the equation $y_j = \beta_{0j} + \beta_{1j}x_1 + \ldots + \beta_{pj}x_p$ for each $j = 1, \ldots, k$.

Usage of Training Alternative

To build a Linear Regression model using methods of the Model Builder class of Linear Regression, complete the following steps:

- Create a Linear Regression model builder using a constructor with the required number of responses and features.
- Use the `setBeta` method to add the set of pre-calculated coefficients to the model. Specify random access iterators to the first and the last element of the set of coefficients [ISO/IEC 14882:2011 §24.2.7]_.

> **NOTE** If your set of coefficients does not contain an intercept, `interceptFlag` is automatically set to `False`, and to `True`, otherwise.

- Use the `getModel` method to get the trained Linear Regression model.
- Use the `getStatus` method to check the status of the model building process. If `DAAL_NOTHROW_EXCEPTIONS` macros is defined, the status report contains the list of errors that describe the problems API encountered (in case of API runtime failure).

> **NOTE** If after calling the `getModel` method you use the `setBeta` method to update coefficients, the initial model will be automatically updated with the new β coefficients.

Examples

C++ (CPU)

- lin_reg_model_builder.cpp

Java*

> **NOTE** There is no support for Java on GPU.

- LinRegModelBuilder.java

Python*

- lin_reg_model_builder.py

Ridge Regression

The ridge regression method is similar to the least squares procedure except that it penalizes the sizes of the regression coefficients. Ridge regression is one of the most commonly used methods to overcome data multicollinearity.

Details

Let $(x_1, \ldots, x_p)$ be a vector of input variables and $y = (y_1, \ldots, y_k)$ be the response. For each $j = 1, \ldots, k$, the ridge regression model has the form similar to the linear regression model [Hoerl70], except that the coefficients are estimated by minimizing a different objective function [James2013]:

$$y_j = \beta_{0j} + \beta_{1j} x_1 + \ldots + \beta_{pj} x_p$$

Here x_i, $i = 1, \ldots, p$, are referred to as independent variables, and y_j are referred to as dependent variables or responses.

Training Stage

Let $(x_{11}, \ldots, x_{1p}, y_{11}, \ldots, y_{1k}), \ldots, (x_{n1}, \ldots, x_{np}, y_{n1}, \ldots, y_{nk})$ be a set of training data, $n \gg p$. The matrix *X* of size $nimesp$ contains observations x_ij, $i = 1, \ldots, n$, $j = 1, \ldots, p$, of independent variables.

For each y_j, $j = 1, \ldots, k$, the ridge regression estimates $(\beta_{0j}, \beta_{1j}, \ldots, \beta_{pj})$ by minimizing the objective function:

$$\sum_{i=1}^{n} (y_{ij} - \beta_{0j} - \sum_{q=1}^{p} (\beta_{qj} x_{iq}))^2 + \lambda_j \sum_{q=1}^{p} \beta_{qj}^2$$

where $ERRORprocessingmath$ are ridge parameters [Hoerl70], [James2013].

Prediction Stage

Ridge regression based prediction is done for input vector $(x_1, \ldots, x_p)$ using the equation $y_j = \beta_{0j} + \beta_{1j} x_1 + \ldots + \beta_{pj} x_p$ for each $j = 1, \ldots, k$.

Linear and Ridge Regressions Computation

Batch Processing

Linear and ridge regressions in the batch processing mode follow the general workflow described in Regression Usage Model.

Training

For a description of the input and output, refer to Regression Usage Model.

The following table lists parameters of linear and ridge regressions at the training stage. Some of these parameters or their values are specific to a linear or ridge regression algorithm.

Linear Regression

Training Parameters for Linear Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for linear regression training: - `defaultDense` - the normal equations method - `qrDense` - the method based on QR decomposition
`interceptFlag`	`true`	A flag that indicates a need to compute β_{0j}.

Ridge Regression

Training Parameters for Ridge Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Default computation method used by the ridge regression. The only method supported at the training stage is the normal equations method.
`ridgeParameters`	A numeric table of size $1imes1$ that contains the default ridge parameter equal to **1**.	The numeric table of size $1imesk$ (*k* is the number of dependent variables) or $1imes1$. The contents of the table depend on its size: - $size = 1 \times k$: values of the ridge parameters λ_j for $j = 1, \ldots, k$. - $size = 1 \times 1$: the value of the ridge parameter for each dependent variable $\lambda_1 = \ldots = \lambda_k$. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`interceptFlag`	`true`	A flag that indicates a need to compute β_{0j}.

Prediction

For a description of the input and output, refer to Regression Usage Model.

At the prediction stage, linear and ridge regressions have the following parameters:

Prediction Parameters for Linear and Ridge Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Default performance-oriented computation method, the only method supported by the regression based prediction.

Online Processing

You can use linear and ridge regression in the online processing mode only at the training stage.

This computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \ldots \text{nblocks}$.

Training

Linear and ridge regression training in the online processing mode follows the general workflow described in Regression Usage Model.

Linear and ridge regression training in the online processing mode accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Linear and Ridge Regression (Online Processing)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the current, *i*-th, data block.
`dependentVariables`	Pointer to the $n_i \times k$ numeric table with responses associated with the current, *i*-th, data block.

NOTE Both input tables can be an object of any class derived from `NumericTable`.

The following table lists parameters of linear and ridge regressions at the training stage in the online processing mode.

Linear Regression

Training Parameters for Linear Regression (Online Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for linear regression training: - `defaultDense` - the normal equations method - `qrDense` - the method based on QR decomposition
`interceptFlag`	`true`	A flag that indicates a need to compute β_{0_j}.

Ridge Regression

Training Parameters for Ridge Regression (Online Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Default computation method used by the ridge regression. The only method supported at the training stage is the normal equations method.
`ridgeParameters`	A numeric table of size $1imes1$ that contains the default ridge parameter equal to **1**.	The numeric table of size $1imesk$ (*k* is the number of dependent variables) or $1imes1$. The contents of the table depend on its size: - size = $1imesk$: values of the ridge parameters λ_j for $j = 1, \ldots, k$. - size = $1imes1$: the value of the ridge parameter for each dependent variable $\lambda_1 = \ldots = \lambda_k$.

Parameter	Default Value	Description
		NOTE This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`interceptFlag`	`true`	A flag that indicates a need to compute β_{0j}.

For a description of the output, refer to Regression Usage Model.

Distributed Processing

You can use linear and ridge regression in the distributed processing mode only at the training stage.

This computation mode assumes that the data set is split in `nblocks` blocks across computation nodes.

Training

Use the two-step computation schema for linear and ridge regression training in the distributed processing mode, as illustrated below:

- Step 1 - on Local Nodes
- Step 2 - on Master Node

Algorithm parameters

The following table lists parameters of linear and ridge regressions at the training stage in the distributed processing mode.

Linear Regression

Training Parameters for Linear Regression (Distributed Processing)

Parameter	Default Value	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available methods for linear regression training: • `defaultDense` - the normal equations method • `qrDense` - the method based on QR decomposition
`interceptFlag`	`true`	A flag that indicates a need to compute β_{0j}.

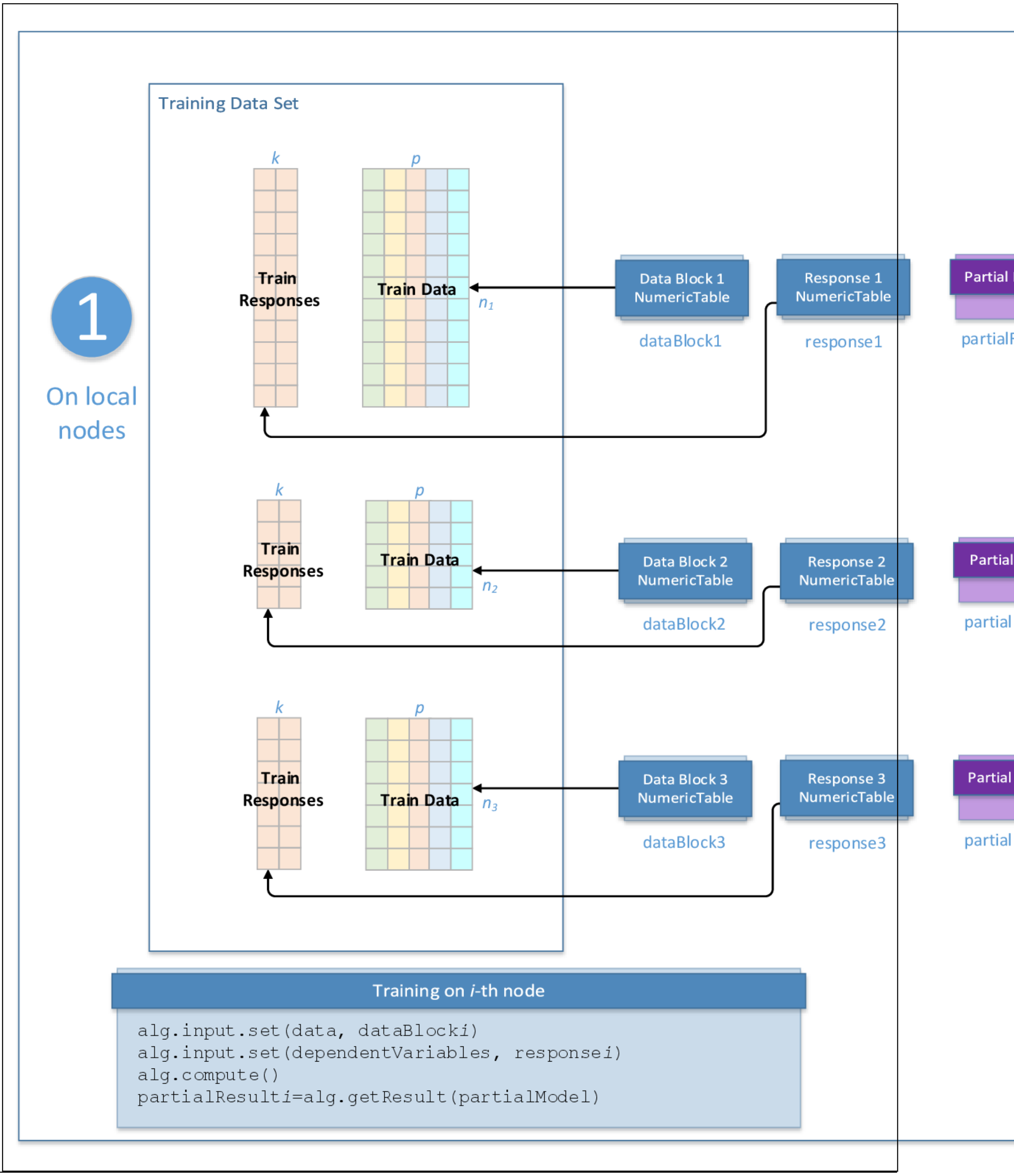
Ridge Regression

Training Parameters for Ridge Regression (Distributed Processing)

Parameter	Default Value	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Default computation method used by the ridge regression. The only method supported at the training stage is the normal equations method.
`ridgeParameters`	A numeric table of size $1imes1$ that contains the default ridge parameter equal to **1**.	The numeric table of size $1imesk$ (*k* is the number of dependent variables) or $1imes1$. The contents of the table depend on its size: • size = $1imesk$: values of the ridge parameters λ_j for $j = 1, \ldots, k$. • size = $1imes1$: the value of the ridge parameter for each dependent variable $\lambda_1 = \ldots = \lambda_k$. **NOTE** This parameter can be an object of any class derived from `NumericTable`, except for `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.
`interceptFlag`	`true`	A flag that indicates a need to compute β_{0_j}.

Step 1 - on Local Nodes

Linear and Ridge Regression Training: Distributed Processing, Step 1 - on Local Nodes



```
alg.input.set(data, dataBlocki)
alg.input.set(dependentVariables, responsei)
alg.compute()
partialResulti=alg.getResult(partialModel)
```

In this step, linear and ridge regression training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Linear and Ridge Regression (Distributed Processing, Step 1)

Input ID	Input
data	Pointer to the $n_i \times p$ numeric table that represents the *i*-th data block on the local node.
dependentVariables	Pointer to the $n_i \times k$ numeric table with responses associated with the *i*-th data block.

NOTE Both input tables can be an object of any class derived from `NumericTable`.

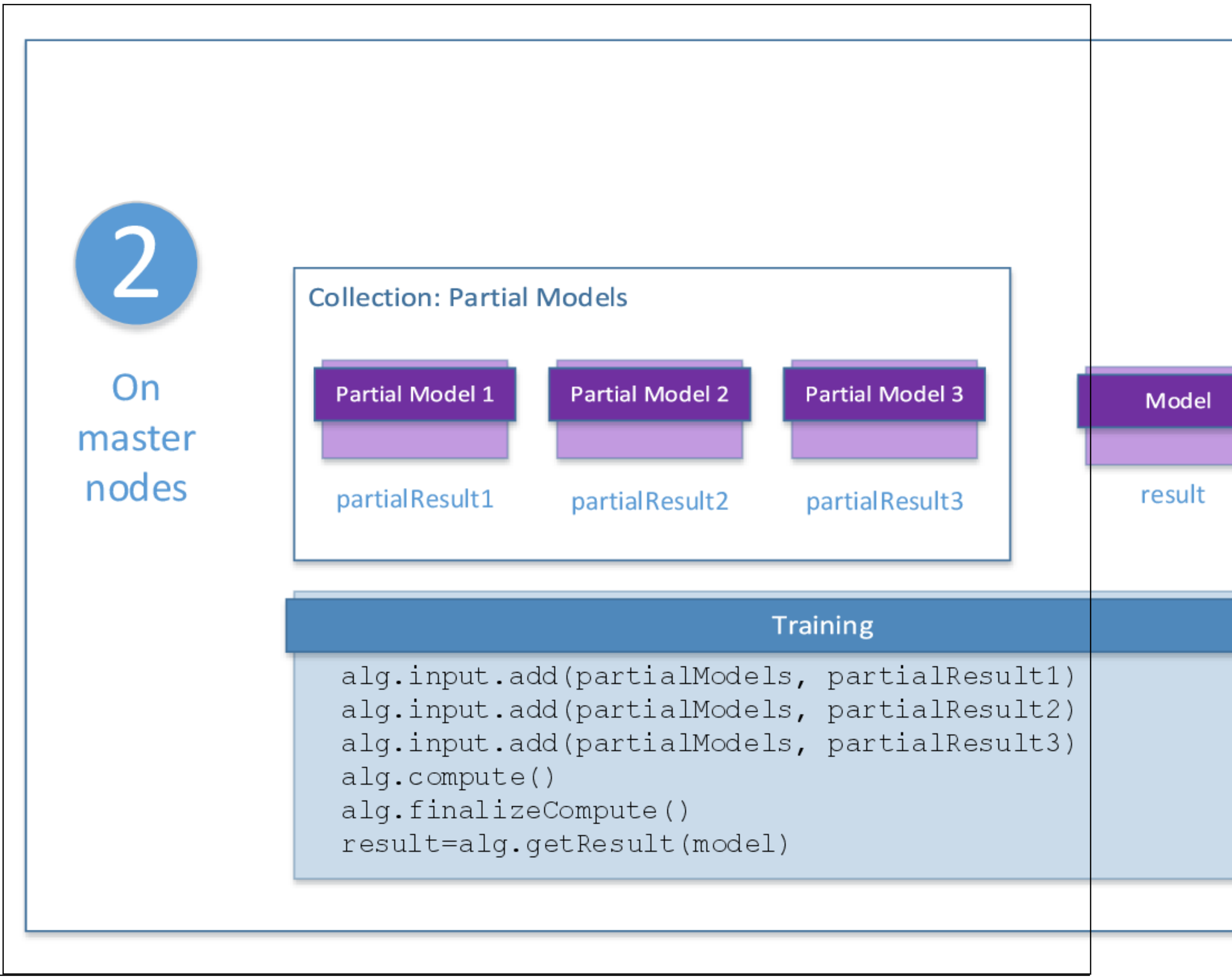
In this step, linear and ridge regression training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Linear and Ridge Regression (Distributed Processing, Step 1)

Result ID	Result
partialModel	Pointer to the partial linear regression model that corresponds to the *i*-th data block. The result can only be an object of the `Model` class.

Step 2 - on Master Node

Linear and Ridge Regression Training: Distributed Processing, Step 2 - on Master Node



```
alg.input.add(partialModels, partialResult1)
alg.input.add(partialModels, partialResult2)
alg.input.add(partialModels, partialResult3)
alg.compute()
alg.finalizeCompute()
result=alg.getResult(model)
```

In this step, linear and ridge regression training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Linear and Ridge Regression (Distributed Processing, Step 2)

Input ID	Input
`partialModels`	A collection of partial models computed on local nodes in Step 1. The collection contains objects of the `Model` class.

In this step, linear and ridge regression training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Linear and Ridge Regression (Distributed Processing, Step 2)

Result ID	Result
`model`	Pointer to the linear or ridge regression model being trained. The result can only be an object of the `Model` class.

Examples

C++ (CPU)

Batch Processing:

- lin_reg_norm_eq_dense_batch.cpp
- lin_reg_qr_dense_batch.cpp
- ridge_reg_norm_eq_dense_batch.cpp

Online Processing:

- lin_reg_norm_eq_dense_online.cpp
- lin_reg_qr_dense_online.cpp
- ridge_reg_norm_eq_dense_online.cpp

Distributed Processing:

- lin_reg_norm_eq_dense_distr.cpp
- lin_reg_qr_dense_distr.cpp
- ridge_reg_norm_eq_dense_distr.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- LinRegNormEqDenseBatch.java
- LinRegQRDenseBatch.java
- RidgeRegNormEqDenseBatch.java

Online Processing:

- LinRegNormEqDenseOnline.java
- LinRegQRDenseOnline.java
- RidgeRegNormEqDenseOnline.java

Distributed Processing:

- LinRegNormEqDenseDistr.java
- LinRegQRDenseDistr.java
- RidgeRegNormEqDenseDistr.java

Python* with DPC++ support

Batch Processing:

- linear_regression_batch.py

Python*

Batch Processing:

- linear_regression_batch.py
- ridge_regression_batch.py

Online Processing:

- linear_regression_streaming.py
- ridge_regression_streaming.py

Distributed Processing:

- linear_regression_spmd.py
- ridge_regression_spmd.py

LASSO and Elastic Net Regressions

To learn the details of LASSO and Elastic regressions, see the following chapters:

- LASSO
- Elastic Net

The following chapter covers the details of the computation process:

- LASSO and Elastic Net Computation

Least Absolute Shrinkage and Selection Operator (LASSO)

Least Absolute Shrinkage and Selection Operator (LASSO) is a method for modeling relationship between a dependent variable (which may be a vector) and one or more explanatory variables by fitting regularized least squares model. Trained LASSO model can produce sparse coefficients due to the use of L_1 regularization term. LASSO regression is widely used in feature selection tasks. For example, in the field of compressed sensing it is used to effectively identify relevant features associated with the dependent variable from a few observations with a large number of features. LASSO regression is also used to overcome multicollinearity of feature vectors in the training data set.

Details

Let $(x_1, \ldots, x_p)$ be a vector of input variables and $y = (y_1, \ldots, y_k)$ be the response. For each $j = 1, \ldots, k$, the LASSO model has the form similar to linear and ridge regression model [Hoerl70], except that the coefficients are trained by minimizing a regularized by L_1 penalty mean squared error (MSE) objective function.

$$y_j = \beta_{0j} + x_1\beta_{1j} + \ldots + x_p\beta_{pj}$$

Here x_i, $i = 1, \ldots, p$ are referred to as independent variables, y_j is referred to as dependent variable or response and $j = 1, \ldots, k$.

Training Stage

Let $(x_{11}, \ldots, x_{1p}, y_{11}, \ldots, y_{1k}) \ldots (x_{n1}, \ldots, x_{np}, y_{n1}, \ldots, y_{nk})$ be a set of training data (for regression task, $n >> p$, and for feature selection *p* could be greater than *n*). The matrix *X* of size $nimesp$ contains observations x_{ij}, $i = 1, \ldots, n$, $j = 1, \ldots, p$ of independent variables.

For each y_j, $j = 1, \ldots, k$, the LASSO regression estimates $(\beta_{0j}, \beta_{1j}, \ldots, \beta_{pj})$ by minimizing the objective function:

$$F_j(\beta) = \frac{1}{2n} \sum_{i=1}^{n} (y_{ij} - \beta_{0j} - \sum_{q=1}^{p} \beta_{qj} x_{iq})^2 + \lambda_{1j} \sum_{q=1}^{p} |\beta_{qj}|$$

In the equation above, the first term is a mean squared error function and the second one is a regularization term that penalizes the L_1 norm of vector β_j

For more details, see [Hastie2009].

By default, Coordinate Descent iterative solver is used to minimize the objective function. SAGA solver is also applicable for minimization.

Prediction Stage

For input vector of independent variables $(x_1, \dots, x_p)$, prediction based on LASSO regression is done using the equation

$$y_j = \beta_{0j} + x_1\beta_{1j} + \dots + x_p\beta_{pj}$$

where $j = 1, \dots, k$.

Elastic Net

Elastic Net is a method for modeling relationship between a dependent variable (which may be a vector) and one or more explanatory variables by fitting regularized least squares model. Elastic Net regression model has the special penalty, a sum of L1 and L2 regularizations, that takes advantage of both Ridge Regression and LASSO algorithms. This penalty is particularly useful in a situation with many correlated predictor variables [Friedman2010].

Details

Let $(x_1, \dots, x_p)$ be a vector of input variables and $y = (y_1, \dots, y_k)$ be the response. For each $j = 1, \dots, k$, the Elastic Net model has the form similar to linear and ridge regression models [Hoerl70] with one exception: the coefficients are estimated by minimizing mean squared error (MSE) objective function that is regularized by L_1 and L_2 penalties.

$$y_j = \beta_{0j} + x_1\beta_{1j} + \dots + x_p\beta_{pj}$$

Here x_i, $i = 1, \dots, p$, are referred to as independent variables, y_j, $j = 1, \dots, k$, is referred to as dependent variable or response.

Training Stage

Let $(x_{11}, \dots, x_{1p}, y_{11}, \dots, y_{1k}) \dots (x_{n1}, \dots, x_{np}, y_{n1}, \dots, y_{nk})$ be a set of training data (for regression task, $n >> p$, and for feature selection *p* could be greater than *n*). The matrix *X* of size $nimesp$ contains observations x_{ij}, $i = 1, \dots, n$, $j = 1, \dots, p$ of independent variables.

For each y_j, $j = 1, \dots, k$, the Elastic Net regression estimates $(\beta_{0j}, \beta_{1j}, \dots, \beta_{pj})$ by minimizing the objective function:

$$F_j(\beta) = \frac{1}{2n}\sum_{i=1}^{n}(y_{ij} - \beta_{0j} - \sum_{q=1}^{p}\beta_{qj}x_{iq})^2 + \lambda_{1j}\sum_{q=1}^{p}|\beta_{qj}| + \lambda_{2j}\frac{1}{2}\sum_{q=1}^{p}\beta_{qj}^2$$

In the equation above, the first term is a mean squared error function, the second and the third are regularization terms that penalize the L_1 and L_2 norms of vector β_j, where $\lambda_{1j} \geq 0$, $\lambda_{2j} \geq 0$, $j = 1, \dots, k$.

For more details, see [Hastie2009] and [Friedman2010].

By default, Coordinate Descent iterative solver is used to minimize the objective function. SAGA solver is also applicable for minimization.

Prediction Stage

Prediction based on Elastic Net regression is done for input vector $(x_1, \dots, x_p)$ using the equation $y_j = \beta_{0j} + x_1\beta_{1j} + \dots + x_p\beta_{pj}$ for each $j = 1, \dots, k$.

LASSO and Elastic Net Computation

Batch Processing

LASSO and Elastic Net algorithms follow the general workflow described in Regression Usage Model.

Training

For a description of common input and output parameters, refer to Regression Usage Model. Both LASSO and Elastic Net algorithms have the following input parameters in addition to the common input parameters:

Training Input for LASSO and Elastic Net (Batch Processing)

Input ID	Input
`weights`	Optional input. Pointer to the $1imesn$ numeric table with weights of samples. The input can be an object of any class derived from NumericTable except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable. By default, all weights are equal to 1.
`gramMatrix`	Optional input. Pointer to the $pimesp$ numeric table with pre-computed Gram Matrix. The input can be an object of any class derived from NumericTable except for CSRNumericTable. By default, the table is set to an empty numeric table. It is used only when the number of features is less than the number of observations.

Chosse the appropriate tab to see the parameters used in LASSO and Elastic Net batch training algorithms:

LASSO

Training Parameters for LASSO (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the LASSO regression. The only training method supported so far is the default dense method.
`interceptFlag`	`True`	A flag that indicates whether or not to compute
`lassoParameters`	A numeric table of size $1imes1$ that contains the default LASSO parameter equal to **0.1**.	L_1 coefficients: λ_i A numeric table of size $1imesk$ (where *k* is the number of dependent variables) or $1imes1$. The contents of the table depend on its size: • For the table of size $1imesk$, use the values of LASSO parameters λ_j for $j = 1, \ldots, k$.

Parameter	Default Value	Description
		• For the table of size $1imes1$, use the value of LASSO parameter for each dependant variable $\lambda_1 = \ldots = \lambda_k$. This parameter can be an object of any class derived from NumericTable, except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
`optimizationSolver`	Coordinate Descent solver	Optimization procedure used at the training stage.
`optResultToCompute`	**0**	The 64-bit integer flag that specifies which extra characteristics of the LASSO regression to compute. Provide the following value to request a characteristic: • `computeGramMatrix` for Computation Gram matrix
`dataUseInComputation`	`doNotUse`	A flag that indicates a permission to overwrite input data. Provide the following value to restrict or allow modification of input data: • `doNotUse` – restricts modification • `doUse` – allows modification

Elastic Net

Training Parameters for Elastic Net (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the Elastic Net regression. The only training method supported so far is the default dense method.
`interceptFlag`	`True`	A flag that indicates whether or not to compute
`penaltyL1`	A umeric table of size $1imes1$ that contains the default Elastic Net parameter equal to **0.5**.	L1 regularization coefficient (penaltyL1 is λ_1 as described in Elastic Net). The numeric table of size $1imesk$ (where *k* is the number of dependent variables) or $1imes1$. The contents of the table depend on its size: • For the table of size $1imesk$, the values of the Elastic Net parameters λ_{1j} for $j = 1, \ldots, k$.

Parameter	Default Value	Description
		• For the table of size $1imes1$, the values of the Elastic Net parameter for each dependent veriable $\lambda_{11} = ... = \lambda_{1k}$. This parameter can be an object of any class derived from NumericTable, except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
`penaltyL2`	A numeric table of size $1imes1$ that contains the default Elastic Net parameter equal to **0.5**.	L2 regularization coefficient (penaltyL2 is λ_2 as described in Elastic Net). The numeric table of size $1imesk$ (where *k* is the number of dependent variables) or $1imes1$. The contents of the table depend on its size: • For the table of size $1imesk$, the values of the Elastic Net parameters λ_{2j} for $j = 1, ..., k$. • For the table of size $1imes1$, the values of the Elastic Net parameter for each dependent veriable $\lambda_{21} = ... = \lambda_{2k}$. This parameter can be an object of any class derived from NumericTable, except for PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
`optimizationSolver`	Coordinate Descent solver	Optimization procedure used at the training stage.
`optResultToCompute`	**0**	The 64-bit integer flag that specifies which extra characteristics of the Elastic Net regression to compute. Provide the following value to request a characteristic: • `computeGramMatrix` for computation of the Gram Matrix
`dataUseInComputation`	`doNotUse`	A flag that indicates a permission to overwrite input data. Provide the following value to restrict or allow modification of input data: • `doNotUse` – restricts modification • `doUse` – allows modification

NOTE Common combinations of Elastic Net regularization parameters [Friedman2010] might be computed as shown below:

- compromise between L1 (lasso penalty) and L2 (ridge-regression penalty) regularization:

$$\mathrm{alpha} = \frac{\mathrm{penaltyL1}}{\mathrm{penaltyL1} + \mathrm{penaltyL2}}$$

- control full regularization:

$$\mathrm{lambda} = \mathrm{penaltyL1} + \mathrm{penaltyL2}$$

In addition, both LASSO and Elastic Net algorithms have the following optional results:

Training Output for LASSO and Elastic Net (Batch Processing)

Result ID	Result
`gramMatrix`	Pointer to the computed Gram Matrix with size $pimesp$

Prediction

For a description of the input and output, refer to Regression Usage Model.

At the prediction stage, LASSO and Elastic Net algorithms have the following parameters:

Prediction Parameters for LASSO and Elastic Net (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Default performance-oriented computation method, the only method supported by the regression-based prediction.

Examples

LASSO

C++: lasso_reg_dense_batch.cpp

Java*: LassoRegDenseBatch.java

Elastic Net

C++: elastic_net_dense_batch.cpp

Java*: ElasticNetDenseBatch.java

Performance Considerations

For better performance when the number of samples is larger than the number of features in the training data set, certain coordinates of gradient and Hessian are computed via the component of Gram matrix. When the number of features is larger than the number of observations, the cost of each iteration via Gram matrix depends on the number of features. In this case, computation is performed via residual update [Friedman2010].

To get the best overall performance for LASSO and Elastic Net training, do the following:

- If the number of features is less than the number of samples, use homogenous table.
- If the number of features is greater than the number of samples, use SOA layout rather than AOS layout.

k-Nearest Neighbors (kNN) Classifier

NOTE k-Nearest Neighbors Classifier is also available with oneAPI interfaces:

- k-Nearest Neighbors Classification and Search (k-NN)

k-Nearest Neighbors (kNN) classification is a non-parametric classification algorithm. The model of the kNN classifier is based on feature vectors and class labels from the training data set. This classifier induces the class of the query vector from the labels of the feature vectors in the training data set to which the query vector is similar. A similarity between feature vectors is determined by the type of distance (for example, Euclidian) in a multidimensional feature space.

Details

Given n feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p* and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in \{0, 1, \ldots, C-1\}$ and *C* is the number of classes, describes the class to which the feature vector x_i belongs, the problem is to build a kNN classifier.

Given a positive integer parameter *k* and a test observation x_0, the kNN classifier does the following:

1. Identifies the set N_0 of the k feature vectors in the training data that are closest to x_0 according to the distance metric
2. Estimates the conditional probability for the class *j* as the fraction of vectors in N_0 whose labels y are equal to *j*
3. Assigns the class with the largest probability to the test observation x_0

On CPU, kNN classification might use K-D tree, a space-partitioning data structure, or Brute Force search to find nearest neighbors, while on GPU only Brute Force search is available.

K-D tree

On CPU, the library provides kNN classification based on multidimensional binary search tree (K-D tree, where D means the dimension and K means the number of dimensions in the feature space). For more details, see [James2013], [Patwary2016].

oneDAL version of the kNN algorithm with K-D trees uses the PANDA algorithm [Patwary2016].

Each non-leaf node of a tree contains the identifier of a feature along which to split the feature space and an appropriate feature value (a cut-point) that defines the splitting hyperplane to partition the feature space into two parts. Each leaf node of the tree has an associated subset (a bucket) of elements of the training data set. Feature vectors from any bucket belong to the region of the space defined by tree nodes on the path from the root node to the respective leaf.

Brute Force

Brute Force kNN algorithm calculates the squared distances from each query feature vector to each reference feature vector in the training data set. Then, for each query feature vector it selects *k* objects from the training set that are closest to that query feature vector. For details, see [Li2015], [Verma2014].

Training Stage

Training using K-D Tree

For each non-leaf node, the process of building a K-D tree involves the choice of the feature (that is, dimension in the feature space) and the value for this feature (a cut-point) to split the feature space. This procedure starts with the entire feature space for the root node of the tree, and for every next level of the tree deals with ever smaller part of the feature space.

The PANDA algorithm constructs the K-D tree by choosing the dimension with the maximum variance for splitting [Patwary2016].

Therefore, for each new non-leaf node of the tree, the algorithm computes the variance of values that belong to the respective region of the space for each of the features and chooses the feature with the largest variance. Due to high computational cost of this operation, PANDA uses a subset of feature values to compute the variance.

PANDA uses a sampling heuristic to estimate the data distribution for the chosen feature and chooses the median estimate as the cut-point.

PANDA generates new K-D tree levels until the number of feature vectors in a leaf node gets less or equal to a predefined threshold. Once the threshold is reached, PANDA stops growing the tree and associates the feature vectors with the bucket of the respective leaf node.

Training using Brute Force

During training with the Brute Force approach, the algorithm stores all feature vectors from the training data set to calculate their distances to the query feature vectors.

Prediction Stage

Given kNN classifier and query vectors $x_0, \ldots, x_r$, the problem is to calculate the labels for those vectors.

Prediction using K-D Tree

To solve the problem for each given query vector x_i, the algorithm traverses the K-D tree to find feature vectors associated with a leaf node that are closest to x_i. During the search, the algorithm limits exploration of the nodes for which the distance between the query vector and respective part of the feature space is not less than the distance from the k^{th} neighbor. This distance is progressively updated during the tree traverse.

Prediction using Brute Force

To solve the problem, the algorithm computes distances between vectors from training and testing sets: $d_{ij} = \text{distance\_metric}(x_i^{\text{test}}, x_j^{\text{train}})$. For example, if Euclidean distance is used, d_{ij} would be the following:

$$d_{ij} = \sum_{k=1}^{p} (x_{ik}^{\text{test}} - x_{jk}^{\text{train}})^2$$

K training vectors with minimal distance to the testing vector are the nearest neighbors the algorithms searches for.

Batch Processing

kNN classification follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Usage Model: Training and Prediction.

At the training stage, both Brute Force and K-D tree based kNN classifier have the following parameters:

Training Parameters for k-Nearest Neighbors Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by kNN classification. The only training method supported so far is the default dense method.
`nClasses`	**2**	The number of classes.
`dataUseInModel`	`doNotUse`	A parameter to enable/disable use of the input data set in the kNN model. Possible values: • `doNotUse` - the algorithm does not include the input data and labels in the trained kNN model but creates a copy of the input data set. • `doUse` - the algorithm includes the input data and labels in the trained kNN model. K-D tree based kNN reorders feature vectors and corresponding labels in the input data set or its copy to improve performance at the prediction stage. If the value is `doUse`, do not deallocate the memory for input data and labels.
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally to perform sampling needed to choose dimensions and cut-points for the K-D tree.

Prediction

For a description of the input and output, refer to Usage Model: Training and Prediction.

At the prediction stage, both Brute Force and K-D tree based kNN classifier have the following parameters:

Prediction Parameters for k-Nearest Neighbors Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used kNN classification. The only prediction method supported so far is the default dense method.
`nClasses`	**2**	The number of classes.
k	**1**	The number of neighbors.
`resultsToCompute`	**0**	The 64-bit integer flag that specifies which extra characteristics of the kNN algorithm to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:

Parameter	Default Value	Description
		• `computeIndicesOfNeighbors` • `computeDistances`
`voteWeights`	`voteUniform`	The voting method for prediction: • `voteUniform` – Uniform weighting is used. All neighbors weight equally. • `voteDistance` – Inverse-distance weighting is used. The closer to the query point the neighbor is, the more it weights.

Output

In addition to classifier output, kNN calculates the results described below. Pass the `Result ID` as a parameter to the methods that access the result of your algorithm.

Output for k-Nearest Neighbors Classifier (Batch Processing)

Result ID	Result
`indices`	A numeric table $nimesk$ containing indices of rows from training dataset that are nearest neighbors computed when the `computeIndicesOfNeigtbors` option is on. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`.
`distances`	A numeric table $nimesk$ containing distances to nearest neighbors computed when the `computeDistances` option is on. **NOTE** By default, this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`.

Examples

oneAPI DPC++

Batch Processing:

- dpc_knn_cls_brute_force_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_knn_cls_kd_tree_dense_batch.cpp

C++ (CPU)

Batch Processing:

- kdtree_knn_dense_batch.cpp
- bf_knn_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- KDTreeKNNDenseBatch.java
- BFKNNDenseBatch.java

Python* with DPC++ support

Batch Processing:

- bf_knn_classification_batch.py

Python*

Batch Processing:

- kdtree_knn_classification_batch.py
- bf_knn_classification_batch.py

Implicit Alternating Least Squares

The library provides the Implicit Alternating Least Squares (implicit ALS) algorithm [Fleischer2008], based on collaborative filtering.

Details

Given the input dataset $R = \{r_{ui}\}$ of size $m \times n$, where m is the number of users and n is the number of items, the problem is to train the Alternating Least Squares (ALS) model represented as two matrices: *X* of size $m \times f$, and *Y* of size $f \times n$, where *f* is the number of factors. The matrices *X* and *Y* are the factors of low-rank factorization of matrix *R*:

$$R \approx X \cdot Y$$

Initialization Stage

Initialization of the matrix Y can be done using the following method: for each $i = 1, \ldots, n$ $y_{1i} = \frac{1}{m} \sum_{u=1}^{m} r_{ui}$ and y_{ki} are independent random numbers uniformly distributed on the interval $(0, 1)$, $k = 2, \ldots, f$.

Training Stage

The ALS model is trained using the implicit ALS algorithm [Hu2008] by minimizing the following cost function:

$$\min_{x_*, y_* u,i} \Sigma c_{ui} \left(p_{ui} - x_u^T y_i\right)^2 + \lambda \left(\Sigma_u n_{x_u} \|x_u\|^2 + \Sigma_i m_{y_i} \|y_i\|^2\right),$$

where:

- p_{ui} indicates the preference of user u of item i:

$$p_{ui} = \begin{cases} 1, & r_{ui} > \epsilon \\ 0, & r_{ui} \leq \epsilon \end{cases}$$

- ϵ is the threshold used to define the preference values. $\epsilon = 0$ is the only threshold valu supported so far.
- $c_{ui} = 1 + \alpha r_{ui}$, c_{ui} measures the confidence in observing p_{ui}

- α is the rate of confidence
- r_{ui} is the element of the matrix *R*
- λ is the parameter of the regularization
- n_{x_u}, m_{y_i} denote the number of ratings of user *u* and item *i* respectively

Prediction Stage

Prediction of Ratings

Given the trained ALS model and the matrix *D* that describes for which pairs of factors *X* and *Y* the rating should be computed, the system calculates the matrix of recommended ratings Res:

$res_{ui} = \sum_{j=1}^{f} x_{uj} y_{ji}$, if $d_{ui} \neq 0$, $u = 1, \ldots, m$, $i = 1, \ldots n$.

Initialization

For initialization, the following computation modes are available:

- Batch Processing
- Distributed Processing

Computation

The following computation modes are available:

- Batch Processing
- Distributed processing for training and prediction of ratings

Examples

C++ (CPU)

Batch Processing:

- impl_als_dense_batch.cpp
- impl_als_csr_batch.cpp

Distributed Processing:

- impl_als_csr_distr.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- ImplAlsDenseBatch.java
- ImplAlsCSRBatch.java

Distributed Processing:

- ImplAlsCSRDistr.java

Python*

Batch Processing:

- implicit_als_batch.py

Performance Considerations

To get the best overall performance of the implicit ALS recommender:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the algorithmFPType class template parameter.
- If input data is sparse, use CSR numeric tables.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Batch Processing

Input

Initialization of item factors for the implicit ALS algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Initialization (Batch Processing)

Input ID	Input
`data`	Pointer to the $m \times n$ numeric table with the mining data. The input can be an object of any class derived from `NumericTable` except `PackedTriangularMatrix` and `PackedSymmetricMatrix`.

Parameters

Initialization of item factors for the implicit ALS algorithm has the following parameters:

Parameters for Implicit Alternating Least Squares Initialization (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available computation methods: • `defaultDense` - performance-oriented method • `fastCSR` - performance-oriented method for CSR numeric tables
`nFactors`	**10**	The total number of factors.
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally at the initialization step.

Output

Initialization of item factors for the implicit ALS algorithm calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Implicit Alternating Least Squares Initialization (Batch Processing)

Result ID	Result
`model`	The model with initialized item factors. The result can only be an object of the `Model` class.

Distributed Processing

The distributed processing mode assumes that the data set R is split in `nblocks` blocks across computation nodes.

Parameters

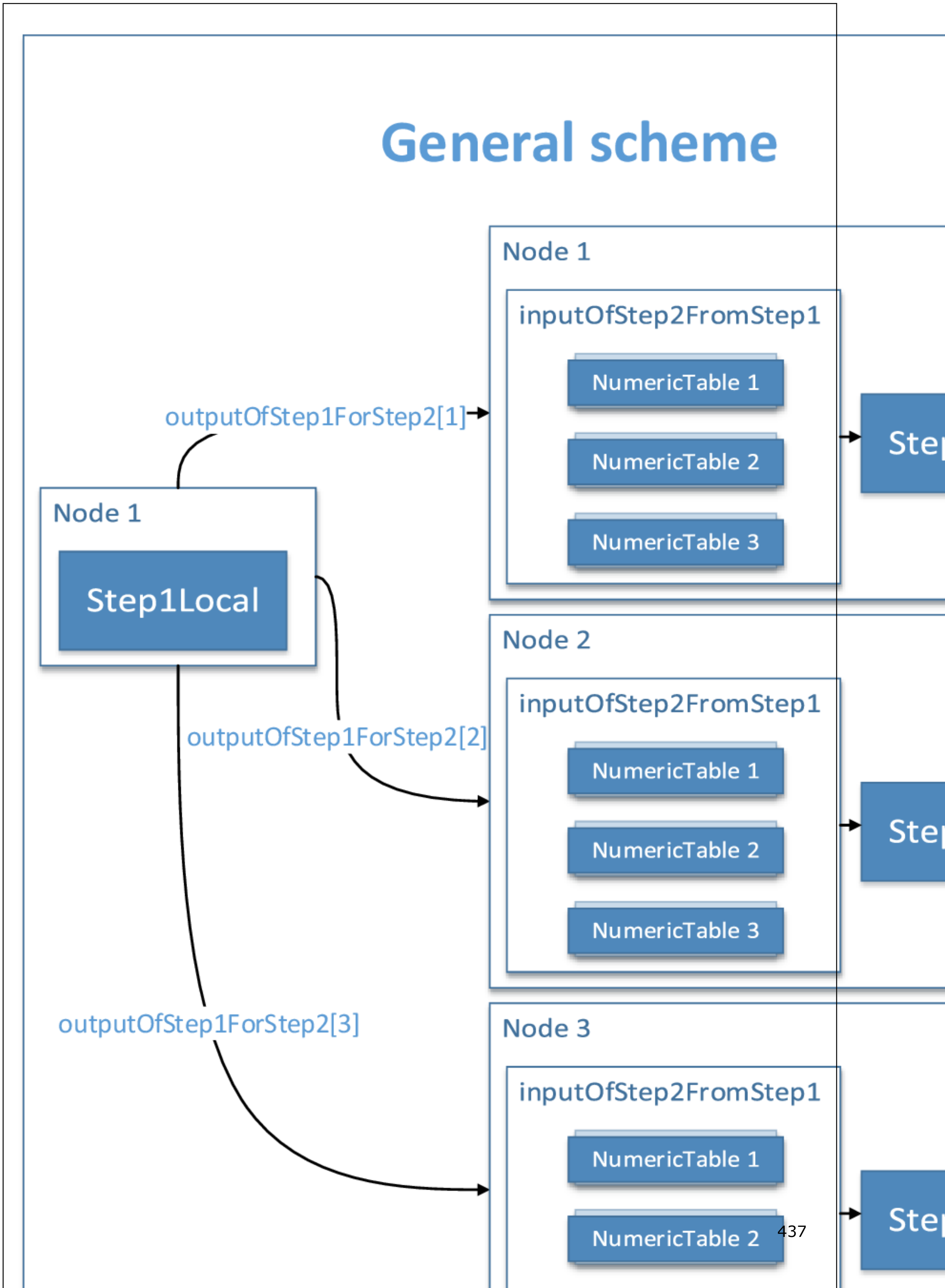
In the distributed processing mode, initialization of item factors for the implicit ALS algorithm has the following parameters:

Parameters for Implicit Alternating Least Squares Initialization (Distributed Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`fastCSR`	Performance-oriented computation method for CSR numeric tables, the only method supported by the algorithm.
`nFactors`	**10**	The total number of factors.
`fullNUsers`	**0**	The total number of users *m*.
`partition`	Not applicable	A numeric table of size either $1imes1$ that provides the number of input data parts or $(\mathrm{nblocks} + 1) \times 1$, where `nblocks` is the number of input data parts, and the *i*-th element contains the offset of the transposed *i*-th data part to be computed by the initialization algorithm.
`engine`	**SharePtr< engines:: mt19937:: Batch>()**	Pointer to the random number generator engine that is used internally at the initialization step.

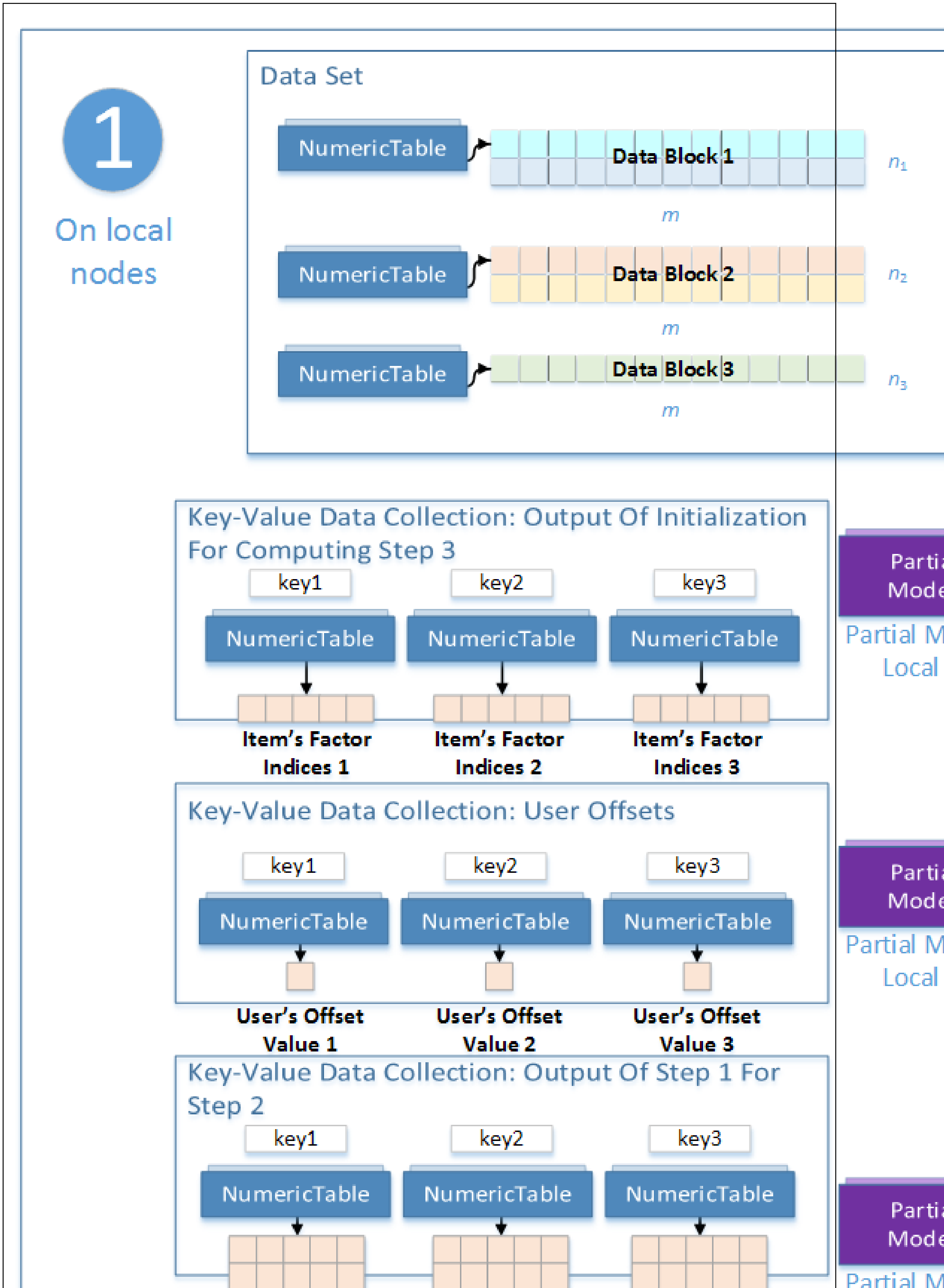
To initialize the implicit ALS algorithm in the distributed processing mode, use the one-step process illustrated by the following diagram for $\mathrm{nblocks} = 3$:

Implicit Alternating Least Squares Initialization: General Schema of Distributed Processing



Step 1 - on Local Nodes

Implicit Alternating Least Squares Initialization: Distributed Processing, Step 1 - on Local Nodes



Input

In the distributed processing mode, initialization of item factors for the implicit ALS algorithm accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Initialization (Distributed Processing, Step 1)

Input ID	Input
`dataColumnSlice`	An $n_i \times m$ numeric table with the part of the input data set. Each node holds n_i rows of the full transposed input data set R^T. The input should be an object of `CSRNumericTable` class.

Output

In the distributed processing mode, initialization of item factors for the implicit ALS algorithm calculates the results described below. Pass the `Partial Result ID` as a parameter to the methods that access the results of your algorithm. Partial results that correspond to the `outputOfInitForComputeStep3` and `offsets` Partial Result IDs should be transferred to Step 3 of the distributed ALS training algorithm.

Output of Initialization for Computing Step 3 (`outputOfInitForComputeStep3`) is a key-value data collection that maps components of the partial model on the *i*-th node to all local nodes. Keys in this data collection are indices of the nodes and the value that corresponds to each key *i* is a numeric table that contains indices of the factors of the items to be transferred to the *i*-th node on Step 3 of the distributed ALS training algorithm.

User Offsets (`offsets`) is a key-value data collection, where the keys are indices of the nodes and the value that correspond to the key *i* is a numeric table of size $1imes1$ that contains the value of the starting offset of the user factors stored on the *i*-th node.

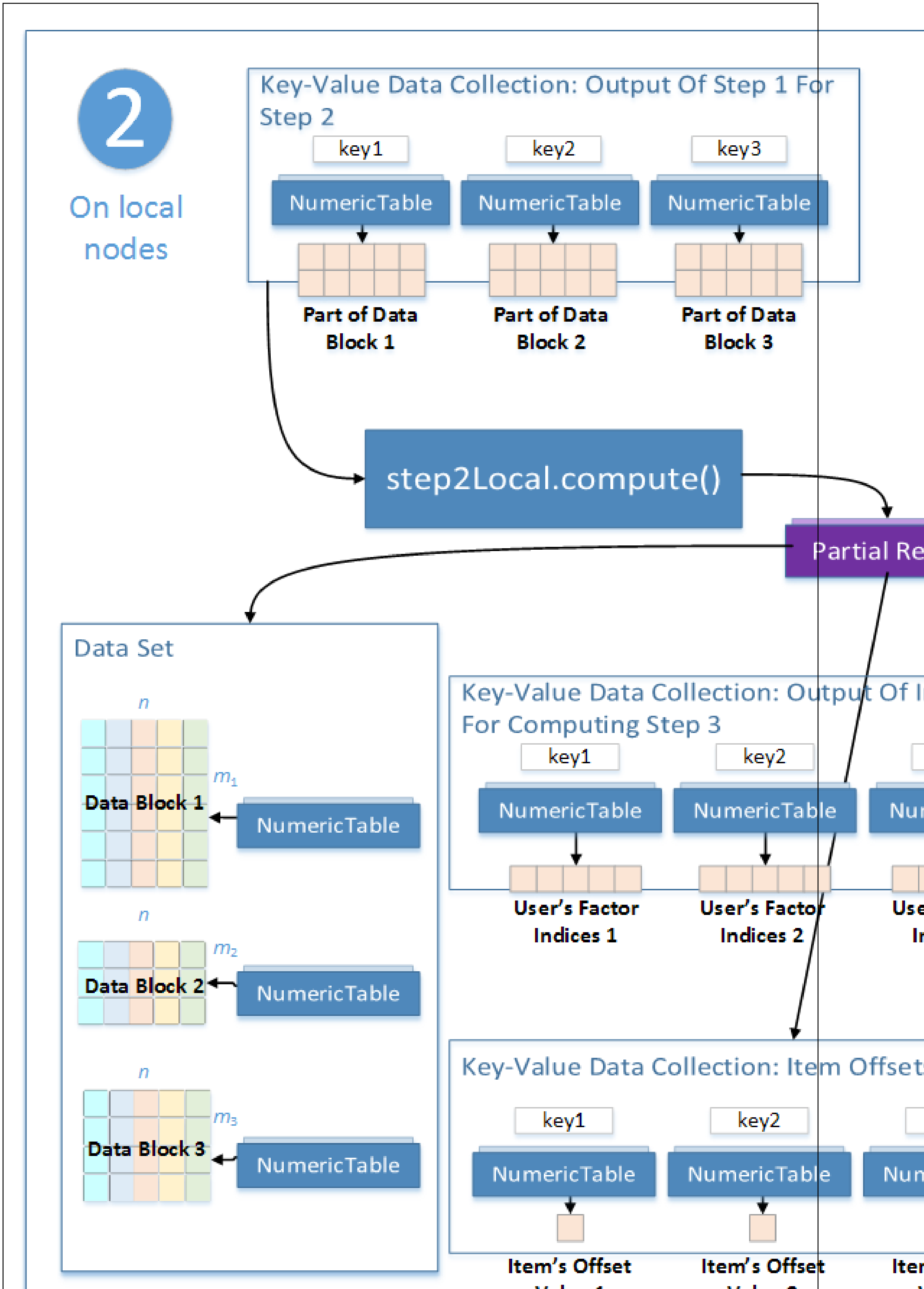
For more details, see Algorithms.

Output for Implicit Alternating Least Squares Initialization (Distributed Processing, Step 1)

Partial Result ID	Result
`partialModel`	The model with initialized item factors. The result can only be an object of the `PartialModel` class.
`outputOfInitForComputeStep3`	A key-value data collection that maps components of the partial model to the local nodes.
`offsets`	A key-value data collection of size `nblocks` that holds the starting offsets of the factor indices on each node.
`outputOfStep1ForStep2`	A key-value data collection of size `nblocks` that contains the parts of the input numeric table: *j* -th element of this collection is a numeric table of size $m_j \times n_i$, where $m_1 + \ldots + m_{\text{nblocks}} = m$ and the values m_j are defined by the `partition` parameter.

Step 2 - on Local Nodes

Implicit Alternating Least Squares Initialization: Distributed Processing, Step 2 - on Local Nodes



Input

This step uses the results of the previous step.

Input for Implicit Alternating Least Squares Initialization (Distributed Processing, Step 3)

Input ID	Input
`inputOfStep2FromStep1`	A key-value data collection of size nblocks that contains the parts of the input data set: *i* -th element of this collection is a numeric table of size $m_i \times n_i$. Each numeric table in the collection should be an object of CSRNumericTable class.

Output

In this step, implicit ALS initialization calculates the partial results described below. Pass the `Partial Result ID` as a parameter to the methods that access the results of your algorithm. Partial results that correspond to the `outputOfInitForComputeStep3` and `offsets` Partial Result IDs should be transferred to Step 3 of the distributed ALS training algorithm.

Output of Initialization for Computing Step 3 (`outputOfInitForComputeStep3`) is a key-value data collection that maps components of the partial model on the *i*-th node to all local nodes. Keys in this data collection are indices of the nodes and the value that corresponds to each key i is a numeric table that contains indices of the user factors to be transferred to the i-th node on Step 3 of the distributed ALS training algorithm.

Item Offsets (`offsets`) is a key-value data collection, where the keys are indices of the nodes and the value that correspond to the key *i* is a numeric table of size $1imes1$ that contains the value of the starting offset of the item factors stored on the *i*-th node.

For more details, see Algorithms.

Output for Implicit Alternating Least Squares Initialization (Distributed Processing, Step 2)

Partial Result ID	Result
`dataRowSlice`	An $m_j \times n$ numeric table with the mining data. *j*-th node gets m_j rows of the full input data set *R*.
`outputOfInitForComputeStep3`	A key-value data collection that maps components of the partial model to the local nodes.
`offsets`	A key-value data collection of size `nblocks` that holds the starting offsets of the factor indices on each node.

Batch Processing

Training

For a description of the input and output, refer to Recommendation Systems Usage Model.

At the training stage, the implicit ALS recommender has the following parameters:

Training Parameters for Implicit Alternating Least Squares Computaion (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available computation methods: • defaultDense - performance-oriented method • fastCSR - performance-oriented method for CSR numeric tables
nFactors	**10**	The total number of factors.
maxIterations	**5**	The number of iterations.
alpha	**40**	The rate of confidence.
lambda	**0.01**	The parameter of the regularization.
preferenceThreshold	**0**	Threshold used to define preference values. **0** is the only threshold supported so far.

Prediction

For a description of the input and output, refer to Recommendation Systems Usage Model.

At the prediction stage, the implicit ALS recommender has the following parameters:

Prediction Parameters for Implicit Alternating Least Squares Computaion (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Distributed Processing: Training

The distributed processing mode assumes that the data set is split in nblocks blocks across computation nodes.

Algorithm Parameters

At the training stage, implicit ALS recommender in the distributed processing mode has the following parameters:

Training Parameters for Implicit Alternating Least Squares Computaion (Distributed Processing)

Parameter	Default Value	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes • `step2Master` - the second step, performed on a master node • `step3Local` - the third step, performed on local nodes • `step4Local` - the fourth step, performed on local nodes
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`fastCSR`	Performance-oriented computation method for CSR numeric tables, the only method supported by the algorithm.
`nFactors`	**10**	The total number of factors.
`maxIterations`	**5**	The number of iterations.
`alpha`	**40**	The rate of confidence.
`lambda`	**0.01**	The parameter of the regularization.
`preferenceThreshold`	**0**	Threshold used to define preference values. **0** is the only threshold supported so far.

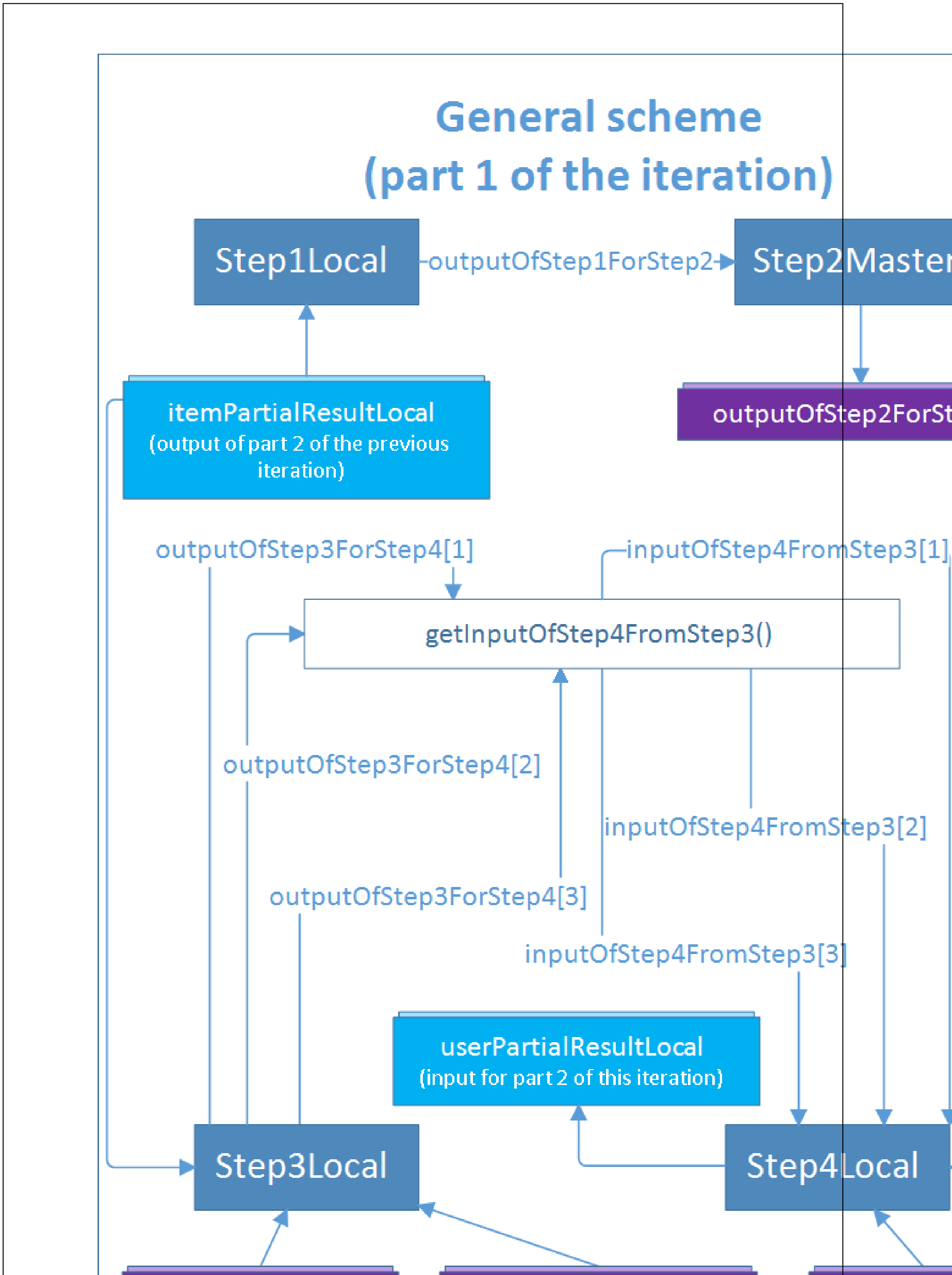
Computation Process

At each iteration, the implicit ALS training algorithm alternates between re-computing user factors (*X*) and item factors (*Y*). These computations split each iteration into the following parts:

1. Re-compute all user factors using the input data sets and item factors computed previously.
2. Re-compute all item factors using input data sets in the transposed format and item factors computed previously.

Each part includes four steps executed either on local nodes or on the master node, as explained below and illustrated by graphics for $\mathrm{nblocks} = 3$. The main loop of the implicit ALS training stage is executed on the master node.

Implicit Alternating Least Squares Computaion: Part 1



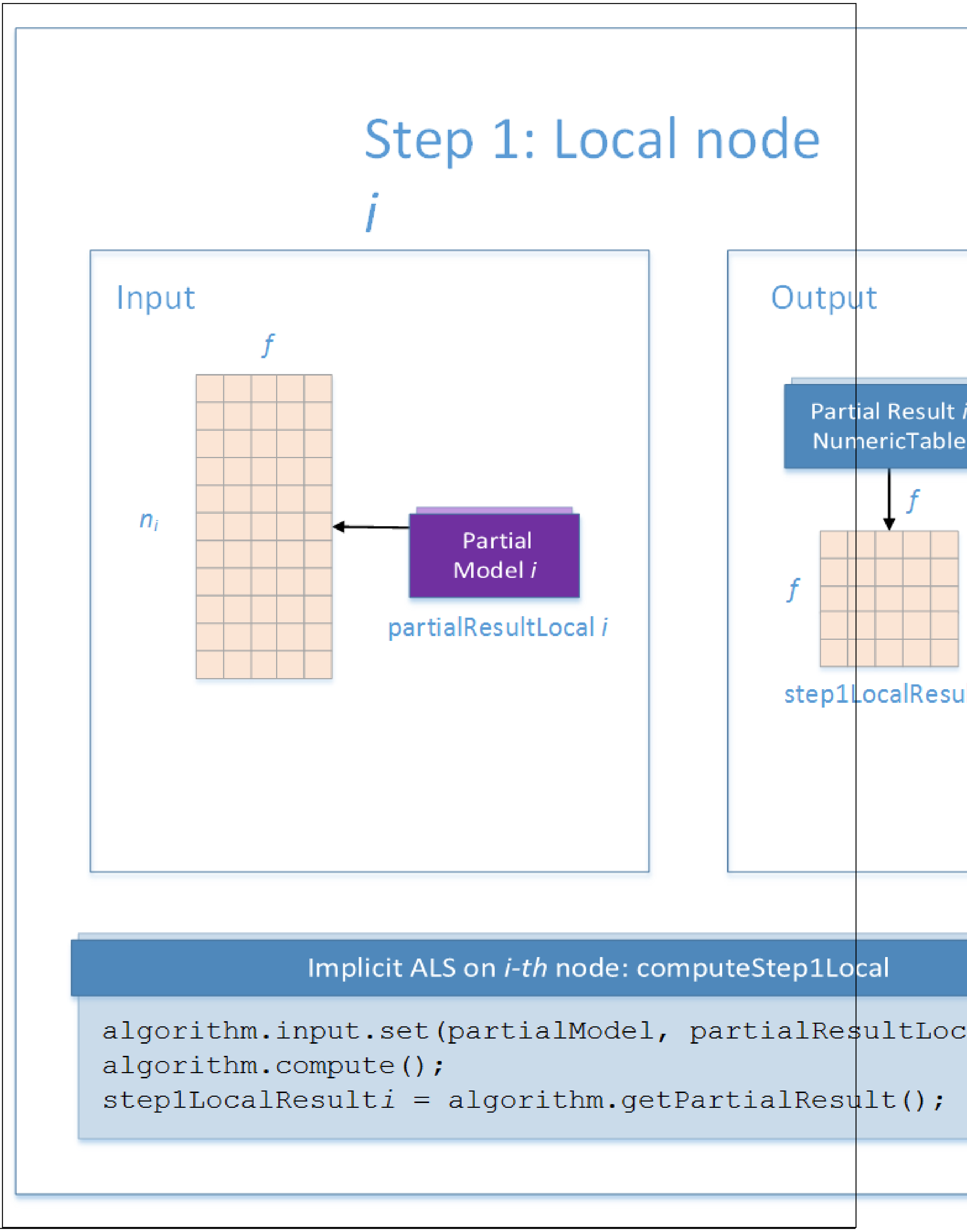
Step 1 - on Local Nodes

This step works with the matrix:

- Y^T in part 1 of the iteration
- *X* in part 2 of the iteration

Parts of this matrix are used as input partial models.

Training with Implicit Alternating Least Squares: Distributed Processing, Step 1 - on Local Nodes



```
algorithm.input.set(partialModel, partialResultLoc
algorithm.compute();
step1LocalResulti = algorithm.getPartialResult();
```

In this step, implicit ALS recommender training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 1)

Input ID	Input
`partialModel`	Partial model computed on the local node.

In this step, implicit ALS recommender training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

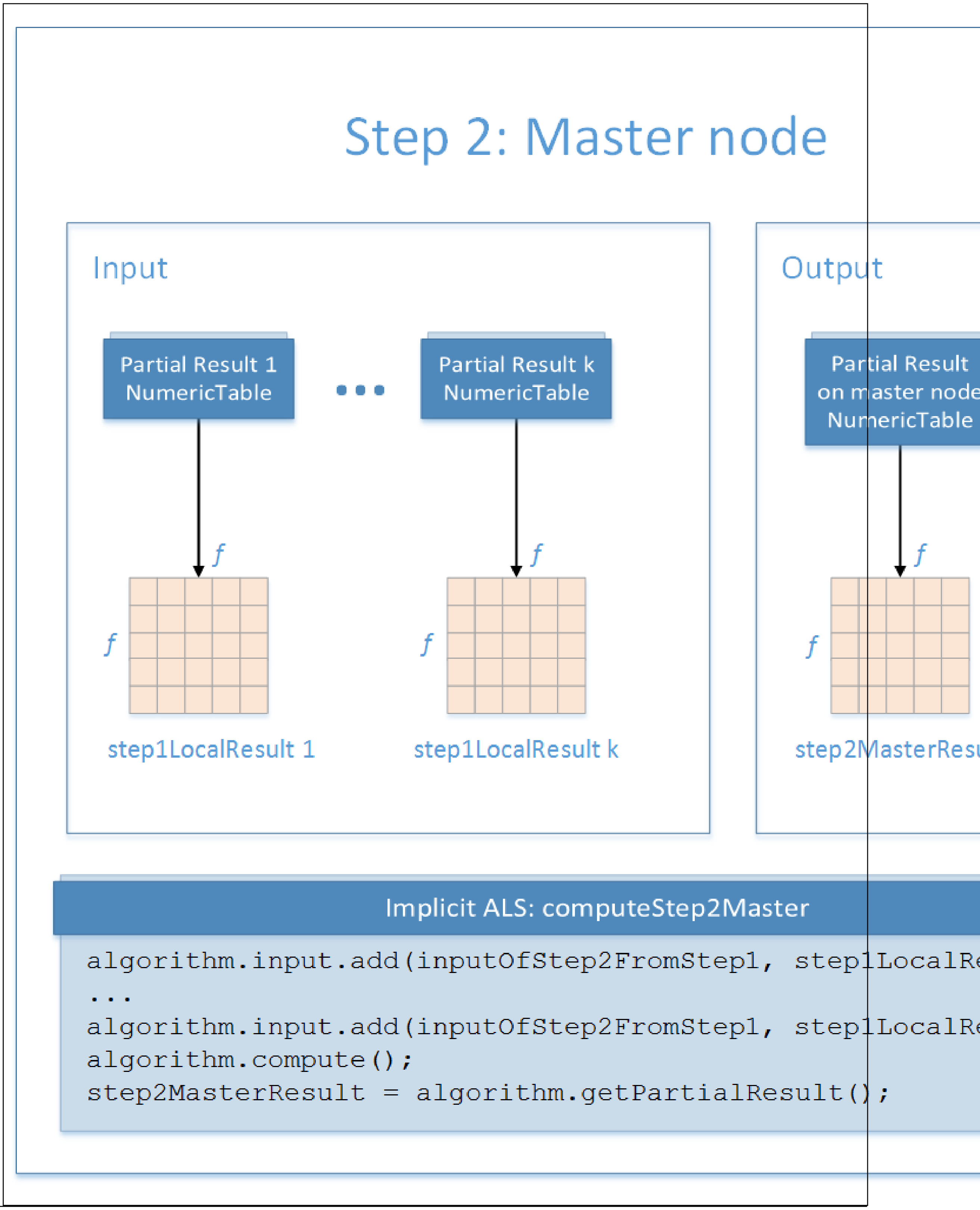
Output for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 1)

Result ID	Result
`outputOfStep1ForStep2`	Pointer to the $f \times f$ numeric table with the sum of numeric tables calculated in Step 1.

Step 2 - on Master Node

This step uses local partial results from Step 1 as input.

Training with Implicit Alternating Least Squares: Distributed Processing, Step 2 - on Master Node



```
algorithm.input.add(inputOfStep2FromStep1, step1LocalRe
...
algorithm.input.add(inputOfStep2FromStep1, step1LocalRe
algorithm.compute();
step2MasterResult = algorithm.getPartialResult();
```

In this step, implicit ALS recommender training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 2)

Input ID	Input
`inputOfStep2FromStep1`	A collection of numeric tables computed on local nodes in Step 1. **NOTE** The collection may contain objects of any class derived from `NumericTable` except the `PackedTriangularMatrix` class with the `lowerPackedTriangularMatrix` layout.

In this step, implicit ALS recommender training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 2)

Result ID	Result
`outputOfStep2ForStep4`	Pointer to the $f \times f$ numeric table with merged cross-products.

Step 3 - on Local Nodes

On each node *i*, this step uses results of the previous steps and requires that you provide two extra matrices Offset Table i and Input of Step 3 From Init i computed at the initialization stage of the algorithm.

The only element of the Offset Table i table refers to the:

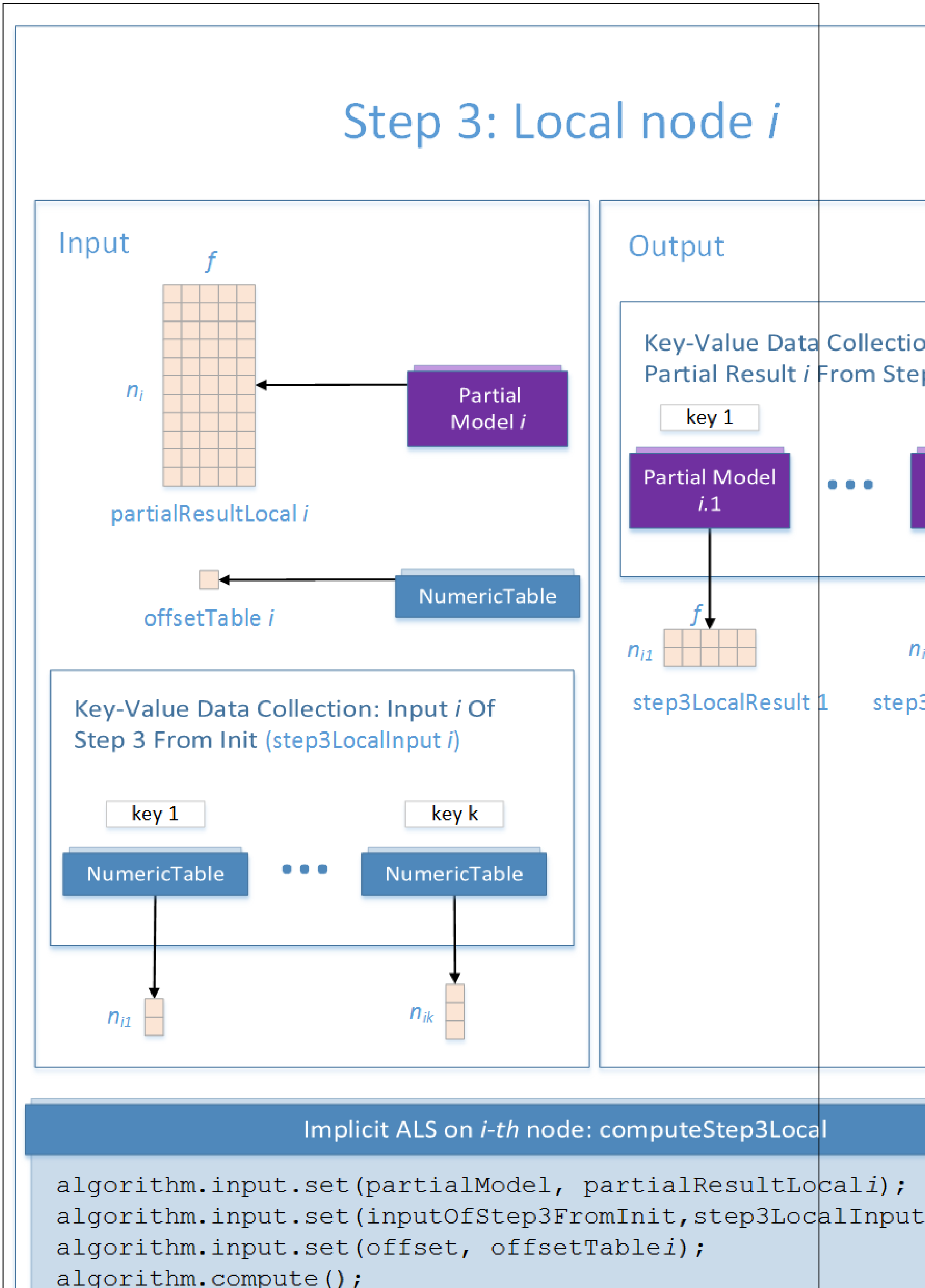
- *i*-th element of the `offsets` collection from the step 2 of the distributed initialization algorithm in part 1 of the iteration
- *i*-th element of the `offsets` collection from the step 1 of the distributed initialization algorithm in part 2 of the iteration

The Input Of Step 3 From Init is a key-value data collection that refers to the `outputOfInitForComputeStep3` output of the initialization stage:

- Output of the step 1 of the distributed initialization algorithm in part 1 of the iteration

- Output of the step 2 of the distributed initialization algorithm in part 2 of the iteration

Training with Implicit Alternating Least Squares: Distributed Processing, Step 3 - on Local Nodes



```
algorithm.input.set(partialModel, partialResultLocali);
algorithm.input.set(inputOfStep3FromInit,step3LocalInput
algorithm.input.set(offset, offsetTablei);
algorithm.compute();
```

In this step, implicit ALS recommender training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 3)

Input ID	Input
`partialModel`	Partial model computed on the local node.
`offset`	A numeric table of size $1imes1$ that holds the global index of the starting row of the input partial model. A part of the key-value data collection `offsets` computed at the initialization stage of the algorithm.

In this step, implicit ALS recommender training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 3)

Result ID	Result
`outputOfStep3ForStep4`	A key-value data collection that contains partial models to be used in Step 4. Each element of the collection contains an object of the `PartialModel` class.

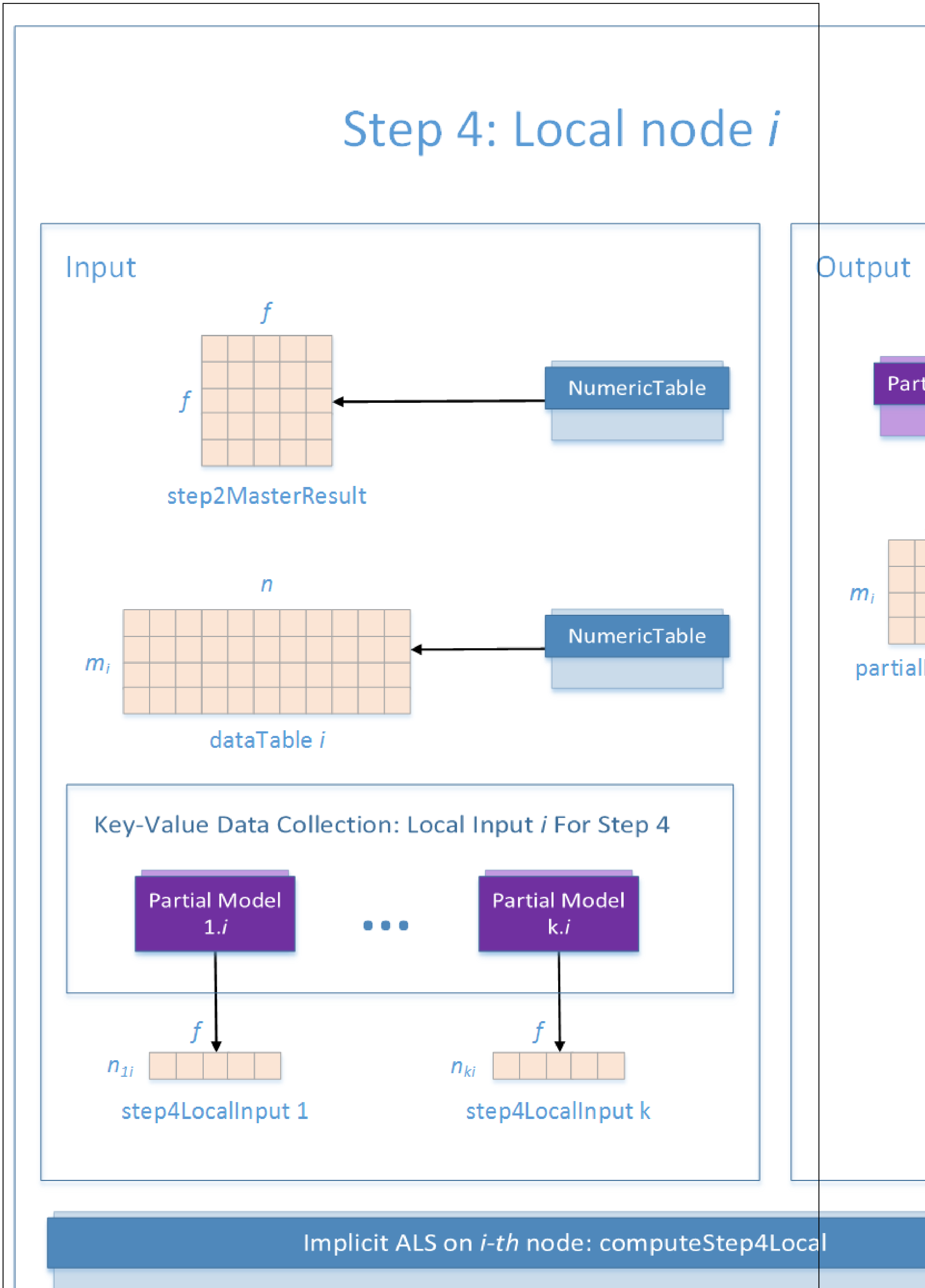
Step 4 - on Local Nodes

This step uses the results of the previous steps and parts of the following matrix in the transposed format:

- X in part 1 of the iteration
- Y^T in part 2 of the iteration

The results of the step are the re-computed parts of this matrix.

Training with Implicit Alternating Least Squares: Distributed Processing, Step 4 - on Local Nodes



In this step, implicit ALS recommender training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 4)

Input ID	Input
`partialModels`	A key-value data collection with partial models that contain user factors/item factors computed in Step 3. Each element of the collection contains an object of the `PartialModel` class.
`partialData`	Pointer to the CSR numeric table that holds the *i*-th part of the input data set, assuming that the data is divided by users/items.
`inputOfStep4FromStep2`	Pointer to the $f \times f$ numeric table computed in Step 2.

In this step, implicit ALS recommender training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 4)

Result ID	Result
`outputOfStep4ForStep1`	Pointer to the partial implicit ALS model that corresponds to the *i*-th data block. The partial model stores user factors/item factors.
`outputOfStep4ForStep3`	Pointer to the partial implicit ALS model that corresponds to the *i*-th data block. The partial model stores user factors/item factors.

Distributed Processing: Prediction of Ratings

The distributed processing mode assumes that the data set is split in `nblocks` blocks across computation nodes.

Algorithm Parameters

At the prediction stage, implicit ALS recommender in the distributed processing mode has the following parameters:

Prediction Parameters for Implicit Alternating Least Squares Computaion (Distributed Processing)

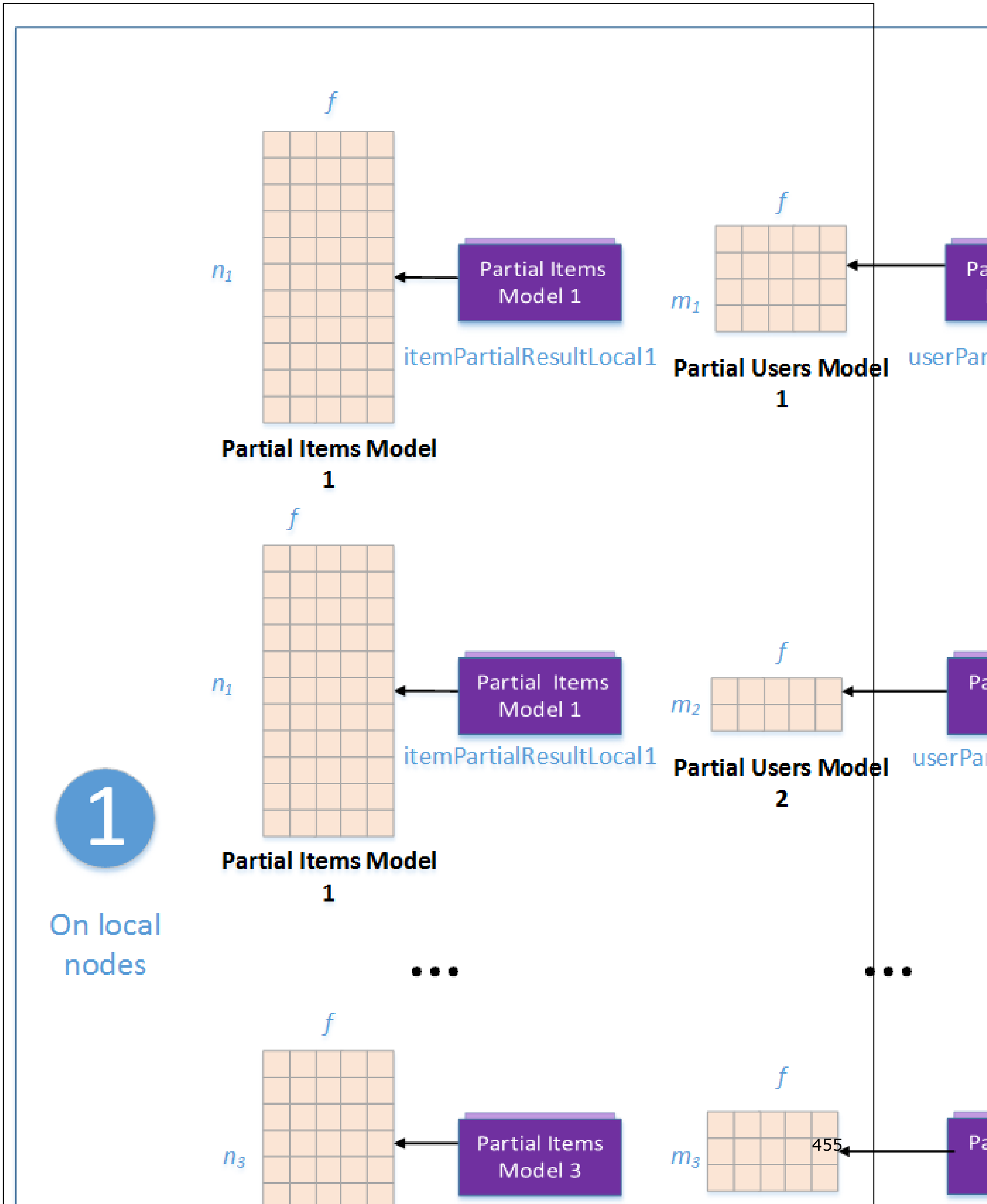
Parameter	Default Value	Description
`computeStep`	Not applicable	The parameter required to initialize the algorithm. Can be: • `step1Local` - the first step, performed on local nodes
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the algorithm.
`nFactors`	**10**	The total number of factors.

Use the one-step computation schema for implicit ALS recommender prediction in the distributed processing mode, as explained below and illustrated by the graphic for $\mathrm{nblocks} = 3$:

Step 1 - on Local Nodes

Prediction of rating uses partial models, which contain the parts of user factors $X_1, X_2, \ldots, X_{\mathrm{nblocks}}$ and item factors $Y_1, Y_2, \ldots, Y_{\mathrm{nblocks}}$ produced at the training stage. Each pair of partial models (X_i, Y_j) is used to compute a numeric table with ratings R_{ij} that correspond to the user factors and item factors from the input partial models.

Prediction with Implicit Alternating Least Squares: Distributed Processing, Step 1 - on Local Nodes



In this step, implicit ALS recommender-based prediction accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Input for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 1)

Input ID	Input
`usersPartialModel`	The partial model trained by the implicit ALS algorithm in the distributed processing mode. Stores user factors that correspond to the *i*-th data block.
`itemsPartialModel`	The partial model trained by the implicit ALS algorithm in the distributed processing mode. Stores item factors that correspond to the *j*-th data block.

In this step, implicit ALS recommender-based prediction calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Output for Implicit Alternating Least Squares Computaion (Distributed Processing, Step 1)

Result ID	Result
`prediction`	Pointer to the $m_i \times n_j$ numeric table with predicted ratings. **NOTE** By default this table is an object of the `HomogenNumericTable` class, but you can define it as an object of any class derived from `NumericTable` except `PackedTriangularMatrix`, `PackedSymmetricMatrix`, and `CSRNumericTable`.

Logistic Regression

Logistic regression is a method for modeling the relationships between one or more explanatory variables and a categorical variable by expressing the posterior statistical distribution of the categorical variable via linear functions on observed data. If the categorical variable is binary, taking only two values, “0” and “1”, the logistic regression is simple, otherwise, it is multinomial.

Details

Given n feature vectors of n p-dimensional feature vectors a vector of class labels $y = (y_1, \dots, y_n)$, where $y_i \in \{0, 1, \dots, K-1\}$ and *K* is the number of classes, describes the class to which the feature vector x_i belongs, the problem is to train a logistic regression model.

The logistic regression model is the set of vectors $\beta = \{\ \beta_0 = (\beta_{00} \dots \beta_{0p})\ ,\ ..\beta_{K-1} = (\beta_{K-10} \dots \beta_{K-1p})\}$ that gives the posterior probability

$$P\{y = k|x\} = \ p_k\left(x,\ \beta\right) = \ \frac{e^{f_k(x,\ \beta)}}{\sum_{i=0}^{K-1}\ e^{f_i(x,\ \beta)}},\ \text{where}\ f_k\left(x,\ \beta\right) = \ \beta_{k0} + \sum_{j=1}^{p} \beta_{kj} * x_j$$

for a given feature vector $x = (x_1, \dots, x_p)$ and class label $y \in \{0, 1, \dots, K-1\}$ for each $k = 0, \dots, K-1$. See [Hastie2009].

If the categorical variable is binary, the model is defined as a single vector $\beta_0 = (\beta_{00} \dots \beta_{0p})$ that determines the posterior probability

$$P\left\{y=1|x\right\}=\sigma\left(x,\beta\right)=\frac{1}{1+e^{-f(x,\beta)}}$$

$$P\left\{y=0|x\right\}=1-P\left\{y=1|x\right\}$$

Training Stage

Training procedure is an iterative algorithm which minimizes objective function

$$L\left(\beta\right)=-\frac{1}{n}\sum_{i=1}^{n}\mathrm{log}p_{y_i}\left(x_i,\beta\right)+\lambda_1\sum_{k=0}^{K-1}\|\beta_k\|_{L1}+\lambda_2\sum_{k=0}^{K-1}\|\beta_k\|_{L2}$$

where the first term is the negative log-likelihood of conditional *Y* given *X*, and the latter terms are regularization ones that penalize the complexity of the model (large β values), λ_1 and λ_2 are non-negative regularization parameters applied to L1 and L2 norm of vectors in β.

For more details, see [Hastie2009], [Bishop2006].

For the objective function minimization the library supports the iterative algorithms defined by the interface of daal::algorithms::iterative_solver. See Iterative Solver.

Prediction Stage

Given logistic regression model and vectors $x_1, \ldots, x_r$, the problem is to calculate the responses for those vectors, and their probabilities and logarithms of probabilities if required. The computation is based on formula (1) in multinomial case and on formula (2) in binary case.

Usage of Training Alternative

To build a Logistic Regression model using methods of the Model Builder class of Logistic Regression, complete the following steps:

- Create a Logistic Regression model builder using a constructor with the required number of responses and features.
- Use the `setBeta` method to add the set of pre-calculated coefficients to the model. Specify random access iterators to the first and the last element of the set of coefficients [ISO/IEC 14882:2011 §24.2.7]_.

> **NOTE** If your set of coefficients does not contain an intercept, interceptFlag is automatically set to `False`, and to `True`, otherwise.

- Use the `getModel` method to get the trained Logistic Regression model.
- Use the `getStatus` method to check the status of the model building process. If `DAAL_NOTHROW_EXCEPTIONS` macros is defined, the status report contains the list of errors that describe the problems API encountered (in case of API runtime failure).

> **NOTE** If after calling the `getModel` method you use the `setBeta` method to update coefficients, the initial model will be automatically updated with the new β coefficients.

Examples

C++ (CPU)

- log_reg_model_builder.cpp

Java*

NOTE There is no support for Java on GPU.

- LogRegModelBuilder.java

Python*

- log_reg_model_builder.py

Batch Processing

Logistic regression algorithm follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Classification Usage Model.

In addition to the parameters of classifier described in Classification Usage Model, the logistic regression batch training algorithm has the following parameters:

Training Parameters for Logistic Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the logistic regression. The only training method supported so far is the default dense method.
`nClasses`	Not applicable	The number of classes. A required parameter.
`interceptFlag`	`True`	A flag that indicates a need to compute θ_j
`penaltyL1`	**0**	L1 regularization coefficient **NOTE** L1 regularization is not supported on GPU.
`penaltyL2`	**0**	L2 regularization coefficient
`optimizationSolver`	SGD solver	All iterative solvers are available as optimization procedures to use at the training stage: • SGD (Stochastic Gradient Descent Algorithm) • ADAGRAD (Adaptive Subgradient Method) • LBFGS (Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm) • SAGA (Stochastic Average Gradient Accelerated Method)

Prediction

For a description of the input, refer to Classification Usage Model.

At the prediction stage logistic regression batch algorithm has the following parameters:

Prediction Parameters for Logistic Regression (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by logistic regression. The only prediction method supported so far is the default dense method.
`nClasses`	Not applicable	The number of classes. A required parameter.
`resultsToCompute`	`computeClassesLabels`	The 64-bit integer flag that specifies which extra characteristics of the logistic regression to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: • `computeClassesLabels` for **prediction** • `computeClassesProbabilities` for **probabilities** • `computeClassesLogProbabilities` for **logProbabilities**

Output

In addition to classifier output, logistic regression prediction calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm.

Prediction Output for Logistic Regression (Batch Processing)

Result ID	Result
`probabilities`	A numeric table of size $n \times nClasses$ containing probabilities of classes computed when `computeClassesProbabilities` option is enabled.
`logProbabilities`	A numeric table of size $n \times nClasses$ containing logarithms of classes' probabilities computed when `computeClassesLogProbabilities` option is enabled.

NOTE Note that:

- If **resultsToCompute** does not contain **computeClassesLabels**, the **prediction** table is **NULL**.
- If **resultsToCompute** does not contain **computeClassesProbabilities**, the **probabilities** table is **NULL**.
- If **resultsToCompute** does not contain **computeClassesLogProbabilities**, the **logProbabilities** table is **NULL**.
- By default, each numeric table of this result is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable` except for `PackedSymmetricMatrix` and `PackedTriangularMatrix`.

Examples

C++ (CPU)

Batch Processing:

- log_reg_dense_batch.cpp
- log_reg_binary_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- LogRegDenseBatch.java
- LogRegBinaryDenseBatch.java

Python* with DPC++ support

Batch Processing:

- log_reg_dense_batch.py
- log_reg_binary_dense_batch.py

Python*

Batch Processing:

- log_reg_dense_batch.py
- log_reg_binary_dense_batch.py

Naïve Bayes Classifier

Naïve Bayes is a set of simple and powerful classification methods often used for text classification, medical diagnosis, and other classification problems. In spite of their main assumption about independence between features, Naïve Bayes classifiers often work well when this assumption does not hold. An advantage of this method is that it requires only a small amount of training data to estimate model parameters.

Details

The library provides Multinomial Naïve Bayes classifier [Renie03].

Let *J* be the number of classes, indexed $0, 1, \ldots, J-1$. The integer-valued feature vector $x_i = (x_{11}, \ldots, x_{ip})$, $i = 1, \ldots, n$, contains scaled frequencies: the value of x_{ik} is the number of times the *k*-th feature is observed in the vector x_i (in terms of the document classification problem, x_{ik} is the number of occurrences of the word indexed *k* in the document x_i. For a given data set (a set of *n* documents), $(x_1, \ldots, x_n)$, the problem is to train a Naïve Bayes classifier.

Training Stage

The Training stage involves calculation of these parameters:

- $\log(\theta_{jk}) = \log\left(\frac{N_{jk}+\alpha_k}{N_j+\alpha}\right)$, where N_{jk} is the number of occurrences of the feature *k* in the class *j*, N_j is the total number of occurrences of all features in the class, the α_k (for example, $\alpha_k = 1$), and α is the sum of all α_k.
- $\log(\theta_j)$, where $p(\theta_j)$ is the prior class estimate.

Prediction Stage

Given a new feature vector x_i, the classifier determines the class the vector belongs to:

$$class\left(x_i\right) = \mathrm{argmax}_j \left(\log\left(p\left(\theta_j\right)\right) + \sum_k \log\left(\theta_{jk}\right)\right).$$

Computation

The following computation modes are available:

- Batch Processing
- Online Processing
- Distributed Processing

Examples

C++ (CPU)

Batch Processing:

- mn_naive_bayes_dense_batch.cpp
- mn_naive_bayes_csr_batch.cpp

Online Processing:

- mn_naive_bayes_dense_online.cpp
- mn_naive_bayes_csr_online.cpp

Distributed Processing:

- mn_naive_bayes_dense_distr.cpp
- mn_naive_bayes_csr_distr.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- MnNaiveBayesDenseBatch.java
- MnNaiveBayesCSRBatch.java

Online Processing:

- MnNaiveBayesDenseOnline.java
- MnNaiveBayesCSROnline.java

Distributed Processing:

- MnNaiveBayesDenseDistr.java
- MnNaiveBayesCSRDistr.java

Python*

Batch Processing:

- naive_bayes_batch.py

Online Processing:

- naive_bayes_streaming.py

Distributed Processing:

- naive_bayes_spmd.py

Performance Considerations

Training Stage

To get the best overall performance at the Naïve Bayes classifier training stage:

- If input data is homogeneous:
  - For the training data set, use a homogeneous numeric table of the same type as specified in the algorithmFPType class template parameter.
  - For class labels, use a homogeneous numeric table of type int.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

The training stage of the Naïve Bayes classifier algorithm is memory access bound in most cases. Therefore, use efficient data layout whenever possible.

Prediction Stage

To get the best overall performance at the Naïve Bayes classifier prediction stage:

- For the working data set, use a homogeneous numeric table of the same type as specified in the algorithmFPType class template parameter.
- For predicted labels, use a homogeneous numeric table of type int.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Batch Processing

Naïve Bayes classifier in the batch processing mode follows the general workflow described in Classification Usage Model.

Training

At the training stage, Naïve Bayes classifier has the following parameters:

Training Parameters for Naïve Bayes Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Available computation methods for the Naïve Bayes classifier: • `defaultDense` - default performance-oriented method • `fastCSR` - performance-oriented method for CSR numeric tables
`nClasses`	Not applicable	The number of classes. A required parameter.
`priorClassEstimates`	$1/\mathrm{nClasses}$	Vector of size `nClasses` that contains prior class estimates. The default value applies to each vector element.
`alpha`	**1**	Vector of size *p* that contains the imagined occurrences of features. The default value applies to each vector element.

Prediction

At the prediction stage, Naïve Bayes classifier has the following parameters:

Prediction Parameters for Naïve Bayes Classifier (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.
nClasses	Not applicable	The number of classes. A required parameter.

Online Processing

You can use the Naïve Bayes classifier algorithm in the online processing mode only at the training stage.

This computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \ldots, \text{nblocks}$.

Training

Naïve Bayes classifier training in the online processing mode follows the general workflow described in Classification Usage Model.

Naïve Bayes classifier in the online processing mode accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Naïve Bayes Classifier (Online Processing)

Input ID	Input
data	Pointer to the $n_i \times p$ numeric table that represents the current data block.
labels	Pointer to the $n_i \times 1$ numeric table with class labels associated with the current data block.

NOTE These tables can be objects of any class derived from NumericTable.

Naïve Bayes classifier in the online processing mode has the following parameters:

Training Parameters for Naïve Bayes Classifier (Online Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available computation methods for the Naïve Bayes classifier: • defaultDense - default performance-oriented method • fastCSR - performance-oriented method for CSR numeric tables
nClasses	Not applicable	The number of classes. A required parameter.

Parameter	Default Value	Description
priorClassEstimates	$1/\mathrm{nClasses}$	Vector of size nClasses that contains prior class estimates. The default value applies to each vector element.
alpha	**1**	Vector of size *p* that contains the imagined occurrences of features. The default value applies to each vector element.

For a description of the output, refer to Classification Usage Model.

Distributed Processing

You can use the Naïve Bayes classifier algorithm in the distributed processing mode only at the training stage.

This computation mode assumes that the data set is split in nblocks blocks across computation nodes.

Training

Algorithm Parameters

At the training stage, Naïve Bayes classifier in the distributed processing mode has the following parameters:

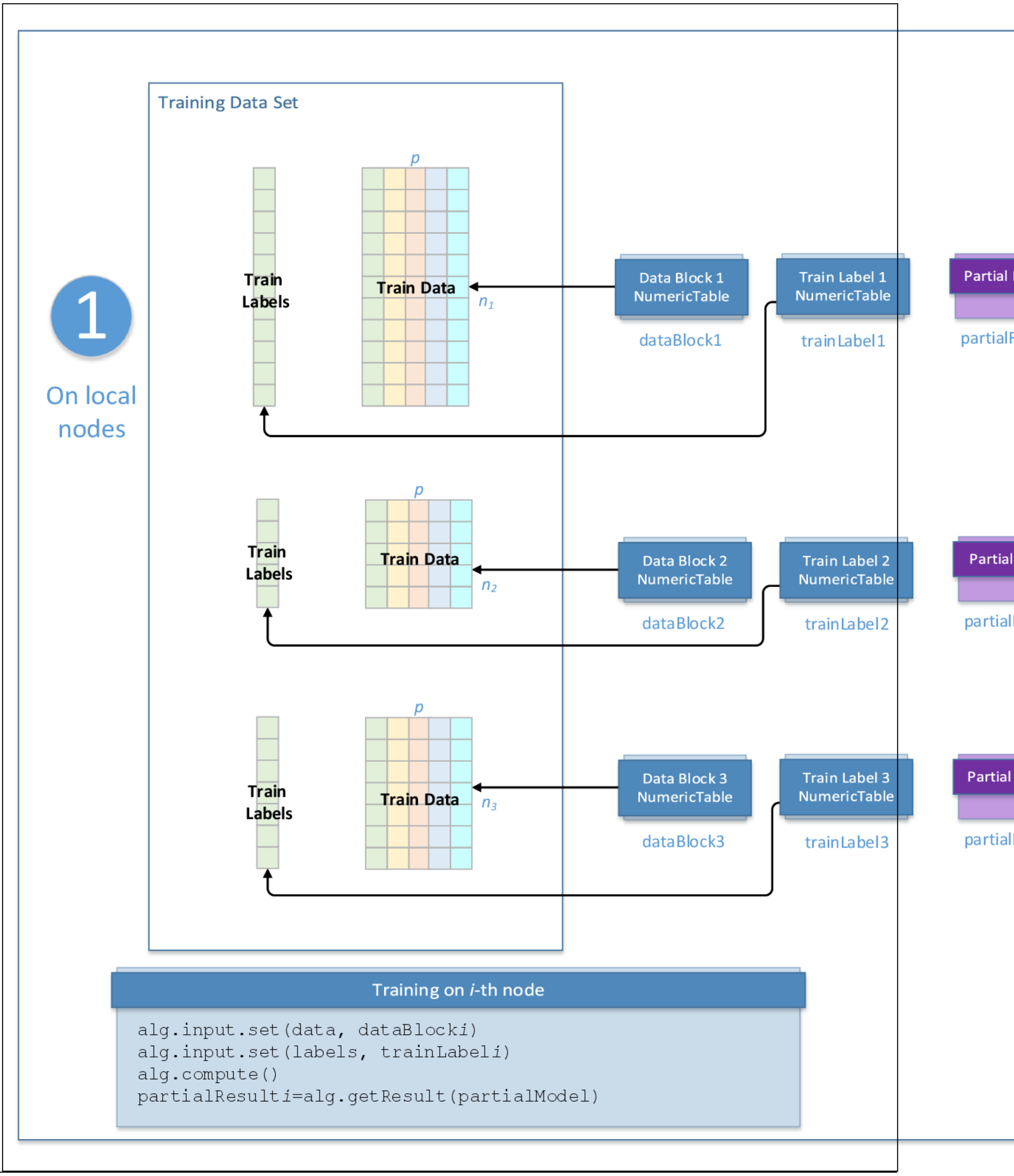
Training Parameters for Naïve Bayes Classifier (Distributed Processing)

Parameter	Default Valude	Description
computeStep	Not applicable	The parameter required to initialize the algorithm. Can be: • step1Local - the first step, performed on local nodes • step2Master - the second step, performed on a master node
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Available computation methods for the Naïve Bayes classifier: • defaultDense - default performance-oriented method • fastCSR - performance-oriented method for CSR numeric tables
nClasses	Not applicable	The number of classes. A required parameter.
priorClassEstimates	$1/\mathrm{nClasses}$	Vector of size nClasses that contains prior class estimates. The default value applies to each vector element.
alpha	**1**	Vector of size *p* that contains the imagined occurrences of features. The default value applies to each vector element.

Use the two-step computation schema for Naïve Bayes classifier training in the distributed processing mode, as illustrated below:

Step 1 - on Local Nodes

Training with Naïve Bayes Classifier: Distributed Processing, Step 1 - on Local Nodes



```
alg.input.set(data, dataBlocki)
alg.input.set(labels, trainLabeli)
alg.compute()
partialResulti=alg.getResult(partialModel)
```

In this step, Naïve Bayes classifier training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Naïve Bayes Classifier (Distributed Processing, Step 1)

Input ID	Input
`data`	Pointer to the $n_i \times p$ numeric table that represents the current data block.
`labels`	Pointer to the $n_i \times 1$ numeric table with class labels associated with the current data block.

NOTE These tables can be objects of any class derived from `NumericTable`.

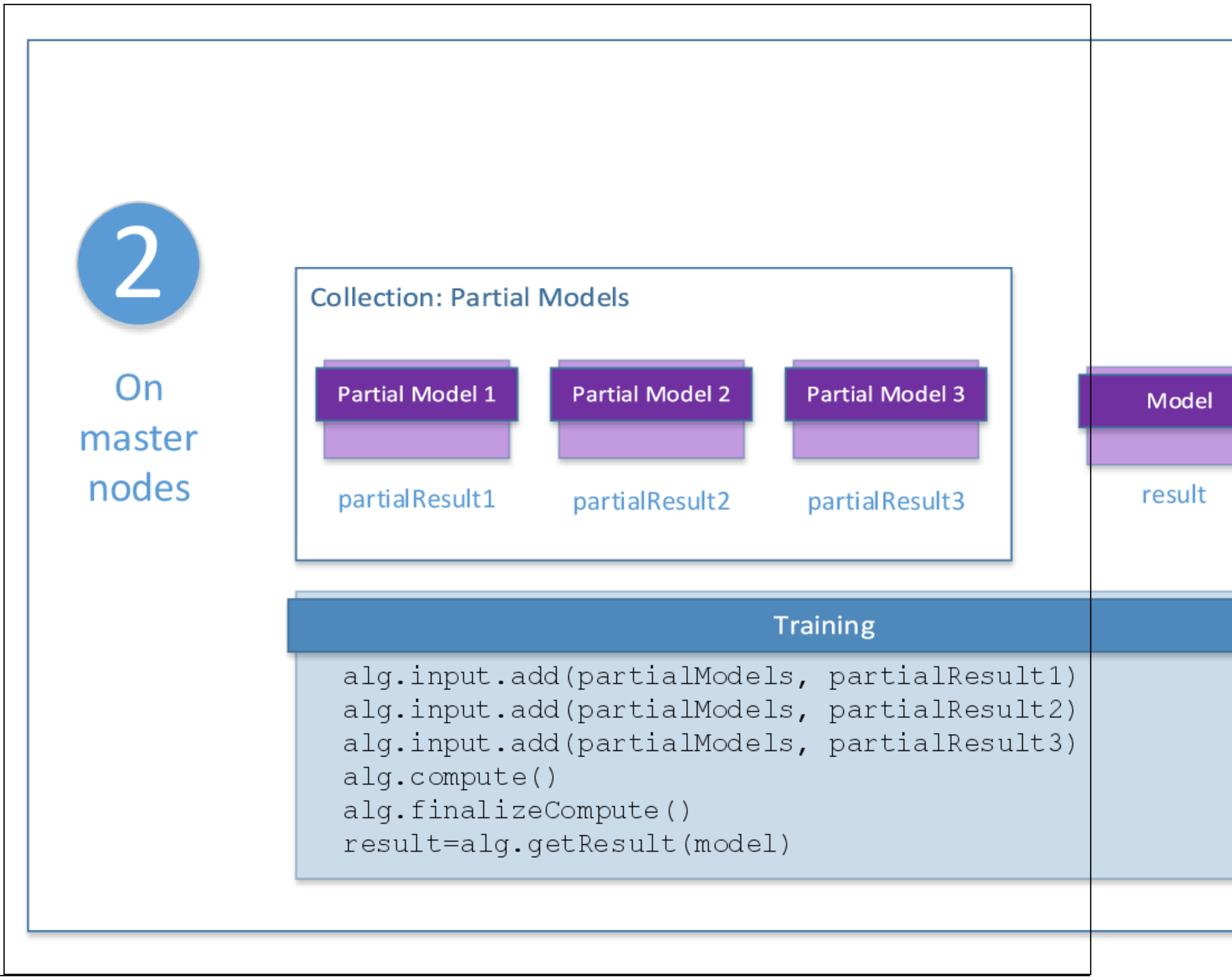
In this step, Naïve Bayes classifier training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Naïve Bayes Classifier (Distributed Processing, Step 1)

Result ID	Result
`partialModel`	Pointer to the partial Naïve Bayes classifier model that corresponds to the *i*-th data block. The result can only be an object of the `Model` class.

Step 2 - on Master Node

Trainin with Naïve Bayes Classifier: Distributed Processing, Step 2 - on Master Node



```
alg.input.add(partialModels, partialResult1)
alg.input.add(partialModels, partialResult2)
alg.input.add(partialModels, partialResult3)
alg.compute()
alg.finalizeCompute()
result=alg.getResult(model)
```

In this step, Naïve Bayes classifier training accepts the input described below. Pass the `Input ID` as a parameter to the methods that provide input for your algorithm. For more details, see Algorithms.

Training Input for Naïve Bayes Classifier (Distributed Processing, Step 2)

Input ID	Input
`partialModels`	A collection of partial models computed on local nodes in Step 1. The collection contains objects of the `Model` class.

In this step, Naïve Bayes classifier training calculates the result described below. Pass the `Result ID` as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms.

Training Output for Naïve Bayes Classifier (Distributed Processing, Step 2)

Result ID	Result
`model`	Pointer to the Naïve Bayes classifier model being trained. The result can only be an object of the `Model` class.

Support Vector Machine Classifier

NOTE Support Vector Machine Classifier is also available with oneAPI interfaces:

- Support Vector Machine Classifier and Regression (SVM)

Support Vector Machine (SVM) is among popular classification algorithms. It belongs to a family of generalized linear classification problems. Because SVM covers binary classification problems only in the multi-class case, SVM must be used in conjunction with multi-class classifier methods. SVM is a binary classifier. For a multi-class case, use Multi-Class Classifier framework of the library.

Details

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p* and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in \{-1, 1\}$ describes the class to which the feature vector x_i belongs, the problem is to build a two-class Support Vector Machine (SVM) classifier.

Training Stage

oneDAL provides two methods to train the SVM model:

- Boser method [Boser92] - performance-oriented variant of Boser [Boser92] and Platt [Platt98] algorithms
- Thunder method [Wen2018]

The SVM model is trained to solve the quadratic optimization problem

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$

with $0 \leq \alpha_i \leq C$, $i = 1, \ldots, n$, $y^T \alpha = 0$, where *e* is the vector of ones, *C* is the upper bound of the coordinates of the vector α, *Q* is a symmetric matrix of size $nimesn$ with $Q_{ij} = y_i y_j K(x_i, x_j)$, and $K(x, y)$ is a kernel function.

Working subset of ɑ updated on each iteration of the algorithm is based on the Working Set Selection (WSS) 3 scheme [Fan05]. The scheme can be optimized using one of these techniques or both:

- **Cache**: the implementation can allocate a predefined amount of memory to store intermediate results of the kernel computation.
- **Shrinking**: the implementation can try to decrease the amount of kernel related computations (see [Joachims99]).

The solution of the problem defines the separating hyperplane and corresponding decision function $D(x) = \sum_k y_k \alpha_k K(x_k, x) + b$ where only those x_k that correspond to non-zero α_k appear in the sum, and *b* is a bias. Each non-zero α_k is called a classification coefficient and the corresponding x_k is called a support vector.

Prediction Stage

Given the SVM classifier and *r* feature vectors $x_1, \ldots, x_r$, the problem is to calculate the signed value of the decision function $D(x_i)$, $i = 1, \ldots, r$. The sign of the value defines the class of the feature vector, and the absolute value of the function is a multiple of the distance between the feature vector and separating hyperplane.

Usage of Training Alternative

To build a Support Vector Machine (SVM) Classifier model using methods of the Model Builder class of SVM Classifier, complete the following steps:

- Create an SVM Classifier model builder using a constructor with the required number of support vectors and features.
- In any sequence:
  - Use the `setSupportVectors`, `setClassificationCoefficients`, and `setSupportIndices` methods to add pre-calculated support vectors, classification coefficients, and support indices (optional), respectively, to the model. For each method specify random access iterators to the first and the last element of the corresponding set of values [ISO/IEC 14882:2011 § 24.2.7]_.
  - Use `setBias` to add a bias term to the model.
- Use the `getModel` method to get the trained SVM Classifier model.
- Use the `getStatus` method to check the status of the model building process. If `DAAL_NOTHROW_EXCEPTIONS` macros is defined, the status report contains the list of errors that describe the problems API encountered (in case of API runtime failure).

> **NOTE** If after calling the getModel method you use the `setBias`, `setSupportVectors`, `setClassificationCoefficients`, or `setSupportIndices` methods, coefficients, the initial model will be automatically updated with the new set of parameters.

Examples

C++ (CPU)

- svm_two_class_model_builder.cpp

Java*

> **NOTE** There is no support for Java on GPU.

- SVMTwoClassModelBuilder.java

Python*

- svm_two_class_model_builder.py

Batch Processing

SVM classifier follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Usage Model: Training and Prediction.

At the training stage, SVM classifier has the following parameters:

Training Parameters for Support Vector Machine Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the SVM classifier. Available methods for the training stage:

Parameter	Default Value	Description
		For CPU: • `defaultDense` – Boser method [Boser92] • `thunder` - Thunder method [Wen2018] For GPU: • `thunder` - Thunder method [Wen2018]
`nClasses`	**2**	The number of classes.
`C`	**1.0**	The upper bound in conditions of the quadratic optimization problem.
`accuracyThreshold`	**0.001**	The training accuracy.
`tau`	$1.0e - 6$	Tau parameter of the WSS scheme.
`maxIterations`	**1000000**	Maximal number of iterations for the algorithm.
`cacheSize`	**8000000**	The size of cache in bytes for storing values of the kernel matrix. A non-zero value enables use of a cache optimization technique.
`doShrinking`	`true`	A flag that enables use of a shrinking optimization technique. **NOTE** This parameter is only supported for `defaultDense` method.
`kernel`	Pointer to an object of the KernelIface class	The kernel function. By default, the algorithm uses a linear kernel.

Prediction

For a description of the input and output, refer to Usage Model: Training and Prediction.

At the prediction stage, SVM classifier has the following parameters:

Prediction Parameters for Support Vector Machine Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only prediction method supported by the algorithm.
`nClasses`	**2**	The number of classes.
`kernel`	Pointer to object of the `KernelIface` class	The kernel function. By default, the algorithm uses a linear kernel.

Examples

oneAPI DPC++

Batch Processing:

- dpc_svm_two_class_thunder_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_svm_two_class_smo_dense_batch.cpp
- cpp_svm_two_class_thunder_dense_batch.cpp

C++ (CPU)

Batch Processing:

- svm_two_class_boser_dense_batch.cpp
- svm_two_class_boser_csr_batch.cpp
- svm_two_class_thunder_dense_batch.cpp
- svm_two_class_thunder_csr_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- SVMTwoClassBoserDenseBatch.java
- SVMTwoClassBoserCSRBatch.java
- SVMTwoClassThunderDenseBatch.java
- SVMTwoClassThunderCSRBatch.java

Python* with DPC++ support

Batch Processing:

- svm_batch.py

Python*

Batch Processing:

- svm_batch.py

Performance Considerations

For the best performance of the SVM classifier, use homogeneous numeric tables if your input data set is homogeneous or SOA numeric tables otherwise.

Performance of the SVM algorithm greatly depends on the cache size cacheSize. Larger cache size typically results in greater performance. For the best SVM algorithm performance, use cacheSize equal to $n^2 \cdot \mathrm{sizeof(algorithmFPType)}$. However, avoid setting the cache size to a larger value than the number of bytes required to store n^2 data elements because the algorithm does not fully utilize the cache in this case.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Multi-class Classifier

While some classification algorithms naturally permit the use of more than two classes, some algorithms, such as Support Vector Machines (SVM), are by nature solving a two-class problem only. These two-class (or binary) classifiers can be turned into multi-class classifiers by using different strategies, such as One-Against-Rest or One-Against-One.

oneDAL implements a Multi-Class Classifier using the One-Against-One strategy.

Multi-class classifiers, such as SVM, are based on two-class classifiers, which are integral components of the models trained with the corresponding multi-class classifier algorithms.

Details

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p*, the number of classes *K*, and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in \{0, 1, \ldots, K-1\}$, the problem is to build a multi-class classifier using a two-class (binary) classifier, such as a two-class SVM.

Training Stage

The model is trained with the One-Against-One method that uses the binary classification described in [Hsu02] as follows: For each pair of classes (i, j), train a binary classifier, such as SVM. The total number of such binary classifiers is $\frac{K(K-1)}{2}$.

Prediction Stage

Given a new feature vector x_i, the classifier determines the class to which the vector belongs.

oneDAL provides two methods for class label prediction:

- Wu method. According to the algorithm 2 for computation of the class probabilities described in [Wu04]. The library returns the index of the class with the largest probability.
- Vote-based method. If the binary classifier predicts the feature vector to be in *i*-th class, the number of votes for the class i is increased by one, otherwise the vote is given to the j-th class. If two classes have equal numbers of votes, the class with the smallest index is selected.

Usage of Training Alternative

To build a Multi-class Classifier model using methods of the Model Builder class of Multi-class Classifier, complete the following steps:

- Create a Multi-class Classifier model builder using a constructor with the required number of features and classes.
- Use the `setTwoClassClassifierModel` method for each pair of classes to add the pre-trained two-class classifiers to the model. In the parameters to the method specify the classes' indices and the pointer to the pre-trained two-class classifier for this pair of classes. You need to do this for each pair of classes, because the One-Against-One strategy is used.
- Use the `getModel` method to get the trained Multi-class Classifier model.
- Use the `getStatus` method to check the status of the model building process. If `DAAL_NOTHROW_EXCEPTIONS` macros is defined, the status report contains the list of errors that describe the problems API encountered (in case of API runtime failure).

Examples

oneAPI C++

Batch Processing

- cpp_svm_two_class_thunder_dense_batch.cpp

C++ (CPU)

Batch Processing

- svm_multi_class_model_builder.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing

- SVMMultiClassModelBuilder.java

Python*

svm_multi_class_model_builder.py

Batch Processing

Multi-class classifier follows the general workflow described in Classification Usage Model.

Training

At the training stage, a multi-class classifier has the following parameters:

Training Parameters for Multi-class Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the multi-class classifier. The only training method supported so far is One-Against-One.
`training`	Pointer to an object of the SVM training class	Pointer to the training algorithm of the two-class classifier. By default, the SVM two-class classifier is used.
`nClasses`	Not applicable	The number of classes. A required parameter.

Prediction

At the prediction stage, a multi-class classifier has the following parameters:

Prediction Parameters for Multi-class Classifier (Batch Processing)

Parameter	Method	Default Value	Description
`algorithmFPType`	`defaultDense` or `voteBased`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`pmethod`	Not applicable	`defaultDense`	Available methods for multi-class classifier prediction stage:

Parameter	Method	Default Value	Description
			• defaultDense - the method described in [Wu04] • voteBased - the method based on the votes obtained from two-class classifiers.
tmethod	defaultDense or voteBased	**training::oneAgainstOne**	The computation method that was used to train the multi-class classifier model.
prediction	defaultDense or voteBased	Pointer to an object of the SVM prediction class	Pointer to the prediction algorithm of the two-class classifier. By default, the SVM two-class classifier is used.
nClasses	defaultDense or voteBased	Not applicable	The number of classes. A required parameter.
maxIterations	defaultDense	**100**	The maximal number of iterations for the algorithm.
accuracyThreshold	defaultDense	1.0e-12	The prediction accuracy.
resultsToEvaluate	voteBased	computeClassLabels	The 64-bit integer flag that specifies which extra characteristics of the decision function to compute. Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics: • computeClassLabels for **prediction** • computeDecisionFunction for **decisionFunction**

Output

In addition to classifier output, multiclass classifier calculates the result described below. Pass the Result ID as a parameter to the methods that access the result of your algorithm. For more details, see Algorithms.

Output for Multi-class Classifier (Batch Processing)

Result ID	Result
decisionFunction	A numeric table of size $n \times \frac{K(K-1)}{2}$ containing the results of the decision function computed for all binary models when the computeDecisionFunction option is enabled.

NOTE If **resultsToEvaluate** does not contain **computeDecisionFunction**, the result of **decisionFunction** table is **NULL**.
By default, each numeric table of this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable except for PackedSymmetricMatrix and PackedTriangularMatrix.

Examples

C++ (CPU)

Batch Processing:

- svm_multi_class_boser_csr_batch.cpp
- svm_multi_class_boser_dense_batch.cpp
- svm_multi_class_thunder_csr_batch.cpp
- svm_multi_class_thunder_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- SVMMultiClassBoserCSRBatch.java
- SVMMultiClassBoserDenseBatch.java
- SVMMultiClassThunderCSRBatch.java
- SVMMultiClassThunderDenseBatch.java

Python*

Batch Processing:

- svm_multiclass_batch.py

Boosting

Boosting is a set of algorithms intended to build a strong classifier from an ensemble of weighted weak learners by iterative re-weighting according to some accuracy measure for weak learners. A weak learner is a classification or regression algorithm that has only slightly better performance than random guessing. Weak learners are usually very simple and fast, and they focus on classification of very specific features.

Boosting algorithms include LogitBoost, BrownBoost, AdaBoost, and others. A Decision Stump classifier is one of the popular weak learners.

In oneDAL, a weak learner is:

- Classification algorithm for AdaBoost and BrownBoost
- Regression algorithm for LogitBoost

Weak learners support training of the boosting model for weighted datasets.

oneDAL boosting algorithms pass pointers to weak learner training and prediction objects through the parameters of boosting algorithms. Use the `getNumberOfWeakLearners()` method to determine the number of weak learners trained.

You can implement your own weak learners by deriving from the appropriate interface classes:

- Classification for AdaBoost and BrownBoost
- Regression for LogitBoost

> **NOTE** When defining your own weak learners to use with boosting classifiers, make sure the prediction component of your weak learner returns:
>
> - The number from $\{-1, 1\}$ in case of binary classification.
> - Class label from $\{0, ..., \mathrm{nClasses} - 1\}$ for `nClasses` > 2.
> - Some boosting algorithms like SAMME.R AdaBoost that require probabilities of classes. For description of each boosting algorithm, refer to a corresponding section in this document.

- AdaBoost Classifier

- AdaBoost Multiclass Classifier
- BrownBoost Classifier
- LogitBoost Classifier

AdaBoost Classifier

AdaBoost (short for “Adaptive Boosting”) is a popular boosting classification algorithm. AdaBoost algorithm performs well on a variety of data sets except some noisy data [Freund99].

AdaBoost is a binary classifier. For a multi-class case, use Multi-class Classifier framework of the library.

Details

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p* and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in K = \{-1, 1\}$ describes the class to which the feature vector x_i belongs, and a weak learner algorithm, the problem is to build an AdaBoost classifier.

Training Stage

The following scheme shows the major steps of the algorithm:

1. Initialize weights $D_1(i) = \frac{1}{n}$ for $i = 1, \ldots, n$.
2. For $t = 1, \ldots, T$:
   a. Train the weak learner $h_t(t) \in \{-1, 1\}$ using weights D_t.
   b. Choose a confidence value α_t.
   c. Update $D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t Y_i h_t(x_i))}{Z_t}$, where Z_t is a normalization factor.
3. Output the final hypothesis:

$$H(x_i) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x_i)\right)$$

Prediction Stage

Given the AdaBoost classifier and *r* feature vectors $x_1, \ldots, x_r$, the problem is to calculate the final class:

$$H(x_i) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x_i)\right)$$

Batch Processing

AdaBoost classifier follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Classification Usage Model.

At the training stage, an AdaBoost classifier has the following parameters:

Training Parameters for AdaBoost Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the AdaBoost classifier. The only training method supported so far is the Y. Freund's method.
`weakLearnerTraining`	Pointer to an object of the stump training class	Pointer to the training algorithm of the weak learner. By default, a stump weak learner is used.
`weakLearnerPrediction`	Pointer to an object of the stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.
`accuracyThreshold`	**0.01**	AdaBoost training accuracy.
`maxIterations`	**100**	The maximal number of iterations for the algorithm.

Prediction

For a description of the input and output, refer to Classification Usage Model.

At the prediction stage, an AdaBoost classifier has the following parameters:

Prediction Parameters for AdaBoost Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the AdaBoost classifier at the prediction stage.
`weakLearnerPrediction`	Pointer to an object of the stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.

Examples

C++ (CPU)

Batch Processing:

- adaboost_dense_batch.cpp

Python*

- adaboost_batch.py

AdaBoost Multiclass Classifier

AdaBoost (short for "Adaptive Boosting") is a popular boosting classification algorithm. The AdaBoost algorithm performs well on a variety of data sets except some noisy data ([Friedman98], [Zhu2005]). The library supports two methods for the algorithms:

- SAMME, or Stagewise Additive Modeling using a Multi-class Exponential loss function [Zhu2005]
- SAMME.R, which is a modification of SAMME method for Real-valued returned probabilities from weak learner

Details

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots x_n = (x_{n1}, \ldots, x_{np})$ of size *p*, a vector of class labels $y = (y_1, \ldots, y_n)$ where $y_i \in K = \{-1, 1\}$ in case of binary classification and $y_i \in K = \{0, \ldots, C-1\}$, where *C* is a number of classes, describes the class *t* the feature vector x_i belongs to, and h_t is a weak learner algorithm, the problem is to build an AdaBoost classifier.

Training Stage

SAMME method

The following scheme shows the major steps of the SAMME algorithm:

1. Initialize weights $D_1(i) = \frac{1}{n}$ for $i = 1, \ldots, n$
2. For $t = 1, \ldots, T$:
   - Train the weak learner $h_t(i)$ using weights D_t.
   - Choose a confidence value $\alpha_t = \log \frac{1-\mathrm{err}_t}{\mathrm{err}_t} + \log(C-1)$, where $\mathrm{err}_t = \frac{\sum_{i=1} nD_t(i) I(y_i \neq h_t(i))}{\sum_{i=1} nD_t(i)}$
   - Update $D_{t+1}(i) = \frac{D_t i \exp(-\alpha_t I(y_i \neq h_t(i)))}{Z_t}$, where Z_t is a normalization factor.
3. Output the final hypothesis:

$$H(x) = \underset{k}{\mathrm{argmax}} \sum_{t=1}^{T} \alpha_t I(h_t x = k)$$

NOTE SAMME algorithm in case of binary classification is equal to the AdaBoost algorithm from [Friedman98].

SAMME.R method

The following scheme shows the major steps of the SAMME.R algorithm:

1. Initialize weights $D_1(i) = \frac{1}{n}$ for $i = 1, \ldots, n$
2. For $t = 1, \ldots, T$:
   - Train the weak learner $h_t(i)$ using weights D_t.
   - Receive the weighed class probability estimates from weak learner:

$$p_k^t(x) = \mathrm{Prob}_w\{c = k | x\}, k = 0, \ldots, C-1$$

   - For $k = 0, \ldots, C-1$, set $s_k^t(x)$:

$$s_k^t(x) = (C-1)\left(\log p_k^t(x) - \frac{1}{C}\sum_{k=0}^{C-1}\log p_k^t(x)\right)$$

- For $i = 1, \ldots, n$, update $D_{t+1}(i)$:

$$D_{t+1}(i) = \frac{1}{Z_t}\exp\left(-\frac{C-1}{C}z_i^T \log p^t(x)\right)$$

where Z_t is a normalization factor, $z_i = (z_{i1}, \ldots, z_{iC})$, $z_{ik} = \begin{cases} 1, & k = y_i \\ -\frac{1}{K-1}, & k \neq y_i \end{cases}$

3. Output the final hypothesis:

$$H(x) = \underset{k}{\operatorname{argmax}} \sum_{t=1}^{T} s_k^t(x)$$

Prediction Stage

SAMME method

Given the AdaBoost classifier and *r* feature vectors $x_1, \ldots, x_r$, the problem is to calculate the final class *H(x)*:

$$H(x) = \underset{k}{\operatorname{argmax}} \sum_{t=1}^{T} \alpha_t I(h_t x = k)$$

SAMME.R method

Given the AdaBoost classifier and *r* feature vectors $x_1, \ldots, x_r$, the problem is to calculate the final class *H(x)*:

$$H(x) = \underset{k}{\operatorname{argmax}} \sum_{t=1}^{T} s_k^t(x)$$

where $s_k^t(x)$ is as defined above in Training Stage.

Batch Processing

AdaBoost classifier follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Classification Usage Model. At the training stage, an AdaBoost classifier has the following parameters:

Training Parameters for AdaBoost Multiclass Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.

Parameter	Default Value	Description
method	defaultDense	Available methods for computation of the AdaBoost algorithm: • samme - uses the classifier that returns labels as weak learner • sammeR - uses the classifier that returns probabilities of belonging to class as weak learner • defaultDense is equal to samme method
weakLearnerTraining	Pointer to an object of the classification stump training class	Pointer to the training algorithm of the weak learner. By default, a classification stump weak learner is used.
weakLearnerPrediction	Pointer to an object of the classification stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a classification stump weak learner is used.
accuracyThreshold	**0.01**	AdaBoost training accuracy.
maxIterations	**100**	The maximal number of iterations for the algorithm.
learningRate	**1.0**	Multiplier for each classifier to shrink its contribution.
nClasses	**2**	The number of classes.
resultsToCompute	**0**	The 64-bit integer flag that specifies which extra characteristics of AdaBoost to compute. Current version of the library only provides the following option: computeWeakLearnersErrors

Output

In addition to classifier output, AdaBoost calculates the result described below. Pass the Result ID as a parameter to the methods that access the result of your algorithm. For more details, see Algorithms.

Training Output for AdaBoost Multiclass Classifier (Batch Processing)

Result ID	Result
weakLearnersErrors	A numeric table $1 \times \mathrm{maxIterations}$ containing weak learner's classification errors computed when the computeWeakLearnersErrors option is on. **NOTE** By default, this result is an object of the HomogenNumericTable class, but you can define the result as an object of any class derived from NumericTable.

Prediction

For a description of the input and output, refer to Classification Usage Model. At the prediction stage, an AdaBoost classifier has the following parameters:

Prediction Parameters for AdaBoost Multiclass Classifier (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	Performance-oriented computation method, the only method supported by the AdaBoost classifier at the prediction stage.
weakLearnerPrediction	Pointer to an object of the classification stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a classification stump weak learner is used.
nClasses	**2**	The number of classes.

Examples

C++ (CPU)

Batch Processing:

- adaboost_samme_two_class_batch.cpp
- adaboost_sammer_two_class_batch.cpp
- adaboost_samme_multi_class_batch.cpp
- adaboost_sammer_multi_class_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- AdaBoostSammeTwoClassBatch.java
- AdaBoostSammerTwoClassBatch.java
- AdaBoostSammeMultiClassBatch.java
- AdaBoostSammerMultiClassBatch.java

BrownBoost Classifier

BrownBoost is a boosting classification algorithm. It is more robust to noisy data sets than other boosting classification algorithms [Freund99].

BrownBoost is a binary classifier. For a multi-class case, use Multi-class Classifier framework of the library.

Details

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p* and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in K = \{-1, 1\}$ describes the class to which the feature vector x_i belongs, and a weak learner algorithm, the problem is to build a two-class BrownBoost classifier.

Training Stage

The model is trained using the Freund method [Freund01] as follows:

1. Calculate $c = \mathrm{erfinv}^2(1 - \varepsilon)$, where:

- $\operatorname{erfinv}(x)$ is an inverse error function,
- ε is a target classification error of the algorithm defined as $\frac{1}{n}\sum_{i=1}^{n}|p(x_i)-y_i|$
- $p(x)=\operatorname{erf}\left(\frac{\sum_{i=1}^{M}\alpha_i h_i(x)}{\sqrt{c}}\right)$
- $\operatorname{erf}(x)$ is the error function,
- $h_i(x)$ is a hypothesis formulated by the *i*-th weak learner, $i=1,\ldots,M$,
- α_i is the weight of the hypothesis.

2. Set initial prediction values: $r_1(x,y)=0$.
3. Set "remaining timing": $s_1=c$.
4. Do for $i=1,2,\ldots$ until $s_{i+1}\le 0$

   a. With each feature vector and its label of positive weight, associate $W_i(x,y)=e^{\frac{-(r_i(x,y)+s_i)^2}{c}}$.

   b. Call the weak learner with the distribution defined by normalizing Lmath:**W_i(x, y)** to receive a hypothesis $h_i(x)$.

   c. Solve the differential equation

   $$\frac{dt}{d\alpha}=\gamma=\frac{\sum_{(x,y)}\exp(-\frac{1}{c}(r_i(x,y)+\alpha h_i(x)y+s_i-t)^2)h_i(x)y}{\sum_{(x,y)}\exp(-\frac{1}{c}(r_i(x,y)+\alpha h_i(x)y+s_i-t)^2)}$$

   with given boundary conditions $t=0$ and $\alpha=0$ to find $t_i=t^*>0$ and $\alpha_i=\alpha^*$ such that either $ERRORprocessingmath$ or $t^*=s_i$, where $ERRORprocessingmath$ is a given small constant needed to avoid degenerate cases.

   d. Update the prediction values: $r_{i+1}(x,y)=r_i(x,y)+\alpha_i h_i(x)y$.

   e. Update "remaining time": $s_{i+1}=s_i-t_i$.

   End do

The result of the model training is the array of *M* weak learners h_i.

Prediction Stage

Given the BrownBoost classifier and *r* feature vectors $x_1,\ldots,x_r$, the problem is to calculate the final classification confidence, a number from the interval $[-1,1]$, using the rule:

$$p(x)=\operatorname{erf}\left(\frac{\sum_{i=1}^{M}\alpha_i h_i(x)}{\sqrt{c}}\right)$$

Batch Processing

BrownBoost classifier follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Classification Usage Model.

At the training stage, a BrownBoost classifier has the following parameters:

Training Parameters for BrownBoost Classifier (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
method	defaultDense	The computation method used by the BrownBoost classifier. The only training method supported so far is the Y. Freund's method.
nClasses	**2**	The number of classes.
weakLearnerTraining	**DEPRECATED**: Pointer to an object of the weak learner training class **USE INSTEAD**: Pointer to an object of the classification stump training class	**DEPRECATED**: Pointer to the training algorithm of the weak learner. By default, a stump weak learner is used. **USE INSTEAD**: Pointer to the classifier training algorithm. Be default, a classification stump with gini split criterion is used.
weakLearnerPrediction	**DEPRECATED**: Pointer to an object of the weak learner prediction class **USE INSTEAD**: Pointer to an object of the classification stump prediction class	**DEPRECATED**: Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used. **USE INSTEAD**: Pointer to the classifier prediction algorithm. Be default, a classification stump with gini split criterion is used.
accuracyThreshold	**0.01**	BrownBoost training accuracy ε.
maxIterations	**100**	The maximal number of iterations for the BrownBoost algorithm.
newtonRaphsonAccuracyThreshold	$1.0e-3$	Accuracy threshold of the Newton-Raphson method used underneath the BrownBoost algorithm.
newtonRaphsonMaxIterations	**100**	The maximal number of Newton-Raphson iterations in the algorithm.
degenerateCasesThreshold	$1.0e-2$	The threshold used to avoid degenerate cases.

Prediction

For a description of the input and output, refer to Classification Usage Model.

At the prediction stage, a BrownBoost classifier has the following parameters:

Prediction Parameters for BrownBoost Classifier (Batch Processing)

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.

Parameter	Default Value	Description
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the BrownBoost classifier.
`nClasses`	**2**	The number of classes.
`weakLearnerPrediction`	**DEPRECATED**: Pointer to an object of the weak learner prediction class **USE INSTEAD**: Pointer to an object of the classification stump prediction class	**DEPRECATED**: Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used. **USE INSTEAD**: Pointer to the classifier prediction algorithm. Be default, a classification stump with gini split criterion is used.
`accuracyThreshold`	**0.01**	BrownBoost training accuracy ε.

Examples

C++ (CPU)

Batch Processing:

- brownboost_dense_batch.cpp

Java*

> **NOTE** There is no support for Java on GPU.

Batch Processing:

- BrownBoostDenseBatch.java

Python*

Batch Processing:

- brownboost_batch.py

LogitBoost Classifier

LogitBoost is a boosting classification algorithm. LogitBoost and AdaBoost are close to each other in the sense that both perform an additive logistic regression. The difference is that AdaBoost minimizes the exponential loss, whereas LogitBoost minimizes the logistic loss.

LogitBoost within oneDAL implements a multi-class classifier.

Details

Given *n* feature vectors $x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np})$ of size *p* and a vector of class labels $y = (y_1, \ldots, y_n)$, where $y_i \in K = \{0, \ldots, J-1\}$ describes the class to which the feature vector x_i belongs and *J* is the number of classes, the problem is to build a multi-class LogitBoost classifier.

Training Stage

The LogitBoost model is trained using the Friedman method [Friedman00].

Let $y_{i,j} = I\{x_i \in j\}$ is the indicator that the *i*-th feature vector belongs to class *j*. The scheme below, which uses the stump weak learner, shows the major steps of the algorithm:

```
1. Start with weights $w_{ij} = \frac{1}{n}$, $F_j(x) = 0$, $p_j(x) = \frac{1}{J}$, $i = 1, \ldots, n$, $j = 0, \ldots, J-1$.
2. For $m = 1, \ldots, M$:
   Do
       For $j = 1, \ldots, J$
       Do
           1. Compute working responses and weights in the j-th class:
              $w_{ij} = p_i(x_i)(1 - p_i(x_i)), w_{ij} = max(z_{ij}, \mathrm{Thr1})$
              $z_{ij} = \frac{(y_{ij} - p_i(x_i))}{w_{ij}}, z_{ij} = \min(\max(z_{ij}, -\mathrm{Thr2}), \mathrm{Thr2})$
           2. Fit the function $f_{mj}(x)$ by a weighted least-squares regression of $z_{ij}$ to $x_i$ with weights $w_{ij}$ using the stump-based approach.
       End do
       $f_{mj}(x) = \frac{J-1}{J}(f_{mj}(x) - \frac{1}{J}\sum_{k=1}^{J} f_{mk}(x))$
       $F_j(x) = F_j(x) + f_{mj}(x)$
       $p_j(x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}$
   End do
```

The result of the model training is a set of *M* stumps.

Prediction Stage

Given the LogitBoost classifier and *r* feature vectors $x_1, \ldots, x_r$, the problem is to calculate the labels $\underset{j}{\mathrm{argmax}} F_j(x)$ of the classes to which the feature vectors belong.

Batch Processing

LogitBoost classifier follows the general workflow described in Classification Usage Model.

Training

For a description of the input and output, refer to Classification Usage Model.

At the training stage, a LogitBoost classifier has the following parameters:

Training Parameters for LogitBoost Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
`method`	`defaultDense`	The computation method used by the LogitBoost classifier. The only training method supported so far is the Friedman method.
`weakLearnerTraining`	**DEPRECATED**: Pointer to an object of the stump training class. **USE INSTEAD**: Pointer to an object of the regression stump training class.	**DEPRECATED**: Pointer to the training algorithm of the weak learner. By default, a stump weak learner is used. **USE INSTEAD**: Pointer to the regression training algorithm. By default, a regression stump with mse split criterion is used.
`weakLearnerPrediction`	**DEPRECATED**: Pointer to an object of the stump prediction class. **USE INSTEAD**: Pointer to an object of the regression stump prediction class.	**DEPRECATED**: Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used. **USE INSTEAD**: Pointer to the regression prediction algorithm. By default, a regression stump with mse split criterion is used.
`accuracyThreshold`	**0.01**	LogitBoost training accuracy.
`maxIterations`	**100**	The maximal number of iterations for the LogitBoost algorithm.
`nClasses`	Not applicable	The number of classes, a required parameter.
`weightsDegenerateCasesThreshold`	$1e-10$	The threshold to avoid degenerate cases when calculating weights w_{ij}.
`responsesDegenerateCasesThreshold`	$1e-10$	The threshold to avoid degenerate cases when calculating responses z_{ij}.

Prediction

For a description of the input and output, refer to Classification Usage Model.

At the prediction stage, a LogitBoost classifier has the following parameters:

Prediction Parameters for LogitBoost Classifier (Batch Processing)

Parameter	Default Value	Description
`algorithmFPType`	`float`	The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.

Parameter	Default Value	Description
`method`	`defaultDense`	Performance-oriented computation method, the only method supported by the LogitBoost classifier at the prediction stage.
`weakLearnerPrediction`	**DEPRECATED**: Pointer to an object of the stump prediction class. **USE INSTEAD**: Pointer to an object of the regression stump prediction class.	**DEPRECATED**: Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used. **USE INSTEAD**: Pointer to the regression prediction algorithm. By default, a regression stump with mse split criterion is used.
`nClasses`	Not applicable	The number of classes, a required parameter.

NOTE The algorithm terminates if it achieves the specified accuracy or reaches the specified maximal number of iterations. To determine the actual number of iterations performed, call the `getNumberOfWeakLearners()` method of the `LogitBoostModel` class and divide it by `nClasses`.

Examples

C++ (CPU)

Batch Processing:

- logitboost_dense_batch.cpp

Java*

NOTE There is no support for Java on GPU.

Batch Processing:

- LogitBoostDenseBatch.java

Python*

Batch Processing:

- logitboost_batch.py

Services

Classes and utilities included in the Services component of the Intel® oneAPI Data Analytics Library (oneDAL) are subdivided into the following groups according to their purpose:

- Extracting Version Information
- Handling Errors
- Managing Memory
- Managing the Computational Environment
- Providing a Callback for the Host Application

Extracting Version Information

The Services component provides methods that enable you to extract information about the version of oneDAL. You can get the following information about the installed version of the library from the fields of the LibraryVersionInfo structure:

Field Name	Description
majorVersion	Major version of the library
minorVersion	Minor version of the library
updateVersion	Update version of the library
productStatus	Status of the library: alpha, beta, or product
build	Build number
name	Library name
processor	Processor optimization

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Examples

C++: services/library_version_info.cpp

Java*: services/LibraryVersionInfoExample.java

Handling Errors

oneDAL provides classes and methods to handle exceptions or errors that can occur during library operation.

The methods of the library return the following computation set status:

- Success - no errors detected
- Warning - recoverable errors detected
- Failure - unrecoverable errors detected

In oneDAL C++ interfaces, the base class for error handling is Status. If the execution of the library methods provided by the Algorithm or Data Management classes is unsuccessful, the Status object returned by the respective routines contains the list of errors and/or warnings extended with additional details about the error conditions. The class includes the list of the following methods for error processing:

- `ok()` - checks whether the Status object contains any unrecoverable errors.
- `add()` - adds information about the error, such as the error identifier or the pointer to the error.
- `getDescription()` - returns the detailed description of the errors contained in the object.
- `clear()` - removes information about the errors from the object.

The error class in oneDAL C++ interfaces is Error. This class contains an error message and details of the issue. For example, an Error object can store the number of the row in the NumericTable that caused the issue or a message that an SQL database generated to describe the reasons of an unsuccessful query. A single Error object can store the error description and an arbitrary number of details of various types: integer or double values or strings.

The class includes the list of the following methods for error processing:

- `id()` - returns the identifier of the error.
- `setId()` - sets the identifier of the error.
- `description()` - returns the detailed description of the error.

- add[Int|Double|String]Detail() adds data type-based details to the error.
- create() - creates an instance of the Error class with the given set of arguments.

By default, the compute() method of the library algorithms throws run-time exception when error is detected. To prevent throwing any exceptions, call the computeNoThrow() method.

Service methods of the algorithms, such as setResult() and setPartialResult(), do not throw exceptions and return the status of the respective operation.

The methods of the Data Management classes do not throw exceptions and return the status of the respective operation.

oneDAL Java* interfaces handle errors by throwing Java exceptions.

Examples

C++:

- error_handling/error_handling_nothrow.cpp
- error_handling/error_handling_throw.cpp

Java*: error_handling/ErrorHandling.java

Managing Memory

To improve performance of your application that calls oneDAL, align your arrays on 64-byte boundaries and ensure that the leading dimensions of the arrays are divisible by 64. For that purpose oneDAL provides daal_malloc() and daal_free() functions to allocate and deallocate memory.

To allocate memory, call daal_malloc() with the specified size of the buffer to be allocated and the alignment of the buffer, which must be a power of 2. If the specified alignment is not a power of 2, the library uses the 32-byte alignment.

To deallocate memory allocated earlier by the daal_malloc() function, call the daal_free() function and set a pointer to the buffer to be freed.

Managing the Computational Environment

oneDAL provides the Environment class to manage settings of the computational environment in which the application that uses the library runs. The methods of this class enable you to specify the number of threads for the application to use or to check the type of the processor running the application. The Environment class is a singleton, which ensures that only one instance of the class is available in the oneDAL based application. To access the instance of the class, call the getInstance() method which returns a pointer to the instance. Once you get the instance of the class, you can use it for multiple purposes:

- Detect the processor type. To do this, call the getCpuId() method.
- Enable dispatching for new Intel® Architecture Processors. To do this, call the enableInstructionsSet() method. For example, to select the version for Intel® Xeon Phi™ processors based on Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support of AVX512_4FMAPS and AVX512_4VNNIW instruction groups, pass the avx512_mic_e1 parameter to the method.
- Restrict dispatching to the required code path. To do this, call the setCpuId() method.
- Detect and modify the number of threads used by the oneDAL based application. To do this, call the getNumberOfThreads() or setNumberOfThreads() method, respectively.
- Specify the single-threaded of multi-threaded mode for oneDAL on Windows. To do this, call to the setDynamicLibraryThreadingTypeOnWindows() method.
- Enable thread pinning. To do this, call the enableThreadPinning() method. This method performs binding of the threads that are used to parallelize algorithms of the library to physical processing units for possible performance improvement. Improper use of the method can result in degradation of the application performance depending on the system (machine) topology, application, and operating system. By default, the method is disabled.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Examples

C++: set_number_of_threads/set_number_of_threads.cpp

Java*: set_number_of_threads/SetNumberOfThreads.java

Providing a Callback for the Host Application

oneDAL provides a possibility for the host application to register a callback interface to be called by the library, e.g. for the purposes of computation interruption. It is done by means of an abstract interface for the host application of the library HostAppIface. In order to use it, the application should define an instance of the class derived from the abstract interface and set its pointer to an instance of Algorithm class.

Following methods of the Algorithm class are used:

Algorithm class methods

Name	Description
`setHostApp(const services::HostAppIfacePtr& pHost)`	Set pHost as the callback interface
`hostApp()`	Get current value of the callback interface set on the Algorithm

HostAppIface class includes following methods:

HostAppIface class Methods

Name	Description
`isCancelled()`	Enables computation cancelling. The method is called by the owning algorithm when computation is in progress. If the method returns true then computation stops and returns ErrorUserCancelled status. Since the method can be called from parallel threads when running with oneDAL threaded version, it is application responsibility to make its implementation thread-safe. It is not recommended for this method to throw exceptions.

Currently HostAppIface is supported in C++ only, cancelling is available with limited number of algorithms as follows: decision forest, gradient boosted trees.

Bibliography

For more information about algorithms implemented in oneDAL, refer to the following publications:

Adams2003

Adams, Robert A., and John JF Fournier. Sobolev spaces. Vol. 140. Elsevier, 2003

Agrawal94

Rakesh Agrawal, Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules*. Proceedings of the 20th VLDB Conference Santiago, Chile, 1994.

Arthur2007

Arthur, D., Vassilvitskii, S. *k-means++: The Advantages of Careful Seeding*. Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007, pp. 1027-1035. Available from http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf.

Bahmani2012

B. Bahmani, B. Moseley, A. Vattani, R. Kumar, S. Vassilvitskii. *Scalable K-means++*. Proceedings of the VLDB Endowment, 2012. Available from http://vldb.org/pvldb/vol5/p622_bahmanbahmani_vldb2012.pdf.

Ben2005

Ben-Gal I. Outlier detection. In: Maimon O. and Rockach L. (Eds.) Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers", Kluwer Academic Publishers, 2005, ISBN 0-387-24435-2.

Bentley80

J. L. Bentley. Multidimensional Divide and Conquer. Communications of the ACM, 23(4):214–229, 1980.

Billor2000

Nedret Billor, Ali S. Hadib, and Paul F. Velleman. BACON: blocked adaptive computationally efficient outlier nominators. Computational Statistics & Data Analysis, 34, 279-298, 2000.

Bishop2006

Christopher M. Bishop. *Pattern Recognition and Machine Learning*, p.198, Computational Statistics & Data Analysis, 34, 279-298, 2000. Springer Science+Business Media, LLC, ISBN-10: 0-387-31073-8, 2006.

Boser92

B. E. Boser, I. Guyon, and V. Vapnik. *A training algorithm for optimal marginclassifiers.*. Proceedings of the Fifth Annual Workshop on Computational Learning Theory, pp: 144–152, ACM Press, 1992.

Breiman2001

Leo Breiman. *Random Forests*. Machine Learning, Volume 45 Issue 1, pp. 5-32, 2001.

Breiman84

Leo Breiman, Jerome H. Friedman, Richard A. Olshen, Charles J. Stone. *Classification and Regression Trees*. Chapman & Hall, 1984.

Bro07

Bro, R.; Acar, E.; Kolda, T.. *Resolving the sign ambiguity in the singular value decomposition*. SANDIA Report, SAND2007-6422, Unlimited Release, October, 2007.

Byrd2015

R. H. Byrd, S. L. Hansen, Jorge Nocedal, Y. Singer. *A Stochastic Quasi-Newton Method for Large-Scale Optimization*, 2015. arXiv:1401.7020v2 [math.OC]. Available from http://arxiv.org/abs/1401.7020v2.

Chen2016

T. Chen, C. Guestrin. *XGBoost: A Scalable Tree Boosting System*, KDD '16 Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Carletti2021

Carletti, Vincenzo, et al. *Parallel Subgraph Isomorphism on Multi-core Architectures: A Comparison of Four Strategies Based on Tree Search.* Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR). Springer, Cham, 2021.

Defazio2014

Defazio, Aaron, Francis Bach, and Simon Lacoste-Julien. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. Advances in neural information processing systems. 2014.

Demmel90

J. W. Demmel and W. Kahan. *Accurate singular values of bidiagonal matrices*. SIAM J. Sci. Stat. Comput., 11 (1990), pp. 873-912.

Dempster77

A.P.Dempster, N.M. Laird, and D.B. Rubin. *Maximum-likelihood from incomplete data via the em algorithm*. J. Royal Statist. Soc. Ser. B., 39, 1977.

Duchi2011

Elad Hazan, John Duchi, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. The Journal of Machine Learning Research, 12:21212159, 2011.

Ester96

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise.. In Proceedings of the 2nd ACM International Conference on Knowledge Discovery and Data Mining (KDD). 226-231, 1996.

Fan05

Rong-En Fan, Pai-Hsuen Chen, Chih-Jen Lin. *Working Set Selection Using Second Order Information for Training Support Vector Machines.*. Journal of Machine Learning Research 6 (2005), pp: 1889–1918.

Fleischer2008

Rudolf Fleischer, Jinhui Xu. Algorithmic Aspects in Information and Management. 4th International conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings, Springer.

Freund99

Yoav Freund, Robert E. Schapire. *Additive Logistic regression: a statistical view of boosting*. Journal of Japanese Society for Artificial Intelligence (14(5)), 771-780, 1999.

Friedman98

Friedman, Jerome H., Trevor J. Hastie and Robert Tibshirani. *Additive Logistic Regression: a Statistical View of Boosting.*. 1998.

Friedman00

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive Logistic regression: a statistical view of boosting. The Annals of Statistics, 28(2), pp: 337-407, 2000.

Friedman2010

Friedman, Jerome, Trevor Hastie, and Rob Tibshirani. *Regularization paths for generalized linear models via coordinate descent.*. Journal of statistical software 33.1 (2010): 1.

Friedman2017

Jerome Friedman, Trevor Hastie, Robert Tibshirani. 2017. *The Elements of Statistical Learning Data Mining, Inference, and Prediction.* Springer.

Freund01

Yoav Freund. An adaptive version of the boost by majority algorithm. Machine Learning (43), pp. 293-318, 2001.

Gross2014

1. Gross, J. Yellen, P. Zhang, Handbook of Graph Theory, Second Edition, 2014.

Hastie2009

Trevor Hastie, Robert Tibshirani, Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second Edition (Springer Series in Statistics), Springer, 2009. Corr. 7th printing 2013 edition (December 23, 2011).

Hoerl70

Arthur E. Hoerl and Robert W. Kennard. *Ridge Regression: Biased Estimation for Nonorthogonal Problems*. Technometrics, Vol. 12, No. 1 (Feb., 1970), pp. 55-67.

Hsu02

Chih-Wei Hsu and Chih-Jen Lin. *A Comparison of Methods for Multiclass Support Vector Machines*. IEEE Transactions on Neural Networks, Vol. 13, No. 2, pp: 415-425, 2002.

Hu2008

Yifan Hu, Yehuda Koren, Chris Volinsky. Collaborative Filtering for Implicit Feedback Datasets. ICDM'08. Eighth IEEE International Conference, 2008.

James2013

Gareth James, Daniela Witten, Trevor Hastie, and Rob Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer Series in Statistics, Springer, 2013 (Corrected at 6th printing 2015).

Joachims99

Thorsten Joachims. *Making Large-Scale SVM Learning Practical*. Advances in Kernel Methods - Support Vector Learning, B. Schölkopf, C. Burges, and A. Smola (ed.), pp: 169 – 184, MIT Press Cambridge, MA, USA 1999.

Lang87

1. Lang. *Linear Algebra*. Springer-Verlag New York, 1987.

Li2015

Li, Shengren, and Nina Amenta. "Brute-force k-nearest neighbors search on the GPU." In International Conference on Similarity Search and Applications, pp. 259-270. Springer, Cham, 2015.

Lloyd82

Stuart P Lloyd. *Least squares quantization in PCM*. IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.

Matsumoto98

Matsumoto, M., Nishimura, T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, pp. 3-30, January 1998.

Matsumoto2000

Matsumoto, M., Nishimura, T. Dynamic Creation of Pseudorandom Number Generators Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, pp. 56-69, available from http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html.

Mitchell97

Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

Mu2014

Mu Li, Tong Zhang, Yuqiang Chen, Alexander J. Smola. *Efficient Mini-batch Training for Stochastic Optimization*, 2014. Available from https://www.cs.cmu.edu/~muli/file/minibatch_sgd.pdf.

OpenCLSpec

Khronos OpenCL Working Group, The OpenCL Specification Version:2.1 Document Revision:24 Available from opencl-2.1.pdf

Patwary2016

Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jialin Liu, Peter Sadowski, Evan Racah, Suren Byna, Craig Tull, Wahid Bhimji, Prabhat, Pradeep Dubey. *PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures*, 2016. Available from https://arxiv.org/abs/1607.08220.

Ping14

Ping Tak Peter and Eric Polizzi. *FEAST as a Subspace Iteration Eigensolver Accelerated by Approximate Spectral Projection.* 2014.

Platt98

Platt, John. "Sequential minimal optimization: A fast algorithm for training support vector machines." (1998). Available from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf.

Quinlan86

J. R. Quinlan. *Induction of Decision Trees*. Machine Learning, Volume 1 Issue 1, pp. 81-106, 1986.

Quinlan87

J. R. Quinlan. *Simplifying decision trees*. International journal of Man-Machine Studies, Volume 27 Issue 3, pp. 221-234, 1987.

Renie03

Jason D.M. Rennie, Lawrence, Shih, Jaime Teevan, David R. Karget. *Tackling the Poor Assumptions of Naïve Bayes Text classifiers*. Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

Rumelhart86

David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams. *Learning representations by back-propagating errors*. Nature (323), pp. 533-536, 1986.

Sokolova09

Marina Sokolova, Guy Lapalme. A systematic analysis of performance measures for classification tasks. Information Processing and Management 45 (2009), pp. 427–437. Available from http://atour.iro.umontreal.ca/rali/sites/default/files/publis/SokolovaLapalme-JIPM09.pdf.

SYCLSpec

Khronos®OpenCL™ Working Group — SYCL™ subgroup, SYCL™ Specification SYCL™ integrates OpenCL™ devices with modern C++, Version 1.2.1 Available from sycl-1.2.1.pdf

Sutton2018

Michael Sutton, Tal Ben-Nun, Amnon Barak. *Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling*. Symposium on Parallel and Distributed Processing, IPDPS 2018.

Tan2005

Pang-Ning Tan, Michael Steinbach, Vipin Kumar, Introduction to Data Mining, (First Edition) Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2005, ISBN: 032132136.

Verma2014

Verma, Deepika, Namita Kakkar, and Neha Mehan. "Comparison of brute-force and KD tree algorithm." International Journal of Advanced Research in Computer and Communication Engineering 3, no. 1 (2014): 5291-5294.

Wen2018

Wen, Zeyi, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. ThunderSVM: A fast SVM library on GPUs and CPUs. The Journal of Machine Learning Research, 19, 1-5 (2018).

Wu04

Ting-Fan Wu, Chih-Jen Lin, Ruby C. Weng. *Probability Estimates for Multi-class Classification by Pairwise Coupling*. Journal of Machine Learning Research 5, pp: 975-1005, 2004.

Zhu2005

Zhu, Ji, Hui Zou, Saharon Rosset and Trevor J. Hastie. *Multi-class AdaBoost*. 2005

C++ API

- Data Management
  - Array
  - Accessors
  - Data Sources
  - Graphs
  - Graph Service
  - Tables
- Algorithms
  - Clustering
  - Covariance
  - Decomposition
  - Ensembles
  - Graph
  - Kernel Functions
  - Nearest Neighbors (kNN)
  - Pairwise Distances
  - Statistics
  - Support Vector Machines
- Distributed Model: Single Process Multiple Data
  - Distributed SPMD model
  - Communicators

Data Management

Refer to Developer Guide: Data Management.

- Array
  - Programming interface
  - Usage example
- Accessors
  - Column accessor
    - Usage example
    - Programming interface
  - Row accessor
    - Usage example
    - Programming interface
- Data Sources
  - CSV data source

Array

Refer to Developer Guide: Array.

Programming interface

All types and functions in this section are declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/array.hpp` header file.

All the `array` class methods can be divided into several groups:

1. Constructors that are used to create an array from external, mutable or immutable memory.
2. Constructors and assignment operators that are used to create an array that shares its data with another one.
3. The group of `reset()` methods that are used to re-assign an array to another external memory block.
4. The group of `reset()` methods that are used to re-assign an array to an internally allocated memory block.
5. The methods that are used to access the data.
6. Static methods that provide simplified ways to create an array either from external memory or by allocating it within a new object.

***template*<*typename*T>*class*array**

Template Parameters

T – The type of the memory block elements within the array. `T` can represent any type.

Public Static Methods

static**array<T>empty(std::int64_tcount)**

Allocates a new memory block for mutable data, does not initialize it, creates a new array instance by passing a pointer to the memory block. The array owns the memory block (for details, see data ownership requirements).

Parameters

count – The number of elements of type `Data` to allocate memory for.

Preconditions

count>0

template<typename**K>*****static*****array<T>full(std::int64_tcount, K&&element)**

Allocates a new memory block for mutable data, fills it with a scalar value, creates a new array instance by passing a pointer to the memory block. The array owns the memory block (for details, see data ownership requirements).

Parameters

- **count** – The number of elements of type `T` to allocate memory for.
- **element** – The value that is used to fill a memory block.

Preconditions

count>0
Elements of type **T** are constructible from the **Element** type.

static**array<T>zeros(std::int64_tcount)**

Allocates a new memory block on mutable data, fills it with zeros, creates a new array instance by passing a pointer to the memory block. The array owns the memory block (for details, see data ownership requirements).

Parameters

count – The number of elements of type `Data` to allocate memory for.

Preconditions

count>0

template<typename**Y>*****static*****array<T>wrap(Y*data, std::int64_tcount)**

Creates a new array instance by passing the pointer to externally-allocated memory block for mutable data. It is the responsibility of the calling application to free the memory block as the array does not free it when the reference count is zero.

Parameters

- **data** – The pointer to externally-allocated memory block.
- **count** – The number of elements of type `Data` in the memory block.

Preconditions

data!=***nullptr*****count>0**

Constructors

array()

Creates a new instance of the class without memory allocation: `mutable_data` and `data` pointers should be set to **nullptr**, `count` should be zero; the pointer to the ownership structure should be set to **nullptr**.

array(*const*array<T>&other)

Creates a new array instance that shares an ownership with `other` on its memory block.

array(array<T>&&other)

Moves `data`, `mutable_data` pointers, `count`, and pointer to the ownership structure in `other` to the new array instance.

template<*typename*Deleter>array(T*data, std::int64_tcount, Deleter&&deleter)

Creates a new array instance which owns a memory block of externally-allocated mutable data. The ownership structure is created for a block, the input `deleter` is assigned to it.

Template Parameters

Deleter – The type of a deleter used to free the `Data`. The deleter provides **void operator()(Data*)** member function.

Parameters

- **data** – The pointer to externally-allocated memory block.
- **count** – The number of elements of type `Data` in the memory block.
- **deleter** – The object used to free `Data`.

template<*typename*ConstDeleter>array(*const*T*data, std::int64_tcount, ConstDeleter&&deleter)

Creates a new array instance which owns a memory block of externally-allocated immutable data. The ownership structure is created for a block, the input `deleter` is assigned to it.

Template Parameters

ConstDeleter – The type of a deleter used to free the `Data`. The deleter implements **void operator()(const Data*)** member function.

Parameters

- **data** – The pointer to externally-allocated memory block.
- **count** – The number of elements of type `Data` in the `Data`.
- **deleter** – The object used to free `Data`.

array(*const*std::shared_ptr<T>&data, std::int64_tcount)

Creates a new array instance that shares ownership with the user-provided shared pointer.

Parameters

- **data** – The shared pointer to externally-allocated memory block.
- **count** – The number of elements of type `Data` in the memory block.

array(*const*std::shared_ptr<*const*T>&data, std::int64_tcount)

Creates a new array instance that shares ownership with the user-provided shared pointer.

Parameters

- **data** – The shared pointer to externally-allocated memory block.
- **count** – The number of elements of type `Data` in the memory block.

template<*typename*Y,*typename*K>array(*const*array<Y>&ref, K*data, std::int64_tcount)

An aliasing constructor: creates a new array instance that stores `Data` pointer, assigns the pointer to the ownership structure of `ref` to the new instance. Array returns `Data` pointer as its mutable or immutable block depending on the `Data` type.

Template Parameters

- **Y** – The type of elements in the referenced array.
- **K** – Either `T` or $constT$ type.

Parameters

- **ref** – The array which shares ownership structure with created one.

- **data** – Mutable or immutable unmanaged pointer hold by created array.
- **count** – The number of elements of type `T` in the `Data`.

Preconditions	**std::is_same_v<data,*const*T>||std::is_same_v<data,T>**

Public Methods

array<T>*operator*=(*const*array<T>&other)

Replaces the `data`, `mutable_data` pointers, `count`, and pointer to the ownership structure in the array instance by the values in `other`.

Postconditions	**data==other.datamutable_data==other.mutable_datacount==other.count**

array<T>*operator*=(array<T>&&other)

Swaps the values of `data`, `mutable_data` pointers, `count`, and pointer to the ownership structure in the array instance and `other`.

T*get_mutable_data()*const*

The pointer to the memory block holding mutable data.

Preconditions	**has_mutable_data()==*true***, othewise throws **domain_error**

const*T*get_data()*constnoexcept

The pointer to the memory block holding immutable data.

boolhas_mutable_data()*constnoexcept*

Returns whether array contains `mutable_data` or not.

array&need_mutable_data()

Returns mutable_data, if array contains it. Otherwise, allocates a memory block for mutable data and fills it with the data stored at `data`. Creates the ownership structure for allocated memory block and stores the pointer.

Postconditions	**has_mutable_data()==*true***

std::int64_tget_count()*constnoexcept*

The number of elements of type `T` in a memory block.

std::int64_tget_size()*constnoexcept*

The size of memory block in bytes.

voidreset()

Resets ownership structure pointer to **nullptr**, sets `count` to zero, `data` and `mutable_data` to ***nullptr***.

voidreset(std::int64_tcount)

Allocates a new memory block for mutable data, does not initialize it, creates ownership structure for this block, assigns the structure inside the array. The array owns allocated memory block.

Parameters	**count** – The number of elements of type `Data` to allocate memory for.

template<*typename*Deleter>voidreset(T*data, std::int64_tcount, Deleter&&deleter)

Creates the ownership structure for memory block of externally-allocated mutable data, assigns input `deleter` object to it, sets `data` and `mutable_data` pointers to this block.

Template Parameters	**Deleter** – The type of a deleter used to free the `Data`. The deleter implements **void operator()(Data*)** member function.
Parameters	• **data** – The mutable memory block pointer to be assigned inside the array. • **count** – The number of elements of type `Data` into the block. • **deleter** – The object used to free `Data`.

template<*typename*ConstDeleter>voidreset(*const*T*data, std::int64_tcount, ConstDeleter&&deleter)

Creates the ownership structure for memory block of externally-allocated immutable data, assigns input `deleter` object to it, sets `data` pointer to this block.

Template Parameters	**ConstDeleter** – The type of a deleter used to free. The deleter implements **void operator()(const Data*)**` member function.
Parameters	• **data** – The immutable memory block pointer to be assigned inside the array. • **count** – The number of elements of type `Data` into the block. • **deleter** – The object used to free `Data`.

template<*typename*Y>voidreset(*const*array<Y>&ref, T*data, std::int64_tcount)

Initializes `data` and `mutable_data` with data pointer, `count` with input `count` value, initializes the pointer to ownership structure with the one from ref. Array returns `Data` pointer as its mutable block.

Template Parameters	**Y** – The type of elements in the referenced array.
Parameters	• **ref** – The array which is used to share ownership structure with current one. • **data** – Mutable unmanaged pointer to be assigned to the array. • **count** – The number of elements of type `T` in the `Data`.

template<*typename*Y>voidreset(*const*array<Y>&ref, *const*T*data, std::int64_tcount)

Initializes `data` with data pointer, `count` with input `count` value, initializes the pointer to ownership structure with the one from ref. Array returns `Data` pointer as its immutable block.

Template Parameters	**Y** – The type of elements in the referenced array.
Parameters	• **ref** – The array which is used to share ownership structure with current one. • **data** – Immutable unmanaged pointer to be assigned to the array. • **count** – The number of elements of type `T` in the `Data`.

*const*T&*operator*[](std::int64_tindex)*constnoexcept*

Provides a read-only access to the elements of array. Does not perform boundary checks.

Usage example

The following listing provides a brief introduction to the array API and an example of basic usage scenario:

```
#include <sycl/sycl.hpp>
#include <iostream>
#include <string>
#include "oneapi/dal/array.hpp"

using namespace oneapi;

void print_property(const std::string& description, const auto& property) {
   std::cout << description << ": " << property << std::endl;
}

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr std::int64_t data_count = 4;
   const float data[] = { 1.0f, 2.0f, 3.0f, 4.0f };

   // Creating an array from immutable user-defined memory
   auto arr_data = dal::array<float>::wrap(data, data_count);

   // Creating an array from internally allocated memory filled by ones
   auto arr_ones = dal::array<float>::full(queue, data_count, 1.0f);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_ones mutable", arr_ones.has_mutable_data()); // true

   // Creating new array from arr_data without data copy - they share ownership information.
   dal::array<float> arr_mdata = arr_data;

   print_property("arr_mdata elements count", arr_mdata.get_count()); // equal to data_count
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // false

   /// Copying data inside arr_mdata to new mutable memory block.
   /// arr_data still refers to the original data pointer.
   arr_mdata.need_mutable_data(queue);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // true

   queue.submit([&](sycl::handler& cgh){
      auto mdata = arr_mdata.get_mutable_data();
      auto cones = arr_ones.get_data();
      cgh.parallel_for<class array_addition>(sycl::range<1>(data_count), [=](sycl::id<1> idx) {
         mdata[idx[0]] += cones[idx[0]];
      });
   }).wait();

   std::cout << "arr_mdata values: ";
   for(std::int64_t i = 0; i < arr_mdata.get_count(); i++) {
      std::cout << arr_mdata[i] << ", ";
   }
   std::cout << std::endl;

   return 0;
}
```

Accessors

The requirements for accessors and accessor types are defined in Developer Guide: Accessors.

Column accessor

The `column_accessor` class provides a read-only access to the column values of the table as contiguoushomogeneous array.

Usage example

```
#include <sycl/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/column_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count * column_count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing whole elements in a first column
   dal::column_accessor<const float> acc { t };

   auto block = acc.pull(queue, 0);
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

All types and functions in this section are declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/column_accessor.hpp` header file.

***template*<*typename*T>*class*column_accessor**

Template Parameters	**T** – The type of data values in blocks returned by the accessor. Should be const-qualified for read-only access. An accessor supports at least `float`, `double`, and `std::int32_t`.

Constructors

***template*<*typename*U=T,std::enable_if_t<std::is_const_v<U>,int>=0>column_accessor(*const*table&table)**

Creates a read-only accessor object from the table. Available only for const-qualified `T`.

column_accessor(*const*detail::table_builder&builder)

Public Methods

dal::array<data_t>pull(std::int64_tcolumn_index, *const*range&row_range={0,-1})*const*

Provides access to the column values of the table. The method returns an array that directly points to the memory within the table if it is possible. In that case, the array refers to the memory as to immutable data. Otherwise, the new memory block is allocated, the data from the table rows is converted and copied into this block. In this case, the array refers to the block as to mutable data.

Parameters	• **column_index** – The index of the column from which the data is returned by the accessor. • **row_range** – The range of rows that should be read in the `column_index` block.
Preconditions	**row_range** are within the range of **[0, obj.row_count)**. **column_index** is within the range of **[0, obj.column_count)**.

T*pull(dal::array<data_t>&block, std::int64_tcolumn_index, *const*range&row_range={0,-1})*const*

Provides access to the column values of the table. The method returns an array that directly points to the memory within the table if it is possible. In that case, the array refers to the memory as to immutable data. Otherwise, the new memory block is allocated, the data from the table rows is converted and copied into this block. In this case, the array refers to the block as to mutable data. The method updates the **block** array.

Parameters	• **block** – The block which memory is reused (if it is possible) to obtain the data from the table. The block memory is reset either when its size is not big enough, or when it contains immutable data, or when direct memory from the table can be used. If the block is reset to use a direct memory pointer from the object, it refers to this pointer as to immutable memory block. • **column_index** – The index of the column from which the data is returned by the accessor. • **row_range** – The range of rows that should be read in the `column_index` block.
Preconditions	**row_range** are within the range of **[0, obj.row_count)**. **column_index** is within the range of **[0, obj.column_count)**.

***template<typename*U=T,std::enable_if_t<! std::is_const_v<U>,int>=0>voidpush(*const*dal::array<data_t>&block, std::int64_tcolumn_index, *const*range&row_range={0,-1})**

Row accessor

The `row_accessor` class provides a read-only access to the rows of the table as contiguoushomogeneous array.

Usage example

```
#include <sycl/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/row_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count * column_count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing second and third rows of the table
   dal::row_accessor<const float> acc { t };

   auto block = acc.pull(queue, {1, 3});
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

All types and functions in this section are declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/row_accessor.hpp` header file.

***template<typename*T>*class*row_accessor**

Template Parameters	**T** – The type of data values in blocks returned by the accessor. Should be const-qualified for read-only access. An accessor supports at least `float`, `double`, and `std::int32_t`.

Constructors

***template*<*typename*U=T,std::enable_if_t<std::is_const_v<U>,int>=0>row_accessor(*const*table& table)**

Creates a read-only accessor object from the table. Available only for const-qualified `T`.

row_accessor(*const*detail::table_builder&builder)

Public Methods

dal::array<data_t>pull(*const*range&row_range={0,-1})*const*

Provides access to the rows of the table. The method returns an array that directly points to the memory within the table if it is possible. In that case, the array refers to the memory as to immutable data. Otherwise, the new memory block is allocated, the data from the table rows is converted and copied into this block. In this case, the array refers to the block as to mutable data.

Parameters	**row_range** – The range of rows that data is returned from the accessor.
Preconditions	**row_range** are within the range of **[0, obj.row_count)**.

T*pull(dal::array<data_t>&block, *const*range&row_range={0,-1})*const*

Provides access to the rows of the table. The method returns an array that directly points to the memory within the table if it is possible. In that case, the array refers to the memory as to immutable data. Otherwise, the new memory block is allocated, the data from the table rows is converted and copied into this block. In this case, the array refers to the block as to mutable data. The method updates the **block** array.

Parameters	• **block** – The block which memory is reused (if it is possible) to obtain the data from the table. The block memory is reset either when its size is not big enough, or when it contains immutable data, or when direct memory from the table can be used. If the block is reset to use a direct memory pointer from the object, it refers to this pointer as to immutable memory block. • **row_range** – The range of rows that data is returned from the accessor.
Preconditions	**rows** are within the range of **[0, obj.row_count)**.

***template*<*typename*U=T,std::enable_if_t<! std::is_const_v<U>,int>=0>voidpush(*const*dal::array<data_t>&block, *const*range&row_range={0,-1})**

Data Sources

CSV data source

Refer to Developer Guide: CSV data source.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::csv` namespace and be available via inclusion of the `oneapi/dal/io/csv.hpp` header file.

```
enum class read_options : std::uint64_t {
   none = 0,
   parse_header = 1 << 0
};

constexpr char default_delimiter = ',';
constexpr read_options default_read_options = read_options::none;

class data_source {
public:
   data_source(const char *file_name,
               char delimiter = default_delimiter,
               read_options opts = default_read_options);

   data_source(const std::string &file_name,
               char delimiter = default_delimiter,
               read_options opts = default_read_options);

   std::string get_file_name() const;
   char get_delimiter() const;
   read_options get_read_options() const;
};
```

***class*data_source**

data_source(*const*char*file_name, chardelimiter=default_delimiter, read_optionsopts=default_read_options)

Creates a new instance of a CSV data source with the given **file_name**, **delimiter** and read options **opts** flag.

data_source(*const*std::string&file_name, chardelimiter=default_delimiter, read_optionsopts=default_read_options)

Creates a new instance of a CSV data source with the given **file_name**, **delimiter** and read options **opts** flag.

std::stringfile_name=""

A string that contains the name of the file with the dataset to read.

Getter	`std::string get_filename() const`

chardelimiter=default_delimiter

A character that represents the delimiter between separate features in the input file.

Getter	`char get_delimter() const`

read_optionsoptions=default_read_options

Value that stores read options to be applied during reading of the input file. Enabled `parse_header` option indicates that the first line in the input file is processed as a header record with features names.

Getter	`read_options get_read_options() const`

Reading oneapi::dal::read<Object>(...)

Args

```
template <typename Object>
class read_args {
public:
   read_args();
};
```

template<typenameObject>*class*read_args**

read_args()

Creates args for the read operation with the default attribute values.

Operation

oneapi::dal::table is the only supported value of the `Object` template parameter for **read** operation with CSV data source.

template<typenameObject,***typename***DataSource>Objectread(*const*DataSource&ds)**

Template Parameters	• **Object** – oneDAL object type that is produced as a result of reading from the data source. • **DataSource** – CSV data source **csv::data_source**.

Usage example

```
using namespace oneapi;

const auto data_source = dal::csv::data_source("data.csv", ',');

const auto table = dal::read<dal::table>(data_source);
```

Graphs

Refer to Developer Guide: Graphs.

Programming interface

All types and functions in this section are declared in the `oneapi::dal::preview` namespace and are available via inclusion of the `oneapi/dal/graph/common.hpp` header file.

Graph

The graph concept is represented by the types with the `_graph` suffix and all of them are reference-counted:

1. The instance stores pointers to the graph topology and attributes of vertices and edges.

2. The reference count indicates how many graph objects refer to the same implementation.
3. The graph increments the reference count for it to be equal to the number of graph objects sharing the same implementation.
4. The graph decrements the reference count when the graph goes out of the scope. If the reference count is zero, the graph frees its implementation.

The graph types are defined as templated classes:

```
template <typename VertexValue,
          typename EdgeValue,
          typename GraphValue,
          typename IndexType,
          typename Allocator>
class [graph_name]_graph;
```

Type name	Description	Supported types
`VertexValue`	The type of the vertex attribute values	Empty value
`EdgeValue`	The type of the edge attribute values	`std::int32`, `double`, Empty value
`GraphValue`	The type of the graph attribute value	Empty value
`IndexType`	The type of the vertex indices	`std::int32`
`Allocator`	The type of a graph allocator	C++17 (ISO/IEC 14882:2017) compliant allocator

Empty value tag structure is used to define the absence of a specified attribute of a graph.

***struct*empty_value**

Graph class contains the default and the move constructor as well as the move assignment operator. The graph is accessed using the service functions.

`graph_type` method	Description
Default constructor	Constructs an empty graph object
Move constructor	Creates a new graph instance and moves the implementation from another instance into this one
Move assignment	Swaps the implementation of this object and another one

Graph traits

Graph traits is a data type that defines the data model and a set of types associated with the graph. Graph traits are used by processing and service functionality.

Type graph_traits is specialized for each graph by following the pattern below.

```
template <typename G>
struct graph_traits {
   using graph_type = ...;
   using allocator_type = ...;
   ...
};
```

The full list of types defined in `graph_traits<G>` is in the table below:

Type, `graph_traits<G>::`	**Description**	**Undirected Adjacency Vector Graph**	**Directed Adjacency Vector Graph**
`graph_type`	The type of the graph `G`	`undirected_adjacency_vector_graph<VertexValue, EdgeValue, GraphValue, IndexType, Allocator>`	`directed_adjacency_vector_graph<VertexValue, EdgeValue, GraphValue, IndexType, Allocator>`
`allocator_type`	The type of the allocator of the graph `G`	`Allocator`[1]	`Allocator`[1]
`graph_user_value_type`	The type of the attribute of the graph `G`	`GraphValue`[1]	`GraphValue`[1]
`const_graph_user_value_type`	The constant type of the attribute of the graph `G`	`const GraphValue`[1]	`const GraphValue`[1]
`vertex_type`	The type of the vertices in the graph `G`	`IndexType`[1]	`IndexType`[1]
`vertex_iterator`	The type of the vertex iterator in the graph `G`	`vertex_type*`	`vertex_type*`
`const_vertex_iterator`	The constant type of the vertex iterator in the graph `G`	`const vertex_type*`	`const vertex_type*`
`vertex_size_type`	The type of the vertex indices in the graph `G`	`std::int64_t`	`std::int64_t`
`vertex_user_value_type`	The type of the vertex attribute of the graph `G`	`VertexValue`[1]	`VertexValue`[1]
`edge_type`	The type of edges in the graph `G`	`std::int64_t`	`std::int64_t`
`edge_iterator`	The type of the edge iterator in the graph `G`	*Not available*	*Not available*
`const_edge_iterator`	The constant type of the edge iterator in the graph `G`	*Not available*	*Not available*
`edge_size_type`	The type of the edge indices in the graph `G`	`std::int64_t`	`std::int64_t`
`edge_user_value_type`	The type of edge attribute	`EdgeValue`[1]	`EdgeValue`[1]
`vertex_edge_size_type`	The type of the vertex neighbors indices	`std::int64_t`	*Not available*
`vertex_outward_edge_size_type`	The type of the vertex outward neighbors indices	*Not available*	`std::int64_t`
`vertex_edge_iterator_type`	The type of the vertex neighbors iterator	`IndexType*`[1]	*Not available*

Type, graph_traits<G> ::	Description	Undirected Adjacency Vector Graph	Directed Adjacency Vector Graph
const_vertex_edge_iterator_type	The type of the vertex neighbors constant iterator	const IndexType*[1]	*Not available*
vertex_outward_edge_iterator_type	The type of the vertex outward neighbors iterator	*Not available*	IndexType*[1]
const_vertex_outward_edge_iterator_type	The type of the vertex outward neighbors constant iterator	*Not available*	const IndexType*[1]
vertex_edge_range	The type of the range of vertex neighbors	std::pair<IndexType*, IndexType*>[1]	*Not available*
const_vertex_edge_range	The type of the constant range of vertex neighbors	std::pair<IndexType*, IndexType*>[1]	*Not available*
vertex_outward_edge_range	The type of the range of vertex outward neighbors	*Not available*	std::pair<IndexType*, IndexType*>[1]
const_vertex_outward_edge_range	The type of the constant range of vertex outward neighbors	*Not available*	std::pair<IndexType*, IndexType*>[1]

[1] VertexValue, EdgeValue, GraphValue, IndexType, Allocator – template parameters of graph G.

This section describes API of the specified graph types.

- Undirected adjacency vector graph
  - Programming interface
- Directed adjacency vector graph
  - Programming interface

Undirected adjacency vector graph

Refer to Developer Guide: Undirected adjacency vector graph.

Programming interface

All types and functions in this section are declared in the oneapi::dal::preview namespace and are available via inclusion of the oneapi/dal/graph/undirected_adjacency_vector_graph.hpp header file.

***template*<*typename*VertexValue=empty_value,*typename*EdgeValue=empty_value,*typename*GraphValue=empty_value,*typename*IndexType=std::int32_t,*typename*Allocator=std::allocator<char>>*class*undirected_adjacency_vector_graph**

Template Parameters

- **VertexValue** – The type of the vertex attribute values.
- **EdgeValue** – The type of the edge attribute values.
- **GraphValue** – The type of the graph attribute value.
- **IndexType** – The type of the vertex indices.
- **Allocator** – The type of a graph allocator.

Constructors

undirected_adjacency_vector_graph()

Constructs an empty graph.

~undirected_adjacency_vector_graph()=*default*

Destructs the graph.

undirected_adjacency_vector_graph(undirected_adjacency_vector_graph&&other)=*default*

Creates a new graph instance and moves the implementation from another instance into this one.

Public Methods

undirected_adjacency_vector_graph&*operator*=(undirected_adjacency_vector_graph&&other)

Swaps the implementation of this object and another one.

Directed adjacency vector graph

Refer to Developer Guide: Directed adjacency vector graph.

Programming interface

All types and functions in this section are declared in the `oneapi::dal::preview` namespace and are available via inclusion of the `oneapi/dal/graph/directed_adjacency_vector_graph.hpp` header file.

***template*<*typename*VertexValue=empty_value,*typename*EdgeValue=empty_value,*typename*GraphValue=empty_value,*typename*IndexType=std::int32_t,*typename*Allocator=std::allocator<char> >*class*directed_adjacency_vector_graph**

Template Parameters

- **VertexValue** – The type of the vertex attribute values.
- **EdgeValue** – The type of the edge attribute values.
- **GraphValue** – The type of the graph attribute value.
- **IndexType** – The type of the vertex indices.
- **Allocator** – The type of a graph allocator.

Constructors

directed_adjacency_vector_graph()

Constructs an empty graph.

~directed_adjacency_vector_graph()=*default*

Destructs the graph.

directed_adjacency_vector_graph(directed_adjacency_vector_graph&&other)=*default*

Creates a new graph instance and moves the implementation from another instance into this one.

Public Methods

directed_adjacency_vector_graph&*operator*=(directed_adjacency_vector_graph&&other)

Swaps the implementation of this object and another one.

Graph Service

Programming interface

All types and functions in this section are declared in the `oneapi::dal::preview` namespace and are available via inclusion of the `oneapi/dal/graph/service_functions.hpp` header file.

The graph service is a set of functions that allow you to get access to the elements and characteristics of the graph, such as vertex degree or edge attribute.

Graph service functions are defined as function templates with `Graph` as a template parameter. Graph service functions introduce aliases to `graph_traits` as shown below.

Related types

Aliases is a way to access graph types using shorter notation.

Alias	Value
`graph_allocator<G>`	`graph_traits<G>::allocator_type`
`graph_user_value_type<G>`	`graph_traits<G>::graph_user_value_type`
`vertex_user_value_type<G>`	`graph_traits<G>::vertex_user_value_type`
`edge_user_value_type<G>`	`graph_traits<G>::edge_user_value_type`
`vertex_type<G>`	`graph_traits<G>::vertex_type`
`vertex_size_type<G>`	`graph_traits<G>::vertex_size_type`
`edge_size_type<G>`	`graph_traits<G>::edge_size_type`
`vertex_edge_size_type<G>`	`graph_traits<G>::vertex_edge_size_type`
`vertex_outward_edge_size_type<G>`	`graph_traits<G>::vertex_outward_edge_size_type`
`vertex_edge_iterator_type<G>`	`graph_traits<G>::vertex_edge_iterator_type`
`const_vertex_edge_iterator_type<G>`	`graph_traits<G>::const_vertex_edge_iterator_type`
`vertex_edge_range_type<G>`	`graph_traits<G>::vertex_edge_range_type`
`const_vertex_edge_range_type<G>`	`graph_traits<G>::const_vertex_edge_range_type`
`const_vertex_outward_edge_range_type<G>`	`graph_traits<G>::const_vertex_outward_edge_range_type`

Graph service functions

Any service function has the following pattern:

```
template <typename Graph>
return_type<Graph> get_[graph_element](const Graph& g, ...);
```

template<typenameGraph>constexprautoget_vertex_count(constGraph&g)noexcept->vertex_size_type<Graph>

Returns the number of vertices in the graph.

Template Parameters	**Graph** – Type of the graph.
Parameters	**g** – Input graph object.

template<typenameGraph>constexprautoget_edge_count(constGraph&g)noexcept->edge_size_type<Graph>

Returns the number of edges in the graph.

Template Parameters	**Graph** – Type of the graph.
Parameters	**g** – Input graph object.

template<typenameGraph>constexprautoget_vertex_degree(constGraph&g, vertex_type<Graph>u)->vertex_edge_size_type<Graph>

Returns the degree for the specified vertex.

Template Parameters	**Graph** – Type of the graph.
Parameters	• **g** – Input graph object. • **u** – Vertex index.

template<typenameGraph>constexprautoget_vertex_neighbors(constGraph&g, vertex_type<Graph>u)->const_vertex_edge_range_type<Graph>

Returns the range of the vertex neighbors for the specified vertex.

Template Parameters	**Graph** – Type of the graph.
Parameters	• **g** – Input graph object. • **u** – Vertex index.

template<typenameGraph>constexprautoget_vertex_outward_degree(constGraph&g, vertex_type<Graph>u)->vertex_outward_edge_size_type<Graph>

Returns the outward degree for the specified vertex.

Template Parameters	**Graph** – Type of the graph.
Parameters	• **g** – Input graph object. • **u** – Vertex index.

template<typenameGraph>constexprautoget_vertex_outward_neighbors(constGraph&g, vertex_type<Graph>u)->const_vertex_outward_edge_range_type<Graph>

Returns the range of the vertex outward neighbors for the specified vertex.

Template Parameters	**Graph** – Type of the graph.

Parameters

- **g** – Input graph object.
- **u** – Vertex index.

***template*<*typename*Graph>*constexpr*autoget_edge_value(*const*Graph&g, vertex_type<Graph>u, vertex_type<Graph>v)->*const*edge_user_value_type<Graph>&**

Returns the value of an edge (u, v).

Template Parameters

Graph – Type of the graph.

Parameters

- **u** – Source vertex index.
- **v** – Destination vertex index.

Usage example

```
using graph_type = ...;
const my_graph_type g = ...;
std::cout << "The number of vertices: " << oneapi::dal::preview::get_vertex_count(g) <<
std::endl;
std::cout << "The number of edges: " << oneapi::dal::preview::get_edge_count(g) << std::endl;
```

Service functions for supported graphs

This section contains description of service functions supported for the specified graph types.

Service function	Valid graph concepts
`get_vertex_count`	undirected graph, directed graph
`get_edge_count`	undirected graph, directed graph
`get_vertex_degree`	undirected graph
`get_vertex_outward_degree`	directed graph
`get_vertex_neighbors`	undirected graph
`get_vertex_outward_neighbors`	directed graph
`get_edge_value`	undirected graph, directed graph

- Undirected adjacency vector graph service
- Directed adjacency vector graph service

Undirected adjacency vector graph service

This section describes graph service functions for Undirected adjacency vector graph.

Service function	Description
get_vertex_count	Get the number of vertices in the graph
get_edge_count	Get the number of edges in the graph
get_vertex_degree	Get the degree for the specified vertex
get_vertex_neighbors	Get the range of the vertex neighbors for the specified vertex

Directed adjacency vector graph service

This section describes graph service functions for Directed adjacency vector graph.

Service function	Description
get_vertex_count	Get the number of vertices in the graph
get_edge_count	Get the number of edges in the graph
get_vertex_outward_degree	Get the outward degree for the specified vertex
get_vertex_outward_neighbors	Get the range of the outward neighbors for the specified vertex
get_edge_value	Get the value of an edge represented as source and destination vertices

Tables

Refer to Developer Guide: Tables.

Programming interface

All types and functions in this section are declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/common.hpp` header file.

Table

A base implementation of the table concept. The `table` type and all of its subtypes are reference-counted:

1. The instance stores a pointer to table implementation that holds all property values and data
2. The reference count indicating how many table objects refer to the same implementation.
3. The table increments the reference count for it to be equal to the number of table objects sharing the same implementation.
4. The table decrements the reference count when the table goes out of the scope. If the reference count is zero, the table frees its implementation.

***class*table**

Constructors

table()

An empty table constructor: creates the table instance with zero number of rows and columns.

table(*const*table&)=*default*

Creates a new table instance that shares the implementation with another one.

table(table&&)

Creates a new table instance and moves implementation from another one into it.

Public Methods

table&*operator*=(*const*table&)=*default*

Replaces the implementation by another one.

table&*operator*=(table&&)

Swaps the implementation of this object and another one.

boolhas_data()*constnoexcept*

Indicates whether a table contains non-zero number of rows and columns.

std::int64_tget_column_count()*const*

The number of columns in the table.

std::int64_tget_row_count()*const*

The number of rows in the table.

const*table_metadata&get_metadata()*const

The metadata object that holds additional information about the data within the table.

std::int64_tget_kind()*const*

The runtime id of the table type. Each table sub-type has its unique **kind**. An empty table has a unique **kind** value as well.

data_layoutget_data_layout()*const*

The layout of the data within the table.

Table metadata

An implementation of the table metadata concept. Holds additional information about data within the table. The objects of `table_metadata` are reference-counted.

***class*table_metadata**

Constructors

table_metadata()

Creates the metadata instance without information about the features. The `feature_count` should be set to zero. The `data_type` and `feature_type` properties should not be initialized.

table_metadata(*const*dal::array<data_type>&dtypes, *const*dal::array<feature_type>&ftypes)

Creates the metadata instance from external information about the data types and the feature types.

Parameters	- **dtypes** – The data types of the features. Assigned into the `data_type` property. - **ftypes** – The feature types. Assigned into the `feature_type` property.
Preconditions	**dtypes.get_count()==ftypes.get_count()**

Public Methods

std::int64_tget_feature_count()*const*

The number of features that metadata contains information about.

const*feature_type&get_feature_type(std::int64_tfeature_index)*const

Feature types in the metadata object. Should be within the range **[0, feature_count)**.

const*data_type&get_data_type(std::int64_tfeature_index)*const

Data types of the features in the metadata object. Should be within the range **[0, feature_count)**.

Data layout

An implementation of the data layout concept.

```
enum class data_layout { unknown, row_major, column_major };
```

data_layout::unknown	Represents the data layout that is undefined or unknown at this moment.
data_layout::row_major	The data block elements are stored in raw-major layout.
data_layout::column_major	The data block elements are stored in column_major layout.

Feature type

An implementation of the logical data types.

```
enum class feature_type { nominal, ordinal, interval, ratio };
```

feature_type::nominal	Represents the type of Nominal feature.
feature_type::ordinal	Represents the type of Ordinal feature.
feature_type::interval	Represents the type of Interval feature.
feature_type::ratio	Represents the type of Ratio feature.

- Homogeneous table
  - Programming interface

Homogeneous table

Refer to Developer Guide: Homogeneous table.

Programming interface

All types and functions in this section are declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/homogen.hpp` header file.

*class*homogen_table

Public Static Methods

*static*std::int64_tkind()

Returns the unique id of **homogen_table** class.

*template<typename*Data>*static*homogen_tablewrap(*const*Data*data_pointer, std::int64_trow_count, std::int64_tcolumn_count, data_layoutlayout=data_layout::row_major)

Creates a new **homogen_table** instance from externally-defined data block. Table object refers to the data but does not own it. The responsibility to free the data remains on the user side. The `data` should point to the **data_pointer** memory block.

Template Parameters

Data – The type of elements in the data block that will be stored into the table. The table initializes data types of metadata with this data type. The feature types should be set to default values for `Data` type: contiguous for floating-point, ordinal for integer types. The `Data` type should be at least **float**, **double** or **std::int32_t**.

Parameters

- **data_pointer** – The pointer to a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.
- **layout** – The layout of the data. Should be `data_layout::row_major` or `data_layout::column_major`.

***template*<*typename*Data>*static*homogen_tablewrap(*const*dal::array<Data>&data, std::int64_trow_count, std::int64_tcolumn_count, data_layoutlayout=data_layout::row_major)**

Creates a new **homogen_table** instance from an array. The created table shares data ownership with the given array.

Template Parameters

Data – The type of elements in the data block that will be stored into the table. The table initializes data types of metadata with this data type. The feature types should be set to default values for `Data` type: contiguous for floating-point, ordinal for integer types. The `Data` type should be at least **float**, **double** or **std::int32_t**.

Parameters

- **data** – The array that stores a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.
- **layout** – The layout of the data. Should be `data_layout::row_major` or `data_layout::column_major`.

Constructors

homogen_table()

Creates a new **homogen_table** instance with zero number of rows and columns.

homogen_table(*const*table&other)

Casts an object of the base table type to a homogen table. If cast is not possible, the operation is equivalent to a default constructor call.

***template*<*typename*Data,*typename*ConstDeleter>homogen_table(*const*Data*data_pointer, std::int64_trow_count, std::int64_tcolumn_count, ConstDeleter&&data_deleter, data_layoutlayout=data_layout::row_major)**

Creates a new **homogen_table** instance from externally-defined data block. Table object owns the data pointer. The `data` should point to the **data_pointer** memory block.

Template Parameters

- **Data** – The type of elements in the data block that will be stored into the table. The `Data` type should be at least **float**, **double** or **std::int32_t**.
- **ConstDeleter** – The type of a deleter called on **data_pointer** when the last table that refers it is out of the scope.

Parameters

- **data_pointer** – The pointer to a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.

- **data_deleter** – The deleter that is called on the **data_pointer** when the last table that refers it is out of the scope.
- **layout** – The layout of the data. Should be `data_layout::row_major` or `data_layout::column_major`.

Public Methods

template<typename*Data*>const*Data*get_data()*const

Returns the `data` pointer cast to the `Data` type. No checks are performed that this type is the actual type of the data within the table. If table has no data, returns `nullptr`.

const*void*get_data()*const

The pointer to the data block within the table. Should be equal to **nullptr** when **row_count==0** and **column_count==0**.

std::int64_tget_kind()*const*

The unique id of the homogen table type.

Algorithms

Refer to Developer Guide for mathematical descriptions of the algorithms.

- Clustering
  - DBSCAN
  - K-Means
  - K-Means initialization
- Covariance
  - Covariance
- Decomposition
  - Principal Components Analysis (PCA)
- Ensembles
  - Decision Forest Classification and Regression (DF)
- Graph
  - Subgraph Isomorphism
  - Connected Components
- Kernel Functions
  - Linear kernel
  - Polynomial kernel
  - Radial Basis Function (RBF) kernel
  - Sigmoid kernel
- Nearest Neighbors (kNN)
  - k-Nearest Neighbors Classification (k-NN)
- Pairwise Distances
  - Minkowski distance
  - Chebyshev distance
  - Cosine distance
- Statistics
  - Basic Statistics
- Support Vector Machines
  - Support Vector Machine Classifier (SVM)

Clustering

This chapter describes programming interfaces of the clustering algorithms implemented in oneDAL:

- DBSCAN
- K-Means
- K-Means initialization

DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed in [Ester96]. It is a density-based clustering non-parametric algorithm: given a set of observations in some space, it groups together observations that are closely packed together (observations with many nearby neighbors), marking as outliers observations that lie alone in low-density regions (whose nearest neighbors are too far away).

Operation	Computational methods	Programming Interface		
Compute	Default method	compute(…)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: DBSCAN.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::dbscan` namespace and are available via inclusion of the `oneapi/dal/algo/dbscan.hpp` header file.

Descriptor

***template<typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::brute_force**.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::clustering**.

Constructors

descriptor(doubleepsilon, std::int64_tmin_observations)

Creates a new instance of the class with the given `epsilon`, `min_observations`.

Properties

doubleepsilon

The distance **epsilon** for neighbor search.

Getter & Setter	`double get_epsilon() constauto & set_epsilon(double value)`
Invariants	**epsilon>=0.0**

boolmem_save_mode

The flag for memory saving mode.

Getter & Setter	`bool get_mem_save_mode() constauto & set_mem_save_mode(bool value)`

std::int64_tmin_observations

The number of neighbors.

Getter & Setter	`std::int64_t get_min_observations() constauto & set_min_observations(std::int64_t value)`

result_option_idresult_options

Choose which results should be computed and returned.

Getter & Setter	`result_option_id get_result_options() constauto & set_result_options(const result_option_id &value)`

Method tags

***struct*brute_force**

***using*by_default=brute_force**

Task tags

***struct*clustering**

Tag-type that parameterizes entities used for solving clustering problem.

***using*by_default=clustering**

Alias tag-type for the clustering task.

Computation compute(...)

Input

***template*<*typename*Task=task::by_default>*class*compute_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

compute_input(*const*table&data={}, *const*table&weights={})

Creates a new instance of the class with the given `data` and `weights`.

Properties

*const*table&data

An $n \times p$ table with the data to be clustered, where each row stores one feature vector.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

*const*table&weights

A single column table with the weights, where each row stores one weight per observation.

Getter & Setter	`const table & get_weights() constauto & set_weights(const table &weights)`

Result

template<*typename*Task=task::by_default>*class*compute_result

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

*const*table&responses

An $n \times 1$ table with the responses y_i assigned to the samples x_i in the input data. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

*const*table&core_flags

An $n \times 1$ table with the core flags y_i assigned to the samples x_i in the input data.

Getter & Setter	`const table & get_core_flags() constauto & set_core_flags(const table &value)`

*const*result_option_id&result_options

Result options that indicates availability of the properties. **Default value**: default_result_options<Task>.

Getter & Setter	`const result_option_id & get_result_options() constauto & set_result_options(const result_option_id &value)`

*const*table&core_observations

An $m \times p$ table with the core observations in the input data. m is a number of core observations.

Getter & Setter

`const table & get_core_observations() constauto & set_core_observations(const table &value)`

_const_table&core_observation_indices

An $m \times 1$ table with the indices of core observations in the input data. m is a number of core observations.

Getter & Setter

`const table & get_core_observation_indices() constauto & set_core_observation_indices(const table &value)`

std::int64_tcluster_count

The number of clusters found by the algorithm.

Getter & Setter

`std::int64_t get_cluster_count() constauto & set_cluster_count(std::int64_t value)`

Invariants

cluster_count>=0

Operation

***template<typename*Descriptor>dbscan::compute_resultcompute(*const*Descriptor&desc, *const*dbscan::compute_input&input)**

Parameters

- **desc** – DBSCAN algorithm descriptor **dbscan::descriptor**
- **input** – Input data for the compute operation

Preconditions

input.data.has_data==*true*!input.weights.has_data|| input.weights.row_count==input.data.row_count&&input.weights.column_count==1

Usage example

Compute

```
void run_compute(const table& data,
                           const table& weights) {
   double epsilon = 1.0;
   std::int64_t max_observations = 5;
   const auto dbscan_desc = kmeans::descriptor<float>{epsilon, max_observations}
      .set_result_options(dal::dbscan::result_options::responses);

   const auto result = compute(dbscan_desc, data, weights);

   print_table("responses", result.get_responses());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_dbscan_brute_force_batch.cpp

oneAPI C++

Batch Processing:

- cpp_dbscan_brute_force_batch.cpp

Python* with DPC++ support

Batch Processing:

- dbscan_batch.py

K-Means

The K-Means algorithm solves clustering problem by partitioning *n* feature vectors into *k* clusters minimizing some criterion. Each cluster is characterized by a representative point, called *a centroid*.

Operation	Computational methods	Programming Interface		
Training	Lloyd's	train(…)	train_input	train_result
Inference	Lloyd's	infer(…)	infer_input	infer_result

Mathematical formulation

Refer to Developer Guide: K-Means.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::kmeans` namespace and be available via inclusion of the `oneapi/dal/algo/kmeans.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::lloyd_dense**.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::clustering**.

Constructors

descriptor(std::int64_tcluster_count=2)

Creates a new instance of the class with the given `cluster_count`.

Properties

std::int64_tmax_iteration_count

The maximum number of iterations `T`. **Default value**: 100.

Getter & Setter	`std::int64_t get_max_iteration_count() constauto & set_max_iteration_count(std::int64_t value)`
Invariants	**max_iteration_count>=0**

std::int64_tcluster_count

The number of clusters k. **Default value**: 2.

Getter & Setter	`std::int64_t get_cluster_count() constauto & set_cluster_count(std::int64_t value)`
Invariants	**cluster_count>0**

doubleaccuracy_threshold

The threshold ε for the stop condition. **Default value**: 0.0.

Getter & Setter	`double get_accuracy_threshold() constauto & set_accuracy_threshold(double value)`
Invariants	**accuracy_threshold>=0.0**

Method tags

***struct*lloyd_dense**

Tag-type that denotes Lloyd's computational method.

***using*by_default=lloyd_dense**

Alias tag-type for Lloyd's computational method.

Task tags

***struct*clustering**

Tag-type that parameterizes entities used for solving clustering problem.

***using*by_default=clustering**

Alias tag-type for the clustering task.

Model

***template*<*typename*Task=task::by_default>*class*model**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

model()

Creates a new instance of the class with the default property values.

Public Methods

std::int64_tget_cluster_count()*const*

Number of clusters k in the trained model.

Properties

***const*table&centroids**

A $k \times p$ table with the cluster centroids. Each row of the table stores one centroid. **Default value**: table{}.

Getter & Setter	`const table & get_centroids() constauto & set_centroids(const table &value)`

Training train(...)

Input

***template<typename*Task=task::by_default>*class*train_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

train_input(*const*table&data)

train_input(*const*table&data, *const*table&initial_centroids)

Creates a new instance of the class with the given `data` and `initial_centroids`.

Properties

***const*table&data**

An $n \times p$ table with the data to be clustered, where each row stores one feature vector.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

***const*table&initial_centroids**

A $k \times p$ table with the initial centroids, where each row stores one centroid.

Getter & Setter	`const table & get_initial_centroids() constauto & set_initial_centroids(const table &data)`

Result

***template<typename*Task=task::by_default>*class*train_result**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Properties

*const*table&responses

An $n \times 1$ table with the responses y_i assigned to the samples x_i in the input data, $1 \leq 1 \leq n$. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

std::int64_titeration_count

The number of iterations performed by the algorithm. **Default value**: 0.

Getter & Setter	`std::int64_t get_iteration_count() constauto & set_iteration_count(std::int64_t value)`
Invariants	**iteration_count>=0**

doubleobjective_function_value

The value of the objective function $\Phi_X(C)$, where C is **model.centroids**.

Getter & Setter	`double get_objective_function_value() constauto & set_objective_function_value(double value)`
Invariants	**objective_function_value>=0.0**

*const*table&labels

An $n \times 1$ table with the labels y_i assigned to the samples x_i in the input data, $1 \leq 1 \leq n$. **Default value**: table{}.

Getter & Setter	`const table & get_labels() constauto & set_labels(const table &value)`

*const*model<Task>&model

The trained K-means model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Operation

*template<typename*Descriptor>kmeans::train_resulttrain(*const*Descriptor&desc, *const*kmeans::train_input&input)

Parameters	• **desc** – K-Means algorithm descriptor **kmeans::descriptor** • **input** – Input data for the training operation

Preconditions	**input.data.has_data==*true*input.initial_centroids.row_count==desc.cluster_countinput.initial_centroids.column_count==input.data.column_count**
Postconditions	**result.labels.row_count==input.data.row_countresult.labels.column_count==1result.labels[i]>=0result.labels[i]<desc.cluster_countresult.iteration_count<=desc.max_iteration_countresult.model.centroids.row_count==desc.cluster_countresult.model.centroids.column_count==input.data.column_count**

Inference infer(...)

Input

***template*<*typename*Task=task::by_default>*class*infer_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

infer_input(*const*model<Task>&trained_model, *const*table&data)

Creates a new instance of the class with the given `model` and `data`.

Properties

***const*table&data**

The trained K-Means model. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &value)`

***const*model<Task>&model**

An $n \times p$ table with the data to be assigned to the clusters, where each row stores one feature vector. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Result

***template*<*typename*Task=task::by_default>*class*infer_result**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::clustering**.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&labels**

An $n \times 1$ table with assignments labels to feature vectors in the input data. **Default value**: table{}.

Getter & Setter	`const table & get_labels() constauto & set_labels(const table &value)`

doubleobjective_function_value

The value of the objective function $\Phi_X(C)$, where C is defined by the corresponding **infer_input::model::centroids**. **Default value**: 0.0.

Getter & Setter	`double get_objective_function_value() constauto & set_objective_function_value(double value)`
Invariants	**objective_function_value>=0.0**

***const*table&responses**

An $n \times 1$ table with assignments responses to feature vectors in the input data. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

Operation

***template*<*typename*Descriptor>kmeans::infer_resultinfer(*const*Descriptor&desc, *const*kmeans::infer_input&input)**

Parameters	• **desc** – K-Means algorithm descriptor **kmeans::descriptor** • **input** – Input data for the inference operation
Preconditions	**input.data.has_data==*true*input.model.centroids.has_data==*true*input.model.centroids.row_count==desc.cluster_countinput.model.centroids.column_count==input.data.column_count**
Postconditions	**result.labels.row_count==input.data.row_countresult.labels.column_count==1result.labels[i]>=0result.labels[i]<desc.cluster_count**

Usage example

Training

```
kmeans::model<> run_training(const table& data,
                           const table& initial_centroids) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(10)
      .set_max_iteration_count(50)
      .set_accuracy_threshold(1e-4);

   const auto result = train(kmeans_desc, data, initial_centroids);

   print_table("labels", result.get_labels());
   print_table("centroids", result.get_model().get_centroids());
   print_value("objective", result.get_objective_function_value());

   return result.get_model();
}
```

Inference

```
table run_inference(const kmeans::model<>& model,
                    const table& new_data) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(model.get_cluster_count());

   const auto result = infer(kmeans_desc, model, new_data);

   print_table("labels", result.get_labels());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_kmeans_lloyd_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_lloyd_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- kmeans_batch.py

K-Means initialization

The K-Means initialization algorithm receives *n* feature vectors as input and chooses *k* initial centroids. After initialization, K-Means algorithm uses the initialization result to partition input data into *k* clusters.

Operation	Computational methods	Programming Interface		
Computing	Dense	compute(…)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: K-Means Initialization.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::kmeans_init` namespace and be available via inclusion of the `oneapi/dal/algo/kmeans_init.hpp` header file.

Descriptor

template<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an implementation of K-Means Initialization algorithm. - **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::init**.

Constructors

descriptor(std::int64_tcluster_count=2)

Creates a new instance of the class with the given `cluster_count`.

Properties

auto&seed

Getter & Setter	`template <typename M = Method, typename None = detail::v1::enable_if_not_default_dense<M>> auto & get_seed() consttemplate <typename M = Method, typename None = detail::v1::enable_if_not_default_dense<M>> auto & set_seed(std::int64_t value)`

std::int64_tcluster_count

The number of clusters k. **Default value**: 2.

Getter & Setter	`std::int64_t get_cluster_count() constauto & set_cluster_count(std::int64_t value)`
Invariants	**cluster_count>0**

Method tags

***struct*dense**

Tag-type that denotes dense computational method.

***struct*parallel_plus_dense**

***struct*plus_plus_dense**

***struct*random_dense**

***using*by_default=dense**

Task tags

***struct*init**

Tag-type that parameterizes entities used for obtaining the initial K-Means centroids.

***using*by_default=init**

Alias tag-type for the initialization task.

Computing compute(...)

Input

***template<typename*Task=task::by_default>*class*compute_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::init**.

Constructors

compute_input(*const*table&data)

Creates a new instance of the class with the given `data`.

Properties

***const*table&data**

An $n \times p$ table with the data to be clustered, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

Result

***template<typename*Task=task::by_default>*class*compute_result**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **oneapi::dal::kmeans::task::clustering**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&centroids**

A $k \times p$ table with the initial centroids. Each row of the table stores one centroid. **Default value**: table{}.

Getter & Setter	`const table & get_centroids() constauto & set_centroids(const table &value)`

Operation

***template<typename*Descriptor>kmeans_init::compute_resultcompute(*const*Descriptor&desc, *const*kmeans_init::compute_input&input)**

Parameters	• **desc** – K-Means algorithm descriptor **kmeans_init::descriptor** • **input** – Input data for the computing operation
Preconditions	**input.data.has_data==*true*input.data.row_count==desc.cluster_count**

Postconditions	**result.centroids.has_data==*true*result.centroids.row_count==desc.cluster_countresult.centroids.column_count==input.data.column_count**

Examples

oneAPI DPC++

Batch Processing:

- dpc_kmeans_init_dense.cpp

oneAPI C++

Batch Processing:

- cpp_kmeans_init_dense.cpp

Covariance

This chapter describes programming interfaces of the covariance algorithm implemented in oneDAL:

- Covariance

Covariance

Covariance algorithm computes the following set of quantitative dataset characteristics:

- means
- covariance
- correlation

Operation	**Computational methods**	**Programming Interface**		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: Covariance.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::covariance` namespace and are available via inclusion of the `oneapi/dal/algo/covariance.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	• **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. • **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::dense**. • **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Properties

result_option_idresult_options

Choose which results should be computed and returned.

Getter & Setter	`result_option_id get_result_options() constauto & set_result_options(const result_option_id &value)`

Method tags

***struct*dense**

Tag-type that denotes dense computational method.

***using*by_default=dense**

Alias tag-type for the dense computational method.

Task tags

***struct*compute**

Tag-type that parameterizes entities that are used to compute statistics.

***using*by_default=compute**

Alias tag-type for the compute task.

Training compute(...)

Input

***template*<*typename*Task=task::by_default>*class*compute_input**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_input(*const*table&data)

Creates a new instance of the class with the given `data` property value.

Properties

***const*table&data**

An $n \times p$ table with the training data, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &value)`

Result

***template*<*typename*Task=task::by_default>*class*compute_result**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&cor_matrix**

The correlation matrix. **Default value**: table{}.

Getter & Setter	`const table & get_cor_matrix() constauto & set_cor_matrix(const table &value)`

***const*table&cov_matrix**

The covariance matrix. **Default value**: table{}.

Getter & Setter	`const table & get_cov_matrix() constauto & set_cov_matrix(const table &value)`

***const*table&means**

Means. **Default value**: table{}.

Getter & Setter	`const table & get_means() constauto & set_means(const table &value)`

***const*result_option_id&result_options**

Result options that indicates availability of the properties. **Default value**: default_result_options<Task>.

Getter & Setter	`const result_option_id & get_result_options() constauto & set_result_options(const result_option_id &value)`

Operation

***template<typename*Descriptor>covariance::compute_resultcompute(*const*Descriptor&desc, *const*covariance::compute_input&input)**

Parameters	- **desc** – Covariance algorithm descriptor **covariance::descriptor** - **input** – Input data for the computing operation
Preconditions	**input.data.is_empty==*false***

Decomposition

This chapter describes programming interfaces of the decomposition algorithms implemented in oneDAL:

- Principal Components Analysis (PCA)

Principal Components Analysis (PCA)

Principal Component Analysis (PCA) is an algorithm for exploratory data analysis and dimensionality reduction. PCA transforms a set of feature vectors of possibly correlated features to a new set of uncorrelated features, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Operation	Computational methods		Programming Interface		
Training	Covariance	SVD	train(...)	train_input	train_result
Inference	Covariance	SVD	infer(...)	infer_input	infer_result

Mathematical formulation

Refer to Developer Guide: Principal Components Analysis.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::pca` namespace and be available via inclusion of the `oneapi/dal/algo/pca.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	• **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. • **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::cov** or **method::svd**. • **Task** – Tag-type that specifies type of the problem to solve. Can be **task::dim_reduction**.

Constructors

descriptor(std::int64_tcomponent_count=0)

Creates a new instance of the class with the given `component_count` property value.

Properties

std::int64_tcomponent_count

The number of principal components r. If it is zero, the algorithm computes the eigenvectors for all features, $r = p$. **Default value**: 0.

Getter & Setter	`std::int64_t get_component_count() constauto & set_component_count(std::int64_t value)`
Invariants	**component_count>=0**

booldeterministic

Specifies whether the algorithm applies the sign-flip technique. If it is **true**, the directions of the eigenvectors must be deterministic. **Default value**: true.

Getter & Setter	`bool get_deterministic() constauto & set_deterministic(bool value)`

result_option_idresult_options

Choose which results should be computed and returned.

Getter & Setter	`result_option_id get_result_options() constauto & set_result_options(const result_option_id &value)`

Method tags

*struct*cov

Tag-type that denotes Covariance computational method.

*struct*precomputed

*struct*svd

Tag-type that denotes SVD computational method.

*using*by_default=cov

Alias tag-type for Covariance computational method.

Task tags

*struct*dim_reduction

Tag-type that parameterizes entities used for solving dimensionality reduction problem.

*using*by_default=dim_reduction

Alias tag-type for dimensionality reduction task.

Model

template<*typename*Task=task::by_default>*class*model

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::dim_reduction**.

Constructors

model()

Creates a new instance of the class with the default property values.

Properties

*const*table&eigenvectors

An $r \times p$ table with the eigenvectors. Each row contains one eigenvector. **Default value**: table{}.

Getter & Setter	`const table & get_eigenvectors() constauto & set_eigenvectors(const table &value)`

Training train(...)

Input

***template*<*typename*Task=task::by_default>*class*train_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::dim_reduction**.

Constructors

train_input(*const*table&data)

Creates a new instance of the class with the given `data` property value.

Properties

***const*table&data**

An $n \times p$ table with the training data, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

Result

***template*<*typename*Task=task::by_default>*class*train_result**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::dim_reduction**.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Public Methods

const*table&get_eigenvectors()*const

An $r \times p$ table with the eigenvectors. Each row contains one eigenvector.

Properties

***const*table&eigenvalues**

A $1 \times r$ table that contains the eigenvalues for for the first `r` features. **Default value**: table{}.

Getter & Setter	`const table & get_eigenvalues() constauto & set_eigenvalues(const table &value)`

***const*result_option_id&result_options**

Result options that indicates availability of the properties. **Default value**: default_result_options<Task>.

Getter & Setter	`const result_option_id & get_result_options() constauto & set_result_options(const result_option_id &value)`

_const_table&means

A $1 \times r$ table that contains the mean values for the first `r` features. **Default value**: table{}.

Getter & Setter	`const table & get_means() constauto & set_means(const table &value)`

_const_table&variances

A $1 \times r$ table that contains the variances for the first `r` features. **Default value**: table{}.

Getter & Setter	`const table & get_variances() constauto & set_variances(const table &value)`

_const_model<Task>&model

The trained PCA model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Operation

***template*<*typename*Descriptor>pca::train_resulttrain(*const*Descriptor&desc, *const*pca::train_input&input)**

Parameters	• **desc** – PCA algorithm descriptor **pca::descriptor** • **input** – Input data for the training operation
Preconditions	**input.data.has_data==*true*input.data.column_count>=desc.component_count**
Postconditions	**result.means.row_count==1result.means.column_count==desc.component_countresult.variances.row_count==1result.variances.column_count==desc.component_countresult.variances[i]>=0.0result.eigenvalues.row_count==1result.eigenvalues.column_count==desc.component_countresult.model.eigenvectors.row_count==1result.model.eigenvectors.column_count==desc.component_count**

Inference infer(...)

Input

***template*<*typename*Task=task::by_default>*class*infer_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::dim_reduction**.

Constructors

infer_input(*const*model<Task>&trained_model, *const*table&data)

Creates a new instance of the class with the given `model` and `data` property values.

Properties

*const*table&data

The dataset for inference X'. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &value)`

*const*model<Task>&model

The trained PCA model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Result

*template<typename*Task=task::by_default>*class*infer_result

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::dim_reduction**.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Properties

*const*table&transformed_data

An $n \times r$ table that contains data projected to the r principal components. **Default value**: table{}.

Getter & Setter	`const table & get_transformed_data() constauto & set_transformed_data(const table &value)`

Operation

*template<typename*Descriptor>pca::infer_resultinfer(*const*Descriptor&desc, *const*pca::infer_input&input)

Parameters	• **desc** – PCA algorithm descriptor **pca::descriptor** • **input** – Input data for the inference operation
Preconditions	**input.data.has_data==*true*input.model.eigenvectors.row_count==desc.component_countinput.model.eigenvectors.column_count==input.data.column_count**
Postconditions	**result.transformed_data.row_count==input.data.row_countresult.transformed_data.column_count==desc.component_count**

Usage example

Training

```
pca::model<> run_training(const table& data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(5)
      .set_deterministic(true);

   const auto result = train(pca_desc, data);

   print_table("means", result.get_means());
   print_table("variances", result.get_variances());
   print_table("eigenvalues", result.get_eigenvalues());
   print_table("eigenvectors", result.get_eigenvectors());

   return result.get_model();
}
```

Inference

```
table run_inference(const pca::model<>& model,
                  const table& new_data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(model.get_component_count());

   const auto result = infer(pca_desc, model, new_data);

   print_table("labels", result.get_transformed_data());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_pca_cor_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_pca_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- pca_batch.py

Ensembles

This chapter describes programming interfaces of the ensemble algorithms implemented in oneDAL:

- Decision Forest Classification and Regression (DF)

Decision Forest Classification and Regression (DF)

Decision Forest (DF) classification and regression algorithms are based on an ensemble of tree-structured classifiers, which are known as decision trees. Decision forest is built using the general technique of bagging, a bootstrap aggregation, and a random choice of features. For more details, see [Breiman84] and [Breiman2001].

Operation	Computational methods	Programming Interface			
Training	Dense	Hist	train(…)	train_input	train_result
Inference	Dense	Hist	infer(…)	infer_input	infer_result

Mathematical formulation

Refer to Developer Guide: Decision Forest Classification and Regression.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::decision_forest` namespace and are available via inclusion of the `oneapi/dal/algo/decision_forest.hpp` header file.

Enum classes

error_metric_mode

error_metric_mode::none	Do not compute error metric.
error_metric_mode::out_of_bag_error	Train produces $1imes1$ table with cumulative prediction error for out of bag observations.
error_metric_mode::out_of_bag_error_per_observation	Train produces $nimes1$ table with prediction error for out-of-bag observations.

variable_importance_mode

variable_importance_mode::none	Do not compute variable importance.
variable_importance_mode::mdi	Mean Decrease Impurity. Computed as the sum of weighted impurity decreases for all nodes where the variable is used, averaged over all trees in the forest.
variable_importance_mode::mda_raw	Mean Decrease Accuracy (permutation importance). For each tree, the prediction error on the out-of-bag portion of the data is computed (error rate for classification, MSE for regression). The same is done after permuting each predictor variable. The difference between the two are then averaged over all trees.
variable_importance_mode::mda_scaled	Mean Decrease Accuracy (permutation importance). This is MDA_Raw value scaled by its standard deviation.

infer_mode

infer_mode::class_labels	Infer produces a “math:**n times 1** table with the predicted labels.
infer_mode::class_responses	deprecated
infer_mode::class_probabilities	Infer produces $n \times c$ table with the predicted class probabilities for each observation.

voting_mode

voting_mode::weighted	The final prediction is combined through a weighted majority voting.

voting_mode::unweighted	The final prediction is combined through a simple majority voting.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	• **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. • **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::dense** or **method::hist**. • **Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::regression**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Properties

error_metric_modeerror_metric_mode

The error metric mode. **Default value**: error_metric_mode::none.

Getter & Setter	`error_metric_mode get_error_metric_mode() constauto & set_error_metric_mode(error_metric_mode value)`

std::int64_tmax_bins

The maximal number of discrete bins to bucket continuous features. Used with **method::hist** split-finding method only. Increasing the number results in higher computation costs. **Default value**: 256.

Getter & Setter	`std::int64_t get_max_bins() constauto & set_max_bins(std::int64_t value)`
Invariants	**max_bins>1**

std::int64_tmax_tree_depth

The maximal depth of the tree. If 0, then nodes are expanded until all leaves are pure or until all leaves contain less or equal to min observations in leaf node samples. **Default value**: 0.

Getter & Setter	`std::int64_t get_max_tree_depth() constauto & set_max_tree_depth(std::int64_t value)`

std::int64_tseed

Seed for the random numbers generator used by the algorithm.

Getter & Setter	`std::int64_t get_seed() constauto & set_seed(std::int64_t value)`
Invariants	**tree_count>0**

doubleimpurity_threshold

The impurity threshold, a node will be split if this split induces a decrease of the impurity greater than or equal to the input value. **Default value**: 0.0.

Getter & Setter	`double get_impurity_threshold() constauto & set_impurity_threshold(double value)`
Invariants	**impurity_threshold>=0.0**

variable_importance_modevariable_importance_mode

The variable importance mode. **Default value**: variable_importance_mode::none.

Getter & Setter	`variable_importance_mode get_variable_importance_mode() constauto & set_variable_importance_mode(variable_importance_mode value)`

boolbootstrap

The bootstrap mode, if true, the training set for a tree is a bootstrap of the whole training set, if False, the whole dataset is used to build each tree. **Default value**: true.

Getter & Setter	`bool get_bootstrap() constauto & set_bootstrap(bool value)`

std::int64_tmin_bin_size

The minimal number of observations in a bin. Used with **method::hist** split-finding method only. **Default value**: 5.

Getter & Setter	`std::int64_t get_min_bin_size() constauto & set_min_bin_size(std::int64_t value)`
Invariants	**min_bin_size>0**

std::int64_ttree_count

The number of trees in the forest. **Default value**: 100.

Getter & Setter	`std::int64_t get_tree_count() constauto & set_tree_count(std::int64_t value)`
Invariants	**tree_count>0**

doublemin_impurity_decrease_in_split_node

The min impurity decrease in a split node is a threshold for stopping the tree growth early. A node will be split if its impurity is above the threshold, otherwise it is a leaf. **Default value**: 0.0.

Getter & Setter	`double get_min_impurity_decrease_in_split_node() constauto & set_min_impurity_decrease_in_split_node(double value)`
Invariants	**min_impurity_decrease_in_split_node>=0.0**

std::int64_tmin_observations_in_leaf_node

The minimal number of observations in a leaf node. **Default value**: 1 for classification, 5 for regression.

Getter & Setter

```
std::int64_t get_min_observations_in_leaf_node() constauto &
set_min_observations_in_leaf_node(std::int64_t value)
```

Invariants

min_observations_in_leaf_node>0

voting_modevoting_mode

The voting mode. Used with **task::classification** only.

Getter & Setter

```
template <typename T = Task, typename None =
detail::enable_if_classification_t<T>> voting_mode
get_voting_mode() consttemplate <typename T = Task, typename None
= detail::enable_if_classification_t<T>> auto &
set_voting_mode(voting_mode value)
```

doubleobservations_per_tree_fraction

The fraction of observations per tree. **Default value**: 1.0.

Getter & Setter

```
double get_observations_per_tree_fraction() constauto &
set_observations_per_tree_fraction(double value)
```

Invariants

observations_per_tree_fraction>0.0observations_per_tree_fraction<=1.0

infer_modeinfer_mode

The infer mode. Used with **task::classification** only.

Getter & Setter

```
template <typename T = Task, typename None =
detail::enable_if_classification_t<T>> infer_mode
get_infer_mode() consttemplate <typename T = Task, typename None
= detail::enable_if_classification_t<T>> auto &
set_infer_mode(infer_mode value)
```

std::int64_tmin_observations_in_split_node

The minimal number of observations in a split node. **Default value**: 2.

Getter & Setter

```
std::int64_t get_min_observations_in_split_node() constauto &
set_min_observations_in_split_node(std::int64_t value)
```

Invariants

min_observations_in_split_node>1

std::int64_tclass_count

The class count. Used with **task::classification** only. **Default value**: 2.

Getter & Setter

```
template <typename T = Task, typename None =
detail::enable_if_classification_t<T>> std::int64_t
get_class_count() consttemplate <typename T = Task, typename None
= detail::enable_if_classification_t<T>> auto &
set_class_count(std::int64_t value)
```

boolmemory_saving_mode

The memory saving mode. **Default value**: false.

Getter & Setter

```
bool get_memory_saving_mode() constauto &
set_memory_saving_mode(bool value)
```

std::int64_tfeatures_per_node

The number of features to consider when looking for the best split for a node. **Default value**: task::classification ? sqrt(p) : p/3, where p is the total number of features.

Getter & Setter

```
std::int64_t get_features_per_node() constauto &
set_features_per_node(std::int64_t value)
```

doublemin_weight_fraction_in_leaf_node

The min weight fraction in a leaf node. The minimum weighted fraction of the total sum of weights (of all input observations) required to be at a leaf node. **Default value**: 0.0.

Getter & Setter

```
double get_min_weight_fraction_in_leaf_node() constauto &
set_min_weight_fraction_in_leaf_node(double value)
```

Invariants

min_weight_fraction_in_leaf_node>=0.0min_weight_fraction_in_leaf_node<=0.5

std::int64_tmax_leaf_nodes

The maximal number of the leaf nodes. If 0, the number of leaf nodes is not limited. **Default value**: 0.

Getter & Setter

```
std::int64_t get_max_leaf_nodes() constauto &
set_max_leaf_nodes(std::int64_t value)
```

Method tags

*struct*dense

Tag-type that denotes dense computational method.

*struct*hist

Tag-type that denotes hist computational method.

*using*by_default=dense

Alias tag-type for dense computational method.

Task tags

*struct*classification

Tag-type that parameterizes entities used for solving classification problem.

*struct*regression

Tag-type that parameterizes entities used for solving regression problem.

*using*by_default=classification

Alias tag-type for classification task.

Model

template<*typename*Task=task::by_default>*class*model

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::classification** or **task::regression**.

Constructors

model()

Creates a new instance of the class with the default property values.

Public Methods

std::int64_tget_tree_count()*const*

The number of trees in the forest.

template<*typename*T=Task,*typename*None=detail::enable_if_classification_t<T>>std::int64_tget_class_count()*const*

The class count. Used with **oneapi::dal::decision_forest::task::classification** only.

template<*typename*Visitor>voidtraverse_depth_first(std::int64_ttree_idx, Visitor&&visitor)*const*

Performs Depth First Traversal of i-th tree.

Parameters	- **tree_idx** – Index of the tree to traverse. - **visitor** – This functor gets notified when tree nodes are visited, via corresponding operators: bool operator()(const decision_forest::split_node_info<Task>&) bool operator()(const decision_forest::leaf_node_info<Task>&).

template<*typename*Visitor>voidtraverse_breadth_first(std::int64_ttree_idx, Visitor&&visitor)*const*

Performs Breadth First Traversal of i-th tree.

Parameters	- **tree_idx** – Index of the tree to traverse. - **visitor** – This functor gets notified when tree nodes are visited, via corresponding operators: bool operator()(const decision_forest::split_node_info<Task>&) bool operator()(const decision_forest::leaf_node_info<Task>&).

Training train(...)

Input

_template<typename_Task=task::by_default>_class_train_input

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::regression**.

Constructors

train_input(_const_table&data, _const_table&responses, _const_table&weights=table{})

Creates a new instance of the class with the given `data`, `responses` and `weights` property values.

Properties

_const_table&data

The training set X. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &value)`

_const_table&weights

The vector of weights w for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_weights() constauto & set_weights(const table &value)`

_const_table&responses

Vector of responses y for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

_const_table&labels

Vector of labels y for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_labels() constauto & set_labels(const table &value)`

Result

_template<typename_Task=task::by_default>_class_train_result

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::regression**.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Properties

_const_table&oob_err

A 1×1 table containing cumulative out-of-bag error value. Computed when `error_metric_mode` set with `error_metric_mode::out_of_bag_error`. **Default value**: table{}.

Getter & Setter	`const table & get_oob_err() constauto & set_oob_err(const table &value)`

***const*model<Task>&model**

The trained Decision Forest model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

***const*table&var_importance**

A $1 \times p$ table containing variable importance value for each feature. Computed when **variable_importance_mode!=variable_importance_mode::none**. **Default value**: table{}.

Getter & Setter	`const table & get_var_importance() constauto & set_var_importance(const table &value)`

***const*table&oob_err_per_observation**

A $n \times 1$ table containing out-of-bag error value per observation. Computed when `error_metric_mode` set with `error_metric_mode::out_of_bag_error_per_observation`. **Default value**: table{}.

Getter & Setter	`const table & get_oob_err_per_observation() constauto & set_oob_err_per_observation(const table &value)`

Operation

***template*<*typename*Descriptor>decision_forest::train_resulttrain(*const*Descriptor&desc, *const*decision_forest::train_input&input)**

Parameters	- **desc** – Decision Forest algorithm descriptor **decision_forest::descriptor**. - **input** – Input data for the training operation
Preconditions	**input.data.is_empty==*false*input.labels.is_empty==*false*input.labels.column_count==1input.data.row_count==input.labels.row_countdesc.get_bootstrap()==*true*||(desc.get_bootstrap()==*false*&&desc.get_variable_importance_mode()!=variable_importance_mode::mda_raw&&desc.get_variable_importance_mode()!=variable_importance_mode::mda_scaled)desc.get_bootstrap()==*true*||(desc.get_bootstrap()==*false*&&desc.get_error_metric_mode()==error_metric_mode::none)**

Inference infer(...)

Input

***template<typename*Task=task::by_default>*class*infer_input**

Template Parameters — **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::classification** or **task::regression**.

Constructors

infer_input(*const*model<Task>&trained_model, *const*table&data)

Creates a new instance of the class with the given `model` and `data` property values.

Properties

***const*table&data**

The dataset for inference X'. **Default value**: table{}.

Getter & Setter

```
const table & get_data() constauto & set_data(const table &value)
```

***const*model<Task>&model**

The trained Decision Forest model. **Default value**: model<Task>{}.

Getter & Setter

```
const model< Task > & get_model() constauto & set_model(const
model< Task > &value)
```

Result

***template<typename*Task=task::by_default>*class*infer_result**

Template Parameters — **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::classification** or **task::regression**.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&labels**

The $n \times 1$ table with the predicted labels. **Default value**: table{}.

Getter & Setter

```
const table & get_labels() constauto & set_labels(const table
&value)
```

***const*table&probabilities**

A $n \times c$ table with the predicted class probabilities for each observation.

Getter & Setter

```
template <typename T = Task, typename None =
detail::enable_if_classification_t<T>> const table &
get_probabilities() consttemplate <typename T = Task, typename
None = detail::enable_if_classification_t<T>> auto &
set_probabilities(const table &value)
```

***const*table&responses**

The $n \times 1$ table with the predicted responses. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

Operation

***template<typename*Descriptor>decision_forest::infer_resultinfer(*const*Descriptor&desc, *const*decision_forest::infer_input&input)**

Parameters	• **desc** – Decision Forest algorithm descriptor **decision_forest::descriptor**. • **input** – Input data for the inference operation
Preconditions	**input.data.is_empty==*false***

Graph

This chapter describes programming interfaces of the graph algorithms implemented in oneDAL:

- Subgraph Isomorphism
- Connected Components

Subgraph Isomorphism

Subgraph Isomorphism algorithm receives a target graph *G* and a pattern graph *H* as input and searches the target graph for subgraphs that are isomorphic to the pattern graph. The algorithm returns the mappings of the pattern graph vertices onto the target graph vertices.

Operation	Computational methods	Programming Interface		
Computing	fast	graph_matching(...)	graph_matching_input	graph_matching_result

Mathematical formulation

Refer to Developer Guide: Subgraph Isomorphism.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::preview::subgraph_isomorphism` namespace and available via inclusion of the `oneapi/dal/algo/subgraph_isomorphism.hpp` header file.

Descriptor

***template<typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default,*typename*Allocator=std::allocator<char>>*class*descriptor**

Template Parameters	• **Float** – This parameter is not used for Subgraph Isomorphism algorithm. • **Method** – Tag-type that specifies the implementation of the algorithm. Can be **method::fast**.

- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.
- **Allocator** – Custom allocator for all memory management inside the algorithm.

Constructors

descriptor(Allocatorallocator=std::allocator<char>())

Public Methods

Allocatorget_allocator()*const*

Returns a copy of the allocator used in the algorithm for internal memory management.

Properties

std::int64_tmax_match_count

The maximum number of matchings to search in Subgraph Isomorphism computation.

Getter & Setter	`std::int64_t get_max_match_count() constauto & set_max_match_count(std::int64_t max_match_count)`

boolsemantic_match

The flag that specifies if semantic search is required in Subgraph Isomorphism computation. If true, vertex labels are considered.

Getter & Setter	`bool get_semantic_match() constauto & set_semantic_match(bool semantic_match)`

kindkind

The kind of subgraph to be isomorphic to the pattern graph. Can be **kind::induced** or **kind::non_induced**.

Getter & Setter	`kind get_kind() constauto & set_kind(kind value)`

Method tags

***struct*fast**

Tag-type that denotes fast computational method.

***using*by_default=fast**

Alias tag-type for fast computational method.

Task tags

***struct*compute**

Tag-type that parameterizes entities that are used for Subgraph Isomorphism algorithm.

***using*by_default=compute**

Alias tag-type for the compute task.

Enum classes

***enumclass*kind**

kind::induced	Search for an induced subgraph isomorphic to the pattern graph. All existing and non-existing edges in a subgraph are considered.
kind::non_induced	Search for a non-induced subgraph isomorphic to the pattern graph. Only existing edges in a subgraph are considered.

Computing preview::graph_matching(...)

Input

***template<typename*Graph,*typename*Task=task::compute>*class*graph_matching_input**

Template Parameters

- **Graph** – The type of the input graph.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

graph_matching_input(*const*Graph&target_graph, *const*Graph&pattern_graph)

Constructs the algorithm input initialized with the target and pattern graphs.

Parameters

- **target_graph** – The input target (bigger) graph.
- **pattern_graph** – The input pattern (smaller) graph.

Properties

***const*Graph&pattern_graph**

Returns the constant reference to the input pattern graph.

Getter & Setter

```
const Graph & get_pattern_graph() constconst auto &
set_pattern_graph(const Graph &pattern_graph)
```

***const*Graph&target_graph**

Returns the constant reference to the input target graph.

Getter & Setter

```
const Graph & get_target_graph() constconst auto &
set_target_graph(const Graph &target_graph)
```

Result

***template<typename*Task=task::by_default>*class*graph_matching_result**

Constructors

graph_matching_result()

Constructs the empty result.

Properties

***const*table&vertex_match**

Returns the table of size [match_count x pattern_vertex_count] with matchings of the pattern graph in the target graph. Each row of the table contain ids of vertices in target graph sorted by pattern vertex ids. I.e. j-th element of i-th row contain id of target graph vertex which was matched with j-th vertex of pattern graph in i-th match.

Getter & Setter	`const table & get_vertex_match() constauto &`
	`set_vertex_match(const table &value)`

std::int64_tmatch_count

The number pattern matches in the target graph.

Getter & Setter	`std::int64_t get_match_count() constauto &`
	`set_match_count(std::int64_t value)`

Operation

***template<typename*Graph,*typename*Descriptor>subgraph_isomorphism::graph_matching_result**`p review::`**graph_matching(*const*Descriptor&desc, *const*Graph&target, *const*Graph&pattern)**

Parameters	• **desc** – Subgraph Isomorphism algorithm descriptor **subgraph_isomorphism::descriptor** • **target** – Target (big) graph • **pattern** – Pattern (small) graph

Examples

oneAPI C++

Batch Processing:

- cpp_subgraph_isomorphism_batch.cpp

Connected Components

Connected components algorithm receives an undirected graph *G* as an input and searches for connected components in *G*. For each vertex in *G*, the algorithm returns the label of the component this vertex belongs to. The result of the algorithm is a set of labels for all vertices in *G*.

Operation	Computational methods	Programming Interface		
Computing	afforest	vertex_partitioning(...)	vertex_partitioning_input	vertex_partitioning_result

Mathematical formulation

Refer to Developer Guide: Connected Components.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::preview::connected_components` namespace and available via inclusion of the `oneapi/dal/algo/connected_components.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default,*typename*Allocator=std::allocator<char>>*class*descriptor**

Template Parameters

- **Float** – This parameter is not used for Connected Components algorithm.
- **Method** – Tag-type that specifies the implementation of the algorithm. Can be **method::afforest**.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::vertex_partitioning**.
- **Allocator** – Custom allocator for all memory management inside the algorithm.

Constructors

descriptor(*const*Allocator&allocator=std::allocator<char>())

Public Methods

Allocatorget_allocator()*const*

Returns a copy of the allocator used in the algorithm for internal memory management.

Method tags

***struct*afforest**

Tag-type that denotes Afforest computational method.

***using*by_default=afforest**

Alias tag-type for Afforest computational method.

Task tags

***struct*vertex_partitioning**

Tag-type that parameterizes entities that are used for Connected Components algorithm.

***using*by_default=vertex_partitioning**

Alias tag-type for the vertex partitioning task.

Computing preview::vertex_partitioning(...)

Input

***template*<*typename*Graph,*typename*Task=task::by_default>*class*vertex_partitioning_input**

Template Parameters **Graph** – Type of the input graph.

Constructors

vertex_partitioning_input(*const*Graph&g)

Constructs the algorithm input initialized with the graph.

Parameters **g** – The input graph.

Properties

***const*Graph&graph**

Returns the constant reference to the input graph.

Getter & Setter	`const Graph & get_graph() constauto & set_graph(const Graph &g)`

Result

***template*<*typename*Task=task::by_default>*class*vertex_partitioning_result**

Constructors

vertex_partitioning_result()

Constructs the empty result.

Properties

***const*table&labels**

The table of size [vertex_count x 1] with computed component ids for each vertex.

Getter & Setter	`const table & get_labels() constauto & set_labels(const table &value)`

std::int64_tcomponent_count

The number of connected components.

Getter & Setter	`std::int64_t get_component_count() constauto & set_component_count(std::int64_t value)`

Operation

***template*<*typename*Graph,*typename*Descriptor>connected_components::vertex_partitioning_result`preview::`vertex_partitioning(*const*Descriptor&desc, *const*Graph&g)**

Parameters	- **desc** – Connected Components algorithm descriptor **connected_components::descriptor** - **g** – Input graph

Examples

oneAPI C++

Batch Processing:

- cpp_connected_components_batch.cpp

Kernel Functions

This chapter describes programming interfaces of the kernel functions implemented in oneDAL:

- Linear kernel
- Polynomial kernel
- Radial Basis Function (RBF) kernel
- Sigmoid kernel

Linear kernel

The linear kernel is the simplest kernel function for pattern analysis.

Operation	Computational methods	Programming Interface		
dense	dense	compute(…)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: Linear kernel.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::linear_kernel` namespace and are available via inclusion of the `oneapi/dal/algo/linear_kernel.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::dense**. - **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Properties

doubleshift

The coefficient b of the linear kernel. **Default value**: 0.0.

Getter & Setter	`double get_shift() constauto & set_shift(double value)`

doublescale

The coefficient k of the linear kernel. **Default value**: 1.0.

Getter & Setter	`double get_scale() constauto & set_scale(double value)`

Method tags

***struct*dense**

***using*by_default=dense**

Alias tag-type for the dense method.

Task tags

*struct*compute

Tag-type that parameterizes entities that are used to compute statistics, distance, and so on.

*using*by_default=compute

Alias tag-type for the compute task.

Training compute(...)

Input

template<*typename*Task=task::by_default>*class*compute_input

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_input(*const*table&x, *const*table&y)

Creates a new instance of the class with the given `x` and `y`.

Properties

*const*table&y

An $m \times p$ table with the data y, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_y() constauto & set_y(const table &data)`

*const*table&x

An $n \times p$ table with the data x, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_x() constauto & set_x(const table &data)`

Result

template<*typename*Task=task::by_default>*class*compute_result

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

*const*table&values

A $n \times m$ table with the result kernel functions. **Default value**: table{}.

Getter & Setter	`const table & get_values() constauto & set_values(const table &value)`

Operation

***template<typename*Descriptor>linear_kernel::compute_resultcompute(*const*Descriptor&desc, *const*linear_kernel::compute_input&input)**

Parameters	- **desc** – Linear Kernel algorithm descriptor **linear_kernel::descriptor**. - **input** – Input data for the computing operation
Preconditions	**input.data.is_empty==*false***

Polynomial kernel

The Polynomial kernel is a popular kernel function used in kernelized learning algorithms. It represents the similarity of training samples in a feature space of polynomials of the original data and allows to fit non-linear models.

Operation	Computational methods	Programming Interface		
dense	dense	compute(…)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: Polynomial kernel.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::polynomial_kernel` namespace and are available via inclusion of the `oneapi/dal/algo/polynomial_kernel.hpp` header file.

Descriptor

***template<typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::dense**. - **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Properties

doubleshift

The coefficient b of the polynomial kernel. **Default value**: 0.0.

Getter & Setter	`double get_shift() constauto & set_shift(double value)`

std::int64_tdegree

The degree d of the polynomial kernel. **Default value**: 3.

Getter & Setter	`std::int64_t get_degree() constauto & set_degree(std::int64_t value)`

doublescale

The coefficient k of the polynomial kernel. **Default value**: 1.0.

Getter & Setter	`double get_scale() constauto & set_scale(double value)`

Method tags

***struct*dense**

***using*by_default=dense**

Alias tag-type for the dense method.

Task tags

***struct*compute**

Tag-type that parameterizes entities that are used to compute statistics, distance, and so on.

***using*by_default=compute**

Alias tag-type for the compute task.

Training compute(...)

Input

***template*<*typename*Task=task::by_default>*class*compute_input**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_input(*const*table&x, *const*table&y)

Creates a new instance of the class with the given `x` and `y`.

Properties

***const*table&y**

An $m \times p$ table with the data y, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_y() constauto & set_y(const table &data)`

***const*table&x**

An $n \times p$ table with the data x, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_x() constauto & set_x(const table &data)`

Result

***template*<*typename*Task=task::by_default>*class*compute_result**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&values**

A $n \times m$ table with the result kernel functions. **Default value**: table{}.

Getter & Setter	`const table & get_values() constauto & set_values(const table &value)`

Operation

***template*<*typename*Descriptor>polynomial_kernel::compute_resultcompute(*const*Descriptor&desc, *const*polynomial_kernel::compute_input&input)**

Parameters	• **desc** – Polynomial Kernel algorithm descriptor **polynomial_kernel::descriptor** • **input** – Input data for the computing operation
Preconditions	**input.x.is_empty==*false*input.y.is_empty==*false*input.x.column_count==input.y.column_count**
Postconditions	**result.values.has_data==*true*result.values.row_count==input.x.row_countresult.values.column_count==input.y.row_count**

Radial Basis Function (RBF) kernel

The Radial Basis Function (RBF) kernel is a popular kernel function used in kernelized learning algorithms.

Operation	**Computational methods**	**Programming Interface**		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: Radial Basis Function (RBF) kernel.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::rbf_kernel` namespace and are available via inclusion of the `oneapi/dal/algo/rbf_kernel.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::dense**. - **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Properties

doublesigma

The coefficient σ of the RBF kernel. **Default value**: 1.0.

Getter & Setter	`double get_sigma() constauto & set_sigma(double value)`

Method tags

***struct*dense**

***using*by_default=dense**

Task tags

***struct*compute**

Tag-type that parameterizes entities that are used to compute statistics, distance, and so on.

***using*by_default=compute**

Alias tag-type for the dense method.

Training compute(...)

Input

***template*<*typename*Task=task::by_default>*class*compute_input**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_input(*const*table&x, *const*table&y)

Creates a new instance of the class with the given `x` and `y`.

Properties

*const*table&y

An $m \times p$ table with the data y, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_y() constauto & set_y(const table &data)`

*const*table&x

An $n \times p$ table with the data x, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_x() constauto & set_x(const table &data)`

Result

*template<typename*Task=task::by_default>*class*compute_result

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

*const*table&values

A $n \times m$ table with the result kernel functions. **Default value**: table{}.

Getter & Setter	`const table & get_values() constauto & set_values(const table &value)`

Operation

*template<typename*Descriptor>rbf_kernel::compute_resultcompute(*const*Descriptor&desc, *const*rbf_kernel::compute_input&input)

Parameters	• **desc** – RBF Kernel algorithm descriptor **rbf_kernel::descriptor**. • **input** – Input data for the computing operation
Preconditions	**input.data.is_empty==*false***

Sigmoid kernel

The Sigmoid kernel is a popular kernel function used in kernelized learning algorithms.

Operation	Computational methods	Programming Interface

dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: Sigmoid kernel.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::sigmoid_kernel` namespace and are available via inclusion of the `oneapi/dal/algo/sigmoid_kernel.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	• **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. • **Method** – Tag-type that specifies the implementation of the algorithm. Can be **method::dense**. • **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Properties

doubleshift

The coefficient b of the sigmoid kernel. **Default value**: 0.0.

Getter & Setter	`double get_shift() constauto & set_shift(double value)`

doublescale

The coefficient k of the sigmoid kernel. **Default value**: 1.0.

Getter & Setter	`double get_scale() constauto & set_scale(double value)`

Method tags

***struct*dense**

***using*by_default=dense**

Alias tag-type for the dense method.

Task tags

***struct*compute**

Tag-type that parameterizes entities that are used to compute statistics, distance, and so on.

***using*by_default=compute**

Alias tag-type for the compute task.

Training compute(...)

Input

***template<typename*Task=task::by_default>*class*compute_input**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_input(*const*table&x, *const*table&y)

Creates a new instance of the class with the given `x` and `y`.

Properties

***const*table&y**

An $m \times p$ table with the data y, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_y() constauto & set_y(const table &data)`

***const*table&x**

An $n \times p$ table with the data x, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_x() constauto & set_x(const table &data)`

Result

***template<typename*Task=task::by_default>*class*compute_result**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&values**

An $n \times m$ table with the result kernel functions. **Default value**: table{}.

Getter & Setter	`const table & get_values() constauto & set_values(const table &value)`

Operation

***template<typename*Descriptor>sigmoid_kernel::compute_resultcompute(*const*Descriptor&desc, *const*sigmoid_kernel::compute_input&input)**

Parameters	- **desc** – Sigmoid Kernel algorithm descriptor **sigmoid_kernel::descriptor** - **input** – Input data for the computing operation
Preconditions	**input.x.is_empty==*false*input.y.is_empty==*false*input.x.column_count==input.y.column_count**
Postconditions	**result.values.has_data==*true*result.values.row_count==input.x.row_countresult.values.column_count==input.y.row_count**

Nearest Neighbors (kNN)

This chapter describes programming interfaces of the nearest neighbors algorithms implemented in oneDAL:

- k-Nearest Neighbors Classification (k-NN)

k-Nearest Neighbors Classification (k-NN)

k-NN classification and search algorithms are based on finding the *k* nearest observations to the training set. For classification, the problem is to infer the class of a new feature vector by computing the majority vote of its *k* nearest observations from the training set. For search, the problem is to infer *k* nearest observations from the training set to a new feature vector. The nearest observations are computed based on the chosen distance metric.

Operation	Computational methods	Programming Interface			
Training	Brute-force	k-d tree	train(…)	train_input	train_result
Inference	Brute-force	k-d tree	infer(…)	infer_input	infer_result

Mathematical formulation

Refer to Developer Guide: k-Nearest Neighbors Classification.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::knn` namespace and be available via inclusion of the `oneapi/dal/algo/knn.hpp` header file.

Enum classes

***enumclass*voting_mode**

voting_mode::uniform	Uniform weights for neighbors for prediction voting.
voting_mode::distance	Weight neighbors by the inverse of their distance.

Result options

***class*result_option_id**

Public Methods

constexpr*result_option_id()=*default

***constexpr*result_option_id(*const*result_option_id_base&base)**

Descriptor

***template<typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default,*typename*Distance=oneapi::dal::minkowski_distance::descriptor<Float>>*class*descriptor**

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::brute_force** or **method::kd_tree**. - **Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification**, **task::regression**, or **task::search**. - **Distance** – The descriptor of the distance used for computations. Can be **minkowski_distance::descriptor** or **chebyshev_distance::descriptor**.

Constructors

descriptor(std::int64_tclass_count, std::int64_tneighbor_count)

Creates a new instance of the class with the given `class_count` and `neighbor_count` property values.

***template<typename*M=Method,*typename*None=detail::enable_if_brute_force_t<M>>descriptor(std::int64_tclass_count, std::int64_tneighbor_count, *const*distance_t&distance)**

Creates a new instance of the class with the given `class_count`, `neighbor_count` and `distance` property values. Used with **method::brute_force** only.

***template<typename*T=Task,*typename*None=detail::enable_if_not_classification_t<T>>descriptor(std::int64_tneighbor_count)**

Creates a new instance of the class with the given `neighbor_count` property value. Used with **task::search** and **task::regression** only.

***template<typename*T=Task,*typename*None=detail::enable_if_not_classification_t<T>>descriptor(std::int64_tneighbor_count, *const*distance_t&distance)**

Creates a new instance of the class with the given `neighbor_count` and `distance` property values. Used with **task::search** and **task::regression** only.

Properties

voting_modevoting_mode

The voting mode.

Getter & Setter	`voting_mode get_voting_mode() constauto &`
	`set_voting_mode(voting_mode value)`

result_option_idresult_options

Choose which results should be computed and returned.

Getter & Setter	`result_option_id get_result_options() constauto &`
	`set_result_options(const result_option_id &value)`

***const*distance_t&distance**

Choose distance type for calculations. Used with **method::brute_force** only.

Getter & Setter	`template <typename M = Method, typename None = detail::enable_if_brute_force_t<M>> const distance_t & get_distance() consttemplate <typename M = Method, typename None = detail::enable_if_brute_force_t<M>> auto & set_distance(const distance_t &dist)`

std::int64_tclass_count

The number of classes c.

Getter & Setter	`std::int64_t get_class_count() constauto & set_class_count(std::int64_t value)`
Invariants	**class_count>1**

std::int64_tneighbor_count

The number of neighbors k.

Getter & Setter	`std::int64_t get_neighbor_count() constauto & set_neighbor_count(std::int64_t value)`
Invariants	**neighbor_count>0**

Method tags

*struct*brute_force

Tag-type that denotes brute-force computational method.

*struct*kd_tree

Tag-type that denotes k-d tree computational method.

*using*by_default=brute_force

Alias tag-type for brute-force computational method.

Task tags

*struct*classification

Tag-type that parameterizes entities used for solving classification problem.

*struct*regression

Tag-type that parameterizes entities used for solving the regression problem.

*struct*search

Tag-type that parameterizes entities used for solving the search problem.

*using*by_default=classification

Alias tag-type for classification task.

Model

***template<typename*Task=task::by_default>*class*model**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification**, **task::search** and **task::regression**.

Constructors

model()

Creates a new instance of the class with the default property values.

Training train(...)

Input

***template<typename*Task=task::by_default>*class*train_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::search**.

Constructors

train_input(*const*table&data, *const*table&responses)

Creates a new instance of the class with the given `data` and `responses` property values.

train_input(*const*table&data)

Properties

***const*table&data**

The training set X. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

***const*table&responses**

Vector of responses y for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_responses() consttemplate <typename T = Task, typename None = detail::enable_if_classification_t<T>> auto & set_responses(const table &responses)`

***const*table&labels**

Vector of labels y for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_labels() consttemplate <typename T = Task, typename None = detail::enable_if_classification_t<T>> auto & set_labels(const table &value)`

Result

***template<typename*Task=task::by_default>*class*train_result**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::search**.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Properties

***const*model<Task>&model**

The trained k-NN model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Operation

***template<typename*Descriptor>knn::train_resulttrain(*const*Descriptor&desc, *const*knn::train_input&input)**

Parameters	• **desc** – k-NN algorithm descriptor **knn::descriptor** • **input** – Input data for the training operation
Preconditions	**input.data.has_data==*true*input.labels.has_data==*true*input.data.row_count==input.labels.row_countinput.labels.column_count==1input.labels[i]>=0input.labels[i]<desc.class_count**

Inference infer(...)

Input

***template<typename*Task=task::by_default>*class*infer_input**

Template Parameters	**Task** – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::search**.

Constructors

infer_input(*const*table&data, *const*model<Task>&model)

Creates a new instance of the class with the given `model` and `data` property values.

Properties

***const*table&data**

The dataset for inference X'. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

***const*model<Task>&model**

The trained k-NN model. **Default value**: model<Task>{}.

Getter & Setter

```
const model< Task > & get_model() constauto & set_model(const
model< Task > &m)
```

Result

template<_typename_Task=task::by_default>_class_infer_result

Template Parameters

Task – Tag-type that specifies type of the problem to solve. Can be **task::classification** or **task::search**.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Properties

_const_table&responses

The predicted responses. **Default value**: table{}.

Getter & Setter

```
const table & get_responses() consttemplate <typename T = Task,
typename None = detail::enable_if_not_search_t<T>> auto &
set_responses(const table &value)
```

_const_table&indices

Indices of nearest neighbors. **Default value**: table{}.

Getter & Setter

```
const table & get_indices() constauto & set_indices(const table
&value)
```

_const_result_option_id&result_options

Result options that indicates availability of the properties.

Getter & Setter

```
const result_option_id & get_result_options() constauto &
set_result_options(const result_option_id &value)
```

_const_table&distances

Distances to nearest neighbors. **Default value**: table{}.

Getter & Setter

```
const table & get_distances() constauto & set_distances(const
table &value)
```

_const_table&labels

The predicted labels. **Default value**: table{}.

Getter & Setter

```
const table & get_labels() consttemplate <typename T = Task,
typename None = detail::enable_if_classification_t<T>> auto &
set_labels(const table &value)
```

Operation

***template*<*typename*Descriptor>knn::infer_resultinfer(*const*Descriptor&desc, *const*knn::infer_input&input)**

Parameters

- **desc** – k-NN algorithm descriptor **knn::descriptor**
- **input** – Input data for the inference operation

Preconditions

input.data.has_data==*true*

Postconditions

result.labels.row_count==input.data.row_countresult.labels.column_count==1result.labels[i]>=0result.labels[i]<desc.class_count

Usage example

Training

```
knn::model<> run_training(const table& data,
                          const table& labels) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = train(knn_desc, data, labels);

   return result.get_model();
}
```

Inference

```
table run_inference(const knn::model<>& model,
                    const table& new_data) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = infer(knn_desc, model, new_data);

   print_table("labels", result.get_labels());
}
```

Examples

oneAPI DPC++

Batch Processing:

- dpc_knn_cls_brute_force_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_knn_cls_brute_force_dense_batch.cpp
- cpp_knn_cls_kd_tree_dense_batch.cpp
- cpp_knn_search_brute_force_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- bf_knn_classification_batch.py

Pairwise Distances

This chapter describes programming interfaces of the pairwise distances implemented in oneDAL:

- Minkowski distance
- Chebyshev distance
- Cosine distance

Minkowski distance

The Minkowski distances are the set of distance metrics with different degree $(p > 0)$ and are widely used for distance computation in different algorithms. The most commonly used distance metric, Euclidean distance, is also a Minkowski distance with $p = 2.0$.

Operation	Computational methods
dense	dense

Mathematical formulation

Refer to Developer Guide: Minkowski distance.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::minkowski_distance` namespace.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an the implementation of the algorithm. Can be **method::dense**. - **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

descriptor(doubledegree)

Creates a new instance of the class with the external property values.

Properties

doubledegree

The coefficient p of the Minkowski distance. **Default value**: 2.0.

Getter & Setter	`double get_degree() constauto & set_degree(double value)`

Method tags

***struct*dense**

***using*by_default=dense**

Alias tag-type for the dense method.

Task tags

***struct*compute**

Tag-type that parameterizes entities that are used to compute distances.

***using*by_default=compute**

Alias tag-type for the compute task.

Chebyshev distance

The Chebyshev distance equals the limit of Minkowski distance metric with $p \to \infty$.

Operation	Computational methods
dense	dense

Mathematical formulation

Refer to Developer Guide: Chebyshev distance.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::chebyshev_distance` namespace.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**.
- **Method** – Tag-type that specifies an the implementation of the algorithm. Can be **method::dense**.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Method tags

***struct*dense**

***using*by_default=dense**

Alias tag-type for the dense method.

Task tags

*struct*compute

Tag-type that parameterizes entities that are used to compute distances.

*using*by_default=compute

Alias tag-type for the compute task.

Cosine distance

The Cosine distance is a measure of distance between two non-zero vectors of an inner product space.

Operation	Computational methods
dense	dense

Mathematical formulation

Refer to Developer Guide: Minkowski distance.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::cosine_distance` namespace.

Descriptor

template<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default>*class*descriptor

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**.
- **Method** – Tag-type that specifies the implementation of the algorithm. Can be **method::dense**.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

descriptor()=*default*

Creates a new instance of the class with the default property values.

Method tags

*struct*dense

*using*by_default=dense

Alias tag-type for the dense method.

Task tags

*struct*compute

Tag-type that parameterizes entities that are used to compute distances.

*using*by_default=compute

Alias tag-type for the compute task.

Statistics

This chapter describes programming interfaces of the basic statistics algorithm implemented in oneDAL:

- Basic Statistics

Basic Statistics

Basic statistics algorithm computes the following set of quantitative dataset characteristics:

- minimums/maximums
- sums
- means
- sums of squares
- sums of squared differences from the means
- second order raw moments
- variances
- standard deviations
- variations

Operation	Computational methods	Programming Interface		
dense	dense	compute(...)	compute_input	compute_result

Mathematical formulation

Refer to Developer Guide: Basic statistics.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::basic_statistics` namespace and are available via inclusion of the `oneapi/dal/algo/basic_statistics.hpp` header file.

Descriptor

***template<typename*Float=detail::descriptor_base<>::float_t,*typename*Method=detail::descriptor_base<>::method_t,*typename*Task=detail::descriptor_base<>::task_t>*class*descriptor**

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::dense**.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Properties

result_option_idresult_options

Choose which results should be computed and returned.

Getter & Setter

```
result_option_id get_result_options() constauto &
set_result_options(const result_option_id &value)
```

Method tags

*struct*dense

Tag-type that denotes dense computational method.

*using*by_default=dense

Alias tag-type for dense computational method.

Task tags

*struct*compute

Tag-type that parameterizes entities that are used to compute statistics.

*using*by_default=compute

Alias tag-type for the compute task.

Training compute(...)

Input

template<*typename*Task=task::by_default>*class*compute_input

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_input(*const*table&data)

Creates a new instance of the class with the given `data` property value.

compute_input(*const*table&data, *const*table&weights)

Properties

*const*table&data

An $n \times p$ table with the training data, where each row stores one feature vector. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &data)`

*const*table&weights

Getter & Setter	`const table & get_weights() constauto & set_weights(const table &weights)`

Result

template<*typename*Task=task::by_default>*class*compute_result

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **task::compute**.

Constructors

compute_result()

Creates a new instance of the class with the default property values.

Properties

*const*table&sum

A $1 \times p$ table, where element j is the sum result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_sum() constauto & set_sum(const table &value)`

*const*table&variance

A $1 \times p$ table, where element j is the variance result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_variance() constauto & set_variance(const table &value)`

*const*result_option_id&result_options

Result options that indicates availability of the properties. **Default value**: full set of.

Getter & Setter	`const result_option_id & get_result_options() constauto & set_result_options(const result_option_id &value)`

*const*table&second_order_raw_moment

A $1 \times p$ table, where element j is the second_order_raw_moment result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_second_order_raw_moment() constauto & set_second_order_raw_moment(const table &value)`

*const*table&max

A $1 \times p$ table, where element j is the maximum result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_max() constauto & set_max(const table &value)`

*const*table&standard_deviation

A $1 \times p$ table, where element j is the standard_deviation result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_standard_deviation() constauto & set_standard_deviation(const table &value)`

*const*table&min

A $1 \times p$ table, where element j is the minimum result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_min() constauto & set_min(const table &value)`

_const_table&sum_squares_centered

A $1 \times p$ table, where element j is the sum_squares_centered result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_sum_squares_centered() constauto & set_sum_squares_centered(const table &value)`

_const_table&variation

A $1 \times p$ table, where element j is the variation result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_variation() constauto & set_variation(const table &value)`

_const_table&sum_squares

A $1 \times p$ table, where element j is the sum_squares result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_sum_squares() constauto & set_sum_squares(const table &value)`

_const_table&mean

A $1 \times p$ table, where element j is the mean result for feature j. **Default value**: table{}.

Getter & Setter	`const table & get_mean() constauto & set_mean(const table &value)`

Operation

_template<typename_Descriptor>basic_statistics::compute_resultcompute(_const_Descriptor&desc, _const_basic_statistics::compute_input&input)

Parameters	- **desc** – Basic statistics algorithm descriptor **basic_statistics::descriptor** - **input** – Input data for the computing operation
Preconditions	**input.data.is_empty==_false_**

Support Vector Machines

This chapter describes programming interfaces of the support vector machines implemented in oneDAL:

- Support Vector Machine Classifier (SVM)

Support Vector Machine Classifier (SVM)

Support Vector Machine (SVM) classification and regression are among popular algorithms. It belongs to a family of generalized linear classification problems.

Operation	Computational methods	Programming Interface			
Training	SMO	Thunder	train(...)	train_input	train_result

Inference	SMO	Thunder	infer(...)	infer_input	infer_result

Mathematical formulation

Refer to Developer Guide: Support Vector Machine Classifier.

Programming Interface

All types and functions in this section are declared in the `oneapi::dal::svm` namespace and are available via inclusion of the `oneapi/dal/algo/svm.hpp` header file.

Descriptor

***template*<*typename*Float=float,*typename*Method=method::by_default,*typename*Task=task::by_default,*typename*Kernel=linear_kernel::descriptor<Float>>*class*descriptor**

Template Parameters	- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be **float** or **double**. - **Method** – Tag-type that specifies an implementation of algorithm. Can be **method::thunder** or **method::smo**. - **Task** – Tag-type that specifies the type of the problem to solve. Can be **task::classification**, **task::nu_classification**, **task::regression**, or **task::nu_regression**.

Constructors

descriptor(*const*Kernel&kernel=kernel_t{})

Creates a new instance of the class with the given descriptor of the kernel function.

Properties

std::int64_tmax_iteration_count

The maximum number of iterations T. **Default value**: 100000.

Getter & Setter	`std::int64_t get_max_iteration_count() constauto & set_max_iteration_count(std::int64_t value)`
Invariants	**max_iteration_count>=0**

doubleepsilon

The epsilon. Used with **task::regression** only. **Default value**: 0.1.

Getter & Setter	`template <typename T = Task, typename None = detail::enable_if_epsilon_available_t<T>> double get_epsilon() consttemplate <typename T = Task, typename None = detail::enable_if_epsilon_available_t<T>> auto & set_epsilon(double value)`
Invariants	**epsilon>=0**

doublecache_size

The size of cache (in megabytes) for storing the values of the kernel matrix. **Default value**: 200.0.

Getter & Setter	`double get_cache_size() constauto & set_cache_size(double value)`
Invariants	**cache_size>=0.0**

doublenu

The nu. Used with **task::nu_classification** and **task::nu_regression**. **Default value**: 0.5.

Getter & Setter	`template <typename T = Task, typename None = detail::enable_if_nu_task_t<T>> double get_nu() consttemplate <typename T = Task, typename None = detail::enable_if_nu_task_t<T>> auto & set_nu(double value)`
Invariants	**0<nu<=1**

doublec

The upper bound C in constraints of the quadratic optimization problem. Used with **task::classification**, **task::regression**, and **task::nu_regression**. **Default value**: 1.0.

Getter & Setter	`template <typename T = Task, typename None = detail::enable_if_c_available_t<T>> double get_c() consttemplate <typename T = Task, typename None = detail::enable_if_c_available_t<T>> auto & set_c(double value)`
Invariants	**c>0**

*const*Kernel&kernel

The descriptor of kernel function $K(x, y)$. Can be **linear_kernel::descriptor** or **polynomial_kernel::descriptor** or **rbf_kernel::descriptor** or **sigmoid_kernel::descriptor**.

Getter & Setter	`const Kernel & get_kernel() constauto & set_kernel(const Kernel &kernel)`

std::int64_tclass_count

The number of classes. Used with **task::classification** and **task::nu_classification**. **Default value**: 2.

Getter & Setter	`template <typename T = Task, typename None = detail::enable_if_classification_t<T>> std::int64_t get_class_count() consttemplate <typename T = Task, typename None = detail::enable_if_classification_t<T>> auto & set_class_count(std::int64_t value)`
Invariants	**class_count>=2**

doubleaccuracy_threshold

The threshold ε for the stop condition. **Default value**: 0.0.

Getter & Setter	`double get_accuracy_threshold() constauto & set_accuracy_threshold(double value)`
Invariants	**accuracy_threshold>=0.0**

boolshrinking

A flag that enables the use of a shrinking optimization technique. Used with **method::smo** split-finding method only. **Default value**: true.

Getter & Setter	`bool get_shrinking() constauto & set_shrinking(bool value)`

doubletau

The threshold parameter τ for computing the quadratic coefficient. **Default value**: 1e-6.

Getter & Setter	`double get_tau() constauto & set_tau(double value)`
Invariants	**tau>0.0**

Method tags

*struct*smo

Tag-type that denotes SMO computational method.

*struct*thunder

Tag-type that denotes Thunder computational method.

*using*by_default=thunder

Alias tag-type for Thunder computational method.

Task tags

*struct*classification

Tag-type that parameterizes entities that are used for solving classification problem.

*struct*nu_classification

Tag-type that parameterizes entities that are used for solving nu-classification problem.

*struct*nu_regression

Tag-type that parameterizes entities used for solving nu-regression problem.

*struct*regression

Tag-type that parameterizes entities used for solving regression problem.

*using*by_default=classification

Alias tag-type for classification task.

Model

template<*typename*Task=task::by_default>*class*model

Template Parameters

Task – Tag-type that specifies the type of the problem to solve. Can be **task::classification**, **task::nu_classification**, **task::regression**, or **task::nu_regression**.

Constructors

model()

Creates a new instance of the class with the default property values.

Public Methods

std::int64_tget_support_vector_count()*const*

The number of support vectors.

Properties

*const*table&biases

A $class_count * (class_count - 1)/2 \times 1$ table for **task::classification** and **task::nu_classification** and a 1×1 table for **task::regression** and **task::nu_regression** containing constants in decision function.

Getter & Setter

```
const table & get_biases() constauto & set_biases(const table
&value)
```

std::int64_tfirst_class_response

The first unique value in class responses. Used with **task::classification** and **task::nu_classification**.

Getter & Setter

```
std::int64_t get_first_class_response() consttemplate <typename T
= Task, typename None = detail::enable_if_classification_t<T>>
auto & set_first_class_response(std::int64_t value)
```

std::int64_tfirst_class_label

The first unique value in class labels. Used with **task::classification** and **task::nu_classification**.

Getter & Setter

```
std::int64_t get_first_class_label() consttemplate <typename T =
Task, typename None = detail::enable_if_classification_t<T>> auto
& set_first_class_label(std::int64_t value)
```

*const*table&support_vectors

A $nsv \times p$ table containing support vectors. Where nsv - number of support vectors. **Default value**: table{}.

Getter & Setter

```
const table & get_support_vectors() constauto &
set_support_vectors(const table &value)
```

*const*table&coeffs

A $nsv \times class_count - 1$ table for **task::classification** and **task::nu_classification** and a $nsv \times 1$ table for **task::regression** and **task::nu_regression** containing coefficients of Lagrange multiplier. **Default value**: table{}.

Getter & Setter	`const table & get_coeffs() constauto & set_coeffs(const table &value)`

std::int64_tsecond_class_response

The second unique value in class responses. Used with **task::classification** and **task::nu_classification**.

Getter & Setter	`std::int64_t get_second_class_response() consttemplate <typename T = Task, typename None = detail::enable_if_classification_t<T>> auto & set_second_class_response(std::int64_t value)`

doublebias

The bias. **Default value**: 0.0.

Getter & Setter	`double get_bias() constauto & set_bias(double value)`

std::int64_tsecond_class_label

The second unique value in class labels. Used with **task::classification** and **task::nu_classification**.

Getter & Setter	`std::int64_t get_second_class_label() consttemplate <typename T = Task, typename None = detail::enable_if_classification_t<T>> auto & set_second_class_label(std::int64_t value)`

Training train(...)

Input

template<*typename*Task=task::by_default>*class*train_input

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **oneapi::dal::svm::task::classification**, **oneapi::dal::svm::task::nu_classification**, **oneapi::dal::svm::task::regression**, or **oneapi::dal::svm::task::nu_regression**.

Constructors

train_input(*const*table&data, *const*table&responses, *const*table&weights=table{})

Creates a new instance of the class with the given `data`, `responses` and `weights`.

Properties

*const*table&data

The training set X. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &value)`

*const*table&weights

The vector of weights w for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_weights() constauto & set_weights(const table &value)`

*const*table&responses

The vector of responses y for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

*const*table&labels

The vector of labels y for the training set X. **Default value**: table{}.

Getter & Setter	`const table & get_labels() constauto & set_labels(const table &value)`

Result

*template<typename*Task=task::by_default>*class*train_result

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **oneapi::dal::svm::task::classification**, **oneapi::dal::svm::task::nu_classification**, **oneapi::dal::svm::task::regression**, or **oneapi::dal::svm::task::nu_regression**.

Constructors

train_result()

Creates a new instance of the class with the default property values.

Public Methods

std::int64_tget_support_vector_count()*const*

The number of support vectors.

Properties

*const*table&biases

A $class_{c}ount * (class_{c}ount - 1)/2 \times 1$ table for **task::classification** and **task::classification** and 1×1 table for **task::regression** and **task::nu_regression** containing constants in decision function.

Getter & Setter	`const table & get_biases() constauto & set_biases(const table &value)`

*const*table&support_vectors

A $nsv \times p$ table containing support vectors, where nsv is the number of support vectors. **Default value**: table{}.

Getter & Setter	`const table & get_support_vectors() constauto & set_support_vectors(const table &value)`

*const*table&coeffs

A $nsv \times class_count - 1$ table for **task::classification** and **task::classification** and $nsv \times 1$ table for **task::regression** and **task::nu_regression** containing coefficients of Lagrange multiplier. **Default value**: table{}.

Getter & Setter	`const table & get_coeffs() constauto & set_coeffs(const table &value)`

*const*table&support_indices

A $nsv \times 1$ table containing support indices. **Default value**: table{}.

Getter & Setter	`const table & get_support_indices() constauto & set_support_indices(const table &value)`

doublebias

The bias. **Default value**: 0.0.

Getter & Setter	`double get_bias() constauto & set_bias(double value)`

*const*model<Task>&model

The trained SVM model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Operation

*template<typename*Descriptor>svm::train_resulttrain(*const*Descriptor&desc, *const*svm::train_input&input)

Parameters	• **desc** – SVM algorithm descriptor **svm::descriptor**. • **input** – Input data for the training operation
Preconditions	**input.data.is_empty==*false*input.labels.is_empty==*false*input.labels.column_count==1input.data.row_count==input.labels.row_count**

Inference infer(...)

Input

*template<typename*Task=task::by_default>*class*infer_input

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **oneapi::dal::svm::task::classification**, **oneapi::dal::svm::task::nu_classification**, **oneapi::dal::svm::task::regression**, or **oneapi::dal::svm::task::nu_regression**.

Constructors

infer_input(*const*model<Task>&trained_model, *const*table&data)

Creates a new instance of the class with the given `model` and `data` property values.

Properties

***const*table&data**

The dataset for inference X'. **Default value**: table{}.

Getter & Setter	`const table & get_data() constauto & set_data(const table &value)`

***const*model<Task>&model**

The trained SVM model. **Default value**: model<Task>{}.

Getter & Setter	`const model< Task > & get_model() constauto & set_model(const model< Task > &value)`

Result

***template*<*typename*Task=task::by_default>*class*infer_result**

Template Parameters	**Task** – Tag-type that specifies the type of the problem to solve. Can be **oneapi::dal::svm::task::classification**, **oneapi::dal::svm::task::nu_classification**, **oneapi::dal::svm::task::regression**, or **oneapi::dal::svm::task::nu_regression**.

Constructors

infer_result()

Creates a new instance of the class with the default property values.

Properties

***const*table&labels**

The $n \times 1$ table with the predicted labels. **Default value**: table{}.

Getter & Setter	`const table & get_labels() constauto & set_labels(const table &value)`

***const*table&decision_function**

The $n \times 1$ table with the predicted class. Used with **oneapi::dal::svm::task::classification** and **oneapi::dal::svm::task::nu_classification**. decision function for each observation. **Default value**: table{}.

Getter & Setter	`const table & get_decision_function() consttemplate <typename T = Task, typename None = detail::enable_if_classification_t<T>> auto & set_decision_function(const table &value)`

***const*table&responses**

The $n \times 1$ table with the predicted responses. **Default value**: table{}.

Getter & Setter	`const table & get_responses() constauto & set_responses(const table &value)`

Operation

***template<typename*Descriptor>svm::infer_resultinfer(*const*Descriptor&desc, *const*svm::infer_input&input)**

Parameters	• **desc** – SVM algorithm descriptor **svm::descriptor**. • **input** – Input data for the inference operation
Preconditions	**input.data.is_empty==*false***

Examples

oneAPI DPC++

Batch Processing:

- dpc_svm_two_class_thunder_dense_batch.cpp

oneAPI C++

Batch Processing:

- cpp_svm_two_class_smo_dense_batch.cpp
- cpp_svm_two_class_thunder_dense_batch.cpp
- cpp_svm_reg_thunder_dense_batch.cpp
- cpp_svm_multi_class_thunder_dense_batch.cpp
- cpp_svm_nu_cls_thunder_dense_batch.cpp
- cpp_svm_nu_reg_thunder_dense_batch.cpp

Python* with DPC++ support

Batch Processing:

- svm_batch.py

Distributed Model: Single Process Multiple Data

Refer to Developer Guide: SPMD distributed model.

- Distributed SPMD model
  - Programming interface
  - Usage example
- Communicators
  - Programming interface
    - Communicator

- USM and non-USM memory usage
- Request
- Reducion operations

Distributed SPMD model

Refer to Developer Guide: SPMD.

Programming interface

All types and functions in this section are declared in the `oneapi::dal::spmd::preview` namespace and are available via inclusion of the header file from specified backend.

SPMD distributed model consists of the following components:

1. Additional `train`, `infer`, and `compute` methods that accept `communicator` object as the first parameter. Those methods are expected to be called on all ranks to start distributed simulations.
2. The communicator class that contains methods to perform collective operations among all ranks.
3. Free functions to create a communicator using a specified communicator backend. Available backends are `ccl` and `mpi`.

Usage example

The following listings provide a brief introduction on how to create a particular communicator.

MPI backend

```
#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/kmeans.hpp"
#include "oneapi/dal/spmd/mpi/communicator.hpp"

kmeans::model<> run_training(const table& data,
                           const table& initial_centroids) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(10)
      .set_max_iteration_count(50)
      .set_accuracy_threshold(1e-4);

   auto comm = dal::preview::spmd::make_communicator<dal::preview::spmd::backend::mpi>(queue);
   auto rank_id = comm.get_rank();

   const auto result_train = dal::preview::train(comm, kmeans_desc, local_input);

   if(rank_id == 0) {
      print_table("centroids", result.get_model().get_centroids());
      print_value("objective", result.get_objective_function_value());
   }
   return result.get_model();
}
```

CCL backend

```
#ifndef ONEDAL_DATA_PARALLEL
#define ONEDAL_DATA_PARALLEL
#endif

#include "oneapi/dal/algo/kmeans.hpp"
```

```
#include "oneapi/dal/spmd/ccl/communicator.hpp"

kmeans::model<> run_training(const table& data,
                             const table& initial_centroids) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(10)
      .set_max_iteration_count(50)
      .set_accuracy_threshold(1e-4);

   auto comm = dal::preview::spmd::make_communicator<dal::preview::spmd::backend::ccl>(queue);
   auto rank_id = comm.get_rank();

   const auto result_train = dal::preview::train(comm, kmeans_desc, local_input);

   if(rank_id == 0) {
      print_table("centroids", result.get_model().get_centroids());
      print_value("objective", result.get_objective_function_value());
   }
   return result.get_model();
}
```

Communicators

Programming interface

All types and functions in this section are declared in the `oneapi::dal::spmd::preview` namespace and are available via inclusion of the header file from specified backend.

Communicator

A base implementation of the communicator concept. The communicator type and all of its subtypes are reference-counted:

1. The instance stores a pointer to the communicator implementation that holds all property values and data.
2. The reference count indicates how many communicator objects refer to the same implementation.
3. The communicator increments the reference count for it to be equal to the number of communicator objects sharing the same implementation.
4. The communicator decrements the reference count when the communicator goes out of the scope. If the reference count is zero, the communicator frees its implementation.

USM and non-USM memory usage

There are two types of memory access:

- USM memory access (both USM and non-USM pointers can be used)
- Host, or non-USM, memory access (only non-USM pointers can be used)

Use one of the following tags to select a memory access type:

device_memory_access::none Assumes only non-USM pointers are used for a collective operation.

device_memory_access::usm Both USM and non-USM can be used. Pointer type is controlled by the use of `sycl::queue` object as a first parameter for collective operations. The use of `sycl::queue` object is obligatory for USM pointers.

Request

Request is an object to control asynchronous communication.

Reducion operations

The following reduction operations are supported:

- Max
- Min
- Sum

Notices and Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.