Package ‘cuda.ml’

July 22, 2025

Type Package

Title R Interface for the RAPIDS cuML Suite of Libraries

Version 0.3.2

Maintainer Daniel Falbel <daniel@rstudio.com>

Description R interface for RAPIDS cuML (<https://github.com/rapidsai/cuml>), a suite of GPU-accelerated machine learning libraries powered by CUDA (<https://en.wikipedia.org/wiki/CUDA>).

License MIT + file LICENSE

URL https://mlverse.github.io/cuda.ml/

BugReports https://github.com/mlverse/cuda.ml/issues

Depends R (>= 3.2)

Imports ellipsis, hardhat, parsnip, Rcpp (>= 1.0.6), rlang (>= 0.1.4)

Suggests callr, glmnet, MASS, magrittr, mlbench, purrr, reticulate, testthat, xgboost

LinkingTo Rcpp

Encoding UTF-8

RoxygenNote 7.1.2

OS_type unix

SystemRequirements RAPIDS cuML (see https://rapids.ai/start.html)

NeedsCompilation yes

Author Yitao Li [aut, cph] (ORCID: <https://orcid.org/0000-0002-1261-905X>),
Tomasz Kalinowski [cph, ctb],
Daniel Falbel [aut, cre, cph],
RStudio [cph, fnd]

Repository CRAN

Date/Publication 2022-01-08 01:42:47 UTC

Contents

`cuda.ml` *cuda.ml*

Description

This package provides a R interface for the RAPIDS cuML library.

Author(s)

Yitao Li <yitao@rstudio.com>

`cuda_ml_agglomerative_clustering` *Perform Single-Linkage Agglomerative Clustering.*

Description

Recursively merge the pair of clusters that minimally increases a given linkage distance.

Usage

```
cuda_ml_agglomerative_clustering(
  x,
  n_clusters = 2L,
  metric = c("euclidean", "l1", "l2", "manhattan", "cosine"),
  connectivity = c("pairwise", "knn"),
  n_neighbors = 15L
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`n_clusters`	The number of clusters to find. Default: 2L.
`metric`	Metric used for linkage computation. Must be one of "euclidean", "l1", "l2", "manhattan", "cosine". If connectivity is "knn" then only "euclidean" is accepted. Default: "euclidean".
`connectivity`	The type of connectivity matrix to compute. Must be one of "pairwise", "knn". Default: "pairwise". - 'pairwise' will compute the entire fully-connected graph of pairwise distances between each set of points. This is the fastest to compute and can be very fast for smaller datasets but requires O(n^2) space. - 'knn' will sparsify the fully-connected connectivity matrix to save memory and enable much larger inputs. "n_neighbors" will control the amount of memory used and the graph will be connected automatically in the event "n_neighbors" was not large enough to connect it.
`n_neighbors`	The number of neighbors to compute when `connectivity` is "knn". Default: 15L.

Value

A clustering object with the following attributes: "n_clusters": The number of clusters found by the algorithm. "children": The children of each non-leaf node. Values less than `nrow(x)` correspond to leaves of the tree which are the original samples. `children[i + 1][1]` and `children[i + 1][2]` were merged to form node `(nrow(x) + i)` in the `i`-th iteration. "labels": cluster label of each data point.

Examples

```
library(cuda.ml)
library(MASS)
library(magrittr)
library(purrr)

set.seed(0L)

gen_pts <- function() {
  centers <- list(c(1000, 1000), c(-1000, -1000), c(-1000, 1000))
  pts <- centers %>%
    map(~ mvrnorm(50, mu = .x, Sigma = diag(2)))

  rlang::exec(rbind, !!!pts) %>% as.matrix()
}

clust <- cuda_ml_agglomerative_clustering(
  x = gen_pts(),
  metric = "euclidean",
  n_clusters = 3L
)

print(clust$labels)
```

`cuda_ml_can_predict_class_probabilities` *Determine whether a CuML model can predict class probabilities.*

Description

Given a trained CuML model, return `TRUE` if the model is a classifier and is capable of outputting class probabilities as prediction results (e.g., if the model is a KNN or an ensemble classifier), otherwise return `FALSE`.

Usage

```
cuda_ml_can_predict_class_probabilities(model)
```

Arguments

`model`	A trained CuML model.

Value

A logical value indicating whether the model supports outputting class probabilities.

`cuda_ml_dbscan` *Run the DBSCAN clustering algorithm.*

Description

Run the DBSCAN (Density-based spatial clustering of applications with noise) clustering algorithm.

Usage

```
cuda_ml_dbscan(
  x,
  min_pts,
  eps,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace")
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`min_pts`, `eps`	A point ‘p‘ is a core point if at least ‘min_pts‘ are within distance ‘eps‘ from it.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.

Value

A list containing the cluster assignments of all data points. A data point not belonging to any cluster (i.e., "noise") will have NA its cluster assignment.

Examples

```
library(cuda.ml)
library(magrittr)

gen_pts <- function() {
  centroids <- list(c(1000, 1000), c(-1000, -1000), c(-1000, 1000))

  pts <- centroids %>%
    purrr::map(~ MASS::mvrnorm(10, mu = .x, Sigma = diag(2)))

  rlang::exec(rbind, !!!pts)
}
```

```
m <- gen_pts()
clusters <- cuda_ml_dbscan(m, min_pts = 5, eps = 3)

print(clusters)
```

`cuda_ml_elastic_net` *Train a linear model using elastic regression.*

Description

Train a linear model with combined L1 and L2 priors as the regularizer.

Usage

```
cuda_ml_elastic_net(x, ...)

## Default S3 method:
cuda_ml_elastic_net(x, ...)

## S3 method for class 'data.frame'
cuda_ml_elastic_net(
  x,
  y,
  alpha = 1,
  l1_ratio = 0.5,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)

## S3 method for class 'matrix'
cuda_ml_elastic_net(
  x,
  y,
  alpha = 1,
  l1_ratio = 0.5,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)
```

```
## S3 method for class 'formula'
cuda_ml_elastic_net(
  formula,
  data,
  alpha = 1,
  l1_ratio = 0.5,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)

## S3 method for class 'recipe'
cuda_ml_elastic_net(
  x,
  data,
  alpha = 1,
  l1_ratio = 0.5,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`alpha`	Multiplier of the penalty term (i.e., the result would become and Ordinary Least Square model if `alpha` were set to 0). Default: 1. For numerical reasons, running elastic regression with `alpha` set to 0 is not advised. For the `alpha`-equals-to-0 scenario, one should use `cuda_ml_ols` to train an OLS model instead. Default: 1.
`l1_ratio`	The ElasticNet mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an L2 penalty. For l1_ratio = 1 it is an L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2. The penalty term is computed using the following formula: penalty = `alpha` * `l1_ratio` * ||w||_1 + 0.5 * `alpha` * (1 - `l1_ratio`) * ||w||^2_2 where ||w||_1 is the L1 norm of the coefficients, and ||w||_2 is the L2 norm of the coefficients.

`max_iter`	The maximum number of coordinate descent iterations. Default: 1000L.
`tol`	Stop the coordinate descent when the duality gap is below this threshold. Default: 1e-3.
`fit_intercept`	If TRUE, then the model tries to correct for the global mean of the response variable. If FALSE, then the model expects data to be centered. Default: TRUE.
`normalize_input`	Ignored when `fit_intercept` is FALSE. If TRUE, then the predictors will be normalized to have a L2 norm of 1. Default: FALSE.
`selection`	If "random", then instead of updating coefficients in cyclic order, a random coefficient is updated in each iteration. Default: "cyclic".
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

An elastic net regressor that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

model <- cuda_ml_elastic_net(
  formula = mpg ~ ., data = mtcars, alpha = 1e-3, l1_ratio = 0.6
)
cuda_ml_predictions <- predict(model, mtcars)

# predictions will be comparable to those from a `glmnet` model with `lambda`
# set to 1e-3 and `alpha` set to 0.6
# (in `glmnet`, `lambda` is the weight of the penalty term, and `alpha` is
#  the elastic mixing parameter between L1 and L2 penalties.

library(glmnet)

glmnet_model <- glmnet(
  x = as.matrix(mtcars[names(mtcars) != "mpg"]), y = mtcars$mpg,
  alpha = 0.6, lambda = 1e-3, nlambda = 1, standardize = FALSE
)

glm_predictions <- predict(
  glmnet_model, as.matrix(mtcars[names(mtcars) != "mpg"]),
  s = 0
)

print(
  all.equal(
    as.numeric(glm_predictions),
    cuda_ml_predictions$.pred,
```

```
    tolerance = 1e-2
  )
)
```

`cuda_ml_fil_enabled` *Determine whether Forest Inference Library (FIL) functionalities are enabled in the current installation of cuda.ml.*

Description

CuML Forest Inference Library (FIL) functionalities (see https://github.com/rapidsai/cuml/tree/main/python/cuml/fil#readme will require Treelite C API. If you need FIL to run tree-based model ensemble on GPU, and `fil_enabled()` returns FALSE, then please consider installing Treelite and then re-installing cuda.ml.

Usage

```
cuda_ml_fil_enabled()
```

Value

A logical value indicating whether the Forest Inference Library (FIL) functionalities are enabled.

Examples

```
if (cuda_ml_fil_enabled()) {
  # run GPU-accelerated Forest Inference Library (FIL) functionalities
} else {
  message(
    "FIL functionalities are disabled in the current installation of ",
    "{cuda.ml}. Please reinstall Treelite C library first, and then re-install",
    " {cuda.ml} to enable FIL."
  )
}
```

`cuda_ml_fil_load_model` *Load a XGBoost or LightGBM model file.*

Description

Load a XGBoost or LightGBM model file using Treelite. The resulting model object can be used to perform high-throughput batch inference on new data points using the GPU acceleration functionality from the CuML Forest Inference Library (FIL).

Usage

```
cuda_ml_fil_load_model(
  filename,
  mode = c("classification", "regression"),
  model_type = c("xgboost", "lightgbm"),
  algo = c("auto", "naive", "tree_reorg", "batch_tree_reorg"),
  threshold = 0.5,
  storage_type = c("auto", "dense", "sparse"),
  threads_per_tree = 1L,
  n_items = 0L,
  blocks_per_sm = 0L
)
```

Arguments

`filename`	Path to the saved model file.
`mode`	Type of task to be performed by the model. Must be one of "classification", "regression".
`model_type`	Format of the saved model file. Notice if `filename` ends with ".json" and `model_type` is "xgboost", then cuda.ml will assume the model file is in XGBoost JSON (instead of binary) format. Default: "xgboost".
`algo`	Type of the algorithm for inference, must be one of the following. - "auto": Choose the algorithm automatically. Currently 'batch_tree_reorg' is used for dense storage, and 'naive' for sparse storage. - "naive": Simple inference using shared memory. - "tree_reorg": Similar to naive but with trees rearranged to be more coalescing- friendly. - "batch_tree_reorg": Similar to 'tree_reorg' but predicting multiple rows per thread block. Default: "auto".
`threshold`	Class probability threshold for classification. Ignored for regression tasks. Default: 0.5.
`storage_type`	In-memory storage format of the FIL model. Must be one of the following. - "auto": Choose the storage type automatically, - "dense": Create a dense forest, - "sparse": Create a sparse forest. Requires `algo` to be 'naive' or 'auto'.
`threads_per_tree`	If >1, then have multiple (neighboring) threads infer on the same tree within a block, which will improve memory bandwith near tree root (but consuming more shared memory). Default: 1L.
`n_items`	Number of input samples each thread processes. If 0, then choose (up to 4) that fit into shared memory. Default: 0L.
`blocks_per_sm`	Indicates how CuML should determine the number of thread blocks to lauch for the inference kernel. - 0: Launches the number of blocks proportional to the number of data points. - >= 1: Attempts to lauch `blocks_per_sm` blocks for each streaming multiprocessor. This will fail if `blocks_per_sm` blocks result in more threads than the maximum supported number of threads per GPU. Even if successful, it is not guaranteed that `blocks_per_sm` blocks will run on an SM concurrently.

Value

A GPU-accelerated FIL model that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)
library(xgboost)

model_path <- file.path(tempdir(), "xgboost.model")

model <- xgboost(
  data = as.matrix(mtcars[names(mtcars) != "mpg"]),
  label = as.matrix(mtcars["mpg"]),
  max.depth = 6,
  eta = 1,
  nthread = 2,
  nrounds = 20,
  objective = "reg:squarederror"
)

xgb.save(model, model_path)

model <- cuda_ml_fil_load_model(
  model_path,
  mode = "regression",
  model_type = "xgboost"
)

preds <- predict(model, mtcars[names(mtcars) != "mpg"])

print(preds)
```

`cuda_ml_inverse_transform` *Apply the inverse transformation defined by a trained cuML model.*

Description

Given a trained cuML model, apply the inverse transformation defined by that model to an input dataset.

Usage

```
cuda_ml_inverse_transform(model, x, ...)
```

Arguments

`model`	A model object.
`x`	The dataset to be transformed.
`...`	Additional model-specific parameters (if any).

Value

The transformed data points.

`cuda_ml_is_classifier` *Determine whether a CuML model is a classifier.*

Description

Given a trained CuML model, return `TRUE` if the model is a classifier, otherwise `FALSE` (e.g., if the model is a regressor).

Usage

```
cuda_ml_is_classifier(model)
```

Arguments

`model`	A trained CuML model.

Value

A logical value indicating whether the model is a classifier.

`cuda_ml_kmeans` *Run the K means clustering algorithm.*

Description

Run the K means clustering algorithm.

Usage

```
cuda_ml_kmeans(
  x,
  k,
  max_iters = 300,
  tol = 0,
  init_method = c("kmeans++", "random"),
  seed = 0L,
 cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace")
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`k`	The number of clusters.
`max_iters`	Maximum number of iterations. Default: 300.
`tol`	Relative tolerance with regards to inertia to declare convergence. Default: 0 (i.e., do not use inertia-based stopping criterion).
`init_method`	Method for initializing the centroids. Valid methods include "kmeans++", "random", or a matrix of k rows, each row specifying the initial value of a centroid. Default: "kmeans++".
`seed`	Seed to the random number generator. Default: 0.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.

Value

A list containing the cluster assignments and the centroid of each cluster. Each centroid will be a column within the ‘centroids‘ matrix.

Examples

```
library(cuda.ml)

kclust <- cuda_ml_kmeans(
  iris[names(iris) != "Species"],
  k = 3, max_iters = 100
)

print(kclust)
```

`cuda_ml_knn` *Build a KNN model.*

Description

Build a k-nearest-model for classification or regression tasks.

Usage

```
cuda_ml_knn(x, ...)

## Default S3 method:
cuda_ml_knn(x, ...)

## S3 method for class 'data.frame'
cuda_ml_knn(
```

```
  x,
  y,
  algo = c("brute", "ivfflat", "ivfpq", "ivfsq"),
  metric = c("euclidean", "l2", "l1", "cityblock", "taxicab", "manhattan",
   "braycurtis", "canberra", "minkowski", "chebyshev", "jensenshannon", "cosine",
    "correlation"),
  p = 2,
  neighbors = 5L,
  ...
)

## S3 method for class 'matrix'
cuda_ml_knn(
  x,
  y,
  algo = c("brute", "ivfflat", "ivfpq", "ivfsq"),
  metric = c("euclidean", "l2", "l1", "cityblock", "taxicab", "manhattan",
   "braycurtis", "canberra", "minkowski", "chebyshev", "jensenshannon", "cosine",
    "correlation"),
  p = 2,
  neighbors = 5L,
  ...
)

## S3 method for class 'formula'
cuda_ml_knn(
  formula,
  data,
  algo = c("brute", "ivfflat", "ivfpq", "ivfsq"),
  metric = c("euclidean", "l2", "l1", "cityblock", "taxicab", "manhattan",
   "braycurtis", "canberra", "minkowski", "chebyshev", "jensenshannon", "cosine",
    "correlation"),
  p = 2,
  neighbors = 5L,
  ...
)

## S3 method for class 'recipe'
cuda_ml_knn(
  x,
  data,
  algo = c("brute", "ivfflat", "ivfpq", "ivfsq"),
  metric = c("euclidean", "l2", "l1", "cityblock", "taxicab", "manhattan",
   "braycurtis", "canberra", "minkowski", "chebyshev", "jensenshannon", "cosine",
    "correlation"),
  p = 2,
  neighbors = 5L,
  ...
```

```
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`algo`	The query algorithm to use. Must be one of "brute", "ivfflat", "ivfpq", "ivfsq" or a KNN algorithm specification constructed using the `cuda_ml_knn_algo_*` family of functions. If the algorithm is specified by one of the `cuda_ml_knn_algo_*` functions, then values of all required parameters of the algorithm will need to be specified explicitly. If the algorithm is specified by a character vector, then parameters for the algorithm are generated automatically. Descriptions of supported algorithms: - "brute": for brute-force, slow but produces exact results. - "ivfflat": for inverted file, divide the dataset in partitions and perform search on relevant partitions only. - "ivfpq": for inverted file and product quantization (vectors are divided into sub-vectors, and each sub-vector is encoded using intermediary k-means clusterings to provide partial information). - "ivfsq": for inverted file and scalar quantization (vectors components are quantized into reduced binary representation allowing faster distances calculations). Default: "brute".
`metric`	Distance metric to use. Must be one of "euclidean", "l2", "l1", "cityblock", "taxicab", "manhattan", "braycurtis", "canberra", "minkowski", "lp", "chebyshev", "linf", "jensenshannon", "cosine", "correlation". Default: "euclidean".
`p`	Parameter for the Minkowski metric. If p = 1, then the metric is equivalent to manhattan distance (l1). If p = 2, the metric is equivalent to euclidean distance (l2).
`neighbors`	Number of nearest neighbors to query. Default: 5L.
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A KNN model that can be used with the 'predict' S3 generic to make predictions on new data points. The model object contains the following: - "knn_index": a GPU pointer to the KNN index. - "algo": enum value of the algorithm being used for the KNN query. - "metric": enum value of the distance metric used in KNN computations. - "p": parameter for the Minkowski metric. - "n_samples": number of input data points. - "n_dims": dimension of each input data point.

Examples

```
library(cuda.ml)
library(MASS)
library(magrittr)
library(purrr)

set.seed(0L)

centers <- list(c(3, 3), c(-3, -3), c(-3, 3))

gen_pts <- function(cluster_sz) {
  pts <- centers %>%
    map(~ mvrnorm(cluster_sz, mu = .x, Sigma = diag(2)))

  rlang::exec(rbind, !!!pts) %>% as.matrix()
}

gen_labels <- function(cluster_sz) {
  seq_along(centers) %>%
    sapply(function(x) rep(x, cluster_sz)) %>%
    factor()
}

sample_cluster_sz <- 1000
sample_pts <- cbind(
  gen_pts(sample_cluster_sz) %>% as.data.frame(),
  label = gen_labels(sample_cluster_sz)
)

model <- cuda_ml_knn(label ~ ., sample_pts, algo = "ivfflat", metric = "euclidean")

test_cluster_sz <- 10
test_pts <- gen_pts(test_cluster_sz) %>% as.data.frame()

predictions <- predict(model, test_pts)
print(predictions, n = 30)
```

`cuda_ml_knn_algo_ivfflat`

Build a specification for the "ivfflat" KNN query algorithm.

Description

Build a specification of the flat-inverted-file KNN query algorithm, with all required parameters specified explicitly.

Usage

```
cuda_ml_knn_algo_ivfflat(nlist, nprobe)
```

Arguments

`nlist`	Number of cells to partition dataset into.
`nprobe`	At query time, the number of cells used for approximate nearest neighbor search.

Value

An object encapsulating all required parameters of the "ivfflat" KNN query algorithm.

`cuda_ml_knn_algo_ivfpq`

Build a specification for the "ivfpq" KNN query algorithm.

Description

Build a specification of the inverted-file-product-quantization KNN query algorithm, with all required parameters specified explicitly.

Usage

```
cuda_ml_knn_algo_ivfpq(
  nlist,
  nprobe,
  m,
  n_bits,
  use_precomputed_tables = FALSE
)
```

Arguments

`nlist`	Number of cells to partition dataset into.
`nprobe`	At query time, the number of cells used for approximate nearest neighbor search.
`m`	Number of subquantizers.
`n_bits`	Bits allocated per subquantizer.
`use_precomputed_tables`	Whether to use precomputed tables.

Value

An object encapsulating all required parameters of the "ivfpq" KNN query algorithm.

`cuda_ml_knn_algo_ivfsq` *Build a specification for the "ivfsq" KNN query algorithm.*

Description

Build a specification of the inverted-file-scalar-quantization KNN query algorithm, with all required parameters specified explicitly.

Usage

```
cuda_ml_knn_algo_ivfsq(
  nlist,
  nprobe,
 qtype = c("QT_8bit", "QT_4bit", "QT_8bit_uniform", "QT_4bit_uniform", "QT_fp16",
    "QT_8bit_direct", "QT_6bit"),
  encode_residual = FALSE
)
```

Arguments

`nlist`	Number of cells to partition dataset into.
`nprobe`	At query time, the number of cells used for approximate nearest neighbor search.
`qtype`	Quantizer type. Must be one of "QT_8bit", "QT_4bit", "QT_8bit_uniform", "QT_4bit_uniform", "QT_fp16", "QT_8bit_direct", "QT_6bit".
`encode_residual`	Whether to encode residuals.

Value

An object encapsulating all required parameters of the "ivfsq" KNN query algorithm.

`cuda_ml_lasso` *Train a linear model using LASSO regression.*

Description

Train a linear model using LASSO (Least Absolute Shrinkage and Selection Operator) regression.

Usage

```
cuda_ml_lasso(x, ...)

## Default S3 method:
cuda_ml_lasso(x, ...)

## S3 method for class 'data.frame'
cuda_ml_lasso(
  x,
  y,
  alpha = 1,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)

## S3 method for class 'matrix'
cuda_ml_lasso(
  x,
  y,
  alpha = 1,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)

## S3 method for class 'formula'
cuda_ml_lasso(
  formula,
  data,
  alpha = 1,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)

## S3 method for class 'recipe'
cuda_ml_lasso(
  x,
```

```
  data,
  alpha = 1,
  max_iter = 1000L,
  tol = 0.001,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  selection = c("cyclic", "random"),
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`alpha`	Multiplier of the L1 penalty term (i.e., the result would become and Ordinary Least Square model if `alpha` were set to 0). Default: 1.
`max_iter`	The maximum number of coordinate descent iterations. Default: 1000L.
`tol`	Stop the coordinate descent when the duality gap is below this threshold. Default: 1e-3.
`fit_intercept`	If TRUE, then the model tries to correct for the global mean of the response variable. If FALSE, then the model expects data to be centered. Default: TRUE.
`normalize_input`	Ignored when `fit_intercept` is FALSE. If TRUE, then the predictors will be normalized to have a L2 norm of 1. Default: FALSE.
`selection`	If "random", then instead of updating coefficients in cyclic order, a random coefficient is updated in each iteration. Default: "cyclic".
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A LASSO regressor that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

model <- cuda_ml_lasso(formula = mpg ~ ., data = mtcars, alpha = 1e-3)
```

```
cuda_ml_predictions <- predict(model, mtcars)

# predictions will be comparable to those from a `glmnet` model with `lambda`
# set to 1e-3 and `alpha` set to 1
# (in `glmnet`, `lambda` is the weight of the penalty term, and `alpha` is
#  the elastic mixing parameter between L1 and L2 penalties.

library(glmnet)

glmnet_model <- glmnet(
  x = as.matrix(mtcars[names(mtcars) != "mpg"]), y = mtcars$mpg,
  alpha = 1, lambda = 1e-3, nlambda = 1, standardize = FALSE
)

glm_predictions <- predict(
  glmnet_model, as.matrix(mtcars[names(mtcars) != "mpg"]),
  s = 0
)

print(
  all.equal(
    as.numeric(glm_predictions),
    cuda_ml_predictions$.pred,
    tolerance = 1e-2
  )
)
```

`cuda_ml_logistic_reg` *Train a logistic regression model.*

Description

Train a logistic regression model using Quasi-Newton (QN) algorithms (i.e., Orthant-Wise Limited Memory Quasi-Newton (OWL-QN) if there is L1 regularization, Limited Memory BFGS (L-BFGS) otherwise).

Usage

```
cuda_ml_logistic_reg(x, ...)

## Default S3 method:
cuda_ml_logistic_reg(x, ...)

## S3 method for class 'data.frame'
cuda_ml_logistic_reg(
  x,
  y,
  fit_intercept = TRUE,
  penalty = c("l2", "l1", "elasticnet", "none"),
```

```
  tol = 1e-04,
  C = 1,
  class_weight = NULL,
  sample_weight = NULL,
  max_iters = 1000L,
  linesearch_max_iters = 50L,
  l1_ratio = NULL,
  ...
)

## S3 method for class 'matrix'
cuda_ml_logistic_reg(
  x,
  y,
  fit_intercept = TRUE,
  penalty = c("l2", "l1", "elasticnet", "none"),
  tol = 1e-04,
  C = 1,
  class_weight = NULL,
  sample_weight = NULL,
  max_iters = 1000L,
  linesearch_max_iters = 50L,
  l1_ratio = NULL,
  ...
)

## S3 method for class 'formula'
cuda_ml_logistic_reg(
  formula,
  data,
  fit_intercept = TRUE,
  penalty = c("l2", "l1", "elasticnet", "none"),
  tol = 1e-04,
  C = 1,
  class_weight = NULL,
  sample_weight = NULL,
  max_iters = 1000L,
  linesearch_max_iters = 50L,
  l1_ratio = NULL,
  ...
)

## S3 method for class 'recipe'
cuda_ml_logistic_reg(
  x,
  data,
  fit_intercept = TRUE,
  penalty = c("l2", "l1", "elasticnet", "none"),
```

```
  tol = 1e-04,
  C = 1,
  class_weight = NULL,
  sample_weight = NULL,
  max_iters = 1000L,
  linesearch_max_iters = 50L,
  l1_ratio = NULL,
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`fit_intercept`	If TRUE, then the model tries to correct for the global mean of the response variable. If FALSE, then the model expects data to be centered. Default: TRUE.
`penalty`	The penalty type, must be one of "none", "l1", "l2", "elasticnet". If "none" or "l2" is selected, then L-BFGS solver will be used. If "l1" is selected, solver OWL-QN will be used. If "elasticnet" is selected, OWL-QN will be used if l1_ratio > 0, otherwise L-BFGS will be used. Default: "l2".
`tol`	Tolerance for stopping criteria. Default: 1e-4.
`C`	Inverse of regularization strength; must be a positive float. Default: 1.0.
`class_weight`	If `NULL`, then each class has equal weight of `1`. If `class_weight` is set to `"balanced"`, then weights will be inversely proportional to class frequencies in the input data. If otherwise, then `class_weight` must be a named numeric vector of weight values, with names being class labels. If `class_weight` is not `NULL`, then each entry in `sample_weight` will be adjusted by multiplying its original value with the class weight of the corresponding sample's class. Default: NULL.
`sample_weight`	Array of weights assigned to individual samples. If `NULL`, then each sample has an equal weight of 1. Default: NULL.
`max_iters`	Maximum number of solver iterations. Default: 1000L.
`linesearch_max_iters`	Max number of linesearch iterations per outer iteration used in the LBFGS- and OWL- QN solvers. Default: 50L.
`l1_ratio`	The Elastic-Net mixing parameter, must `NULL` or be within the range of [0, 1]. Default: NULL.
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Examples

```
library(cuda.ml)

X <- scale(as.matrix(iris[names(iris) != "Species"]))
y <- iris$Species

model <- cuda_ml_logistic_reg(X, y, max_iters = 100)
predictions <- predict(model, X)

# NOTE: if we were only performing binary classifications (e.g., by having
# `iris_data <- iris %>% mutate(Species = (Species == "setosa"))`), then the
# above would be conceptually equivalent to the following:
#
# iris_data <- iris %>% mutate(Species = (Species == "setosa"))
# model <- glm(
#   Species ~ ., data = iris_data, family = binomial(link = "logit"),
#   control = glm.control(epsilon = 1e-8, maxit = 100)
# )
#
# predict(model, iris_data, type = "response")
```

`cuda_ml_ols` *Train a OLS model.*

Description

Train an Ordinary Least Square (OLS) model for regression tasks.

Usage

```
cuda_ml_ols(x, ...)

## Default S3 method:
cuda_ml_ols(x, ...)

## S3 method for class 'data.frame'
cuda_ml_ols(
  x,
  y,
  method = c("svd", "eig", "qr"),
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)

## S3 method for class 'matrix'
cuda_ml_ols(
  x,
```

```
  y,
  method = c(”svd”, ”eig”, ”qr”),
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)

## S3 method for class 'formula'
cuda_ml_ols(
  formula,
  data,
  method = c(”svd”, ”eig”, ”qr”),
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)

## S3 method for class 'recipe'
cuda_ml_ols(
  x,
  data,
  method = c(”svd”, ”eig”, ”qr”),
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`method`	Must be one of "svd", "eig", "qr". - "svd": compute SVD decomposition using Jacobi iterations. - "eig": use an eigendecomposition of the covariance matrix. - "qr": use the QR decomposition algorithm and solve ‘Rx = Q^T y‘. If the number of features is larger than the sample size, then the "svd" algorithm will be force-selected because it is the only algorithm that can support this type of scenario. Default: "svd".
`fit_intercept`	If TRUE, then the model tries to correct for the global mean of the response variable. If FALSE, then the model expects data to be centered. Default: TRUE.

`normalize_input`	Ignored when `fit_intercept` is FALSE. If TRUE, then the predictors will be normalized to have a L2 norm of 1. Default: FALSE.
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A OLS regressor that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

model <- cuda_ml_ols(formula = mpg ~ ., data = mtcars, method = "qr")
predictions <- predict(model, mtcars[names(mtcars) != "mpg"])

# predictions will be comparable to those from a `stats::lm` model
lm_model <- stats::lm(formula = mpg ~ ., data = mtcars, method = "qr")
lm_predictions <- predict(lm_model, mtcars[names(mtcars) != "mpg"])

print(
  all.equal(
    as.numeric(lm_predictions),
    predictions$.pred,
    tolerance = 1e-3
  )
)
```

`cuda_ml_pca` *Perform principal component analysis.*

Description

Compute principal component(s) of the input data. Each feature from the input will be mean-centered (but not scaled) before the SVD computation takes place.

Usage

```
cuda_ml_pca(
  x,
  n_components = NULL,
  eig_algo = c("dq", "jacobi"),
  tol = 1e-07,
  n_iters = 15L,
```

```
    whiten = FALSE,
    transform_input = TRUE,
   cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace")
 )
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`n_components`	Number of principal component(s) to keep. Default: min(nrow(x), ncol(x)).
`eig_algo`	Eigen decomposition algorithm to be applied to the covariance matrix. Valid choices are "dq" (divid-and-conquer method for symmetric matrices) and "jacobi" (the Jacobi method for symmetric matrices). Default: "dq".
`tol`	Tolerance for singular values computed by the Jacobi method. Default: 1e-7.
`n_iters`	Maximum number of iterations for the Jacobi method. Default: 15.
`whiten`	If TRUE, then de-correlate all components, making each component have unit variance and removing multi-collinearity. Default: FALSE.
`transform_input`	If TRUE, then compute an approximate representation of the input data. Default: TRUE.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.

Value

A PCA model object with the following attributes: - "components": a matrix of `n_components` rows containing the top principal components. - "explained_variance": amount of variance within the input data explained by each component. - "explained_variance_ratio": fraction of variance within the input data explained by each component. - "singular_values": singular values (non-negative) corresponding to the top principal components. - "mean": the column wise mean of `x` which was used to mean-center `x` first. - "transformed_data": (only present if "transform_input" is set to TRUE) an approximate representation of input data based on principal components. - "pca_params": opaque pointer to PCA parameters which will be used for performing inverse transforms.

The model object can be used as input to the inverse_transform() function to map a representation based on principal components back to the original feature space.

Examples

```
library(cuda.ml)

iris.pca <- cuda_ml_pca(iris[1:4], n_components = 3)
print(iris.pca)
```

`cuda_ml_rand_forest` *Train a random forest model.*

Description

Train a random forest model for classification or regression tasks.

Usage

```
cuda_ml_rand_forest(x, ...)

## Default S3 method:
cuda_ml_rand_forest(x, ...)

## S3 method for class 'data.frame'
cuda_ml_rand_forest(
  x,
  y,
  mtry = NULL,
  trees = NULL,
  min_n = 2L,
  bootstrap = TRUE,
  max_depth = 16L,
  max_leaves = Inf,
  max_predictors_per_note_split = NULL,
  n_bins = 128L,
  min_samples_leaf = 1L,
  split_criterion = NULL,
  min_impurity_decrease = 0,
  max_batch_size = 128L,
  n_streams = 8L,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
  ...
)

## S3 method for class 'matrix'
cuda_ml_rand_forest(
  x,
  y,
  mtry = NULL,
  trees = NULL,
  min_n = 2L,
  bootstrap = TRUE,
  max_depth = 16L,
  max_leaves = Inf,
  max_predictors_per_note_split = NULL,
  n_bins = 128L,
```

```
    min_samples_leaf = 1L,
    split_criterion = NULL,
    min_impurity_decrease = 0,
    max_batch_size = 128L,
    n_streams = 8L,
   cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
    ...
  )

  ## S3 method for class 'formula'
  cuda_ml_rand_forest(
    formula,
    data,
    mtry = NULL,
    trees = NULL,
    min_n = 2L,
    bootstrap = TRUE,
    max_depth = 16L,
    max_leaves = Inf,
    max_predictors_per_note_split = NULL,
    n_bins = 128L,
    min_samples_leaf = 1L,
    split_criterion = NULL,
    min_impurity_decrease = 0,
    max_batch_size = 128L,
    n_streams = 8L,
   cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
    ...
  )

  ## S3 method for class 'recipe'
  cuda_ml_rand_forest(
    x,
    data,
    mtry = NULL,
    trees = NULL,
    min_n = 2L,
    bootstrap = TRUE,
    max_depth = 16L,
    max_leaves = Inf,
    max_predictors_per_note_split = NULL,
    n_bins = 128L,
    min_samples_leaf = 1L,
    split_criterion = NULL,
    min_impurity_decrease = 0,
    max_batch_size = 128L,
    n_streams = 8L,
   cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
```

```
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`mtry`	The number of predictors that will be randomly sampled at each split when creating the tree models. Default: the square root of the total number of predictors.
`trees`	An integer for the number of trees contained in the ensemble. Default: 100L.
`min_n`	An integer for the minimum number of data points in a node that are required for the node to be split further. Default: 2L.
`bootstrap`	Whether to perform bootstrap. If TRUE, each tree in the forest is built on a bootstrapped sample with replacement. If FALSE, the whole dataset is used to build each tree.
`max_depth`	Maximum tree depth. Default: 16L.
`max_leaves`	Maximum leaf nodes per tree. Soft constraint. Default: Inf (unlimited).
`max_predictors_per_note_split`	Number of predictor to consider per node split. Default: square root of the total number predictors.
`n_bins`	Number of bins used by the split algorithm. Default: 128L.
`min_samples_leaf`	The minimum number of data points in each leaf node. Default: 1L.
`split_criterion`	The criterion used to split nodes, can be "gini" or "entropy" for classifications, and "mse" or "mae" for regressions. Default: "gini" for classification; "mse" for regression.
`min_impurity_decrease`	Minimum decrease in impurity requried for node to be spilt. Default: 0.
`max_batch_size`	Maximum number of nodes that can be processed in a given batch. Default: 128L.
`n_streams`	Number of CUDA streams to use for building trees. Default: 8L.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A random forest classifier / regressor object that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

# Classification

model <- cuda_ml_rand_forest(
  formula = Species ~ .,
  data = iris,
  trees = 100
)

predictions <- predict(model, iris[names(iris) != "Species"])

# Regression

model <- cuda_ml_rand_forest(
  formula = mpg ~ .,
  data = mtcars,
  trees = 100
)

predictions <- predict(model, mtcars[names(mtcars) != "mpg"])
```

`cuda_ml_rand_proj` *Random projection for dimensionality reduction.*

Description

Generate a random projection matrix for dimensionality reduction, and optionally transform input data to a projection in a lower dimension space using the generated random matrix.

Usage

```
cuda_ml_rand_proj(
  x,
  n_components = NULL,
  eps = 0.1,
  gaussian_method = TRUE,
  density = NULL,
  transform_input = TRUE,
  seed = 0L
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`n_components`	Dimensionality of the target projection space. If NULL, then the parameter is deducted using the Johnson-Lindenstrauss lemma, taking into consideration the number of samples and the eps parameter. Default: NULL.
`eps`	Error tolerance during projection. Default: 0.1.
`gaussian_method`	If TRUE, then use the Gaussian random projection method. Otherwise, use the sparse random projection method. See https://en.wikipedia.org/wiki/Random_projection for details. Default: TRUE.
`density`	Ratio of non-zero component in the random projection matrix. If NULL, then the value is set to the minimum density as recommended by Ping Li et al.: 1 / sqrt(n_features). Default: NULL.
`transform_input`	Whether to project input data onto a lower dimension space using the random matrix. Default: TRUE.
`seed`	Seed for the pseudorandom number generator. Default: 0L.

Value

A context object containing GPU pointer to a random matrix that can be used as input to the `cuda_ml_transform()` function. If `transform_input` is set to TRUE, then the context object will also contain a "transformed_data" attribute containing the lower dimensional projection of the input data.

Examples

```
library(cuda.ml)
library(mlbench)

data(Vehicle)
vehicle_data <- Vehicle[order(Vehicle$Class), which(names(Vehicle) != "Class")]

model <- cuda_ml_rand_proj(vehicle_data, n_components = 4)

set.seed(0L)
print(kmeans(model$transformed_data, centers = 4, iter.max = 1000))
```

`cuda_ml_ridge` *Train a linear model using ridge regression.*

Description

Train a linear model with L2 regularization.

Usage

```
cuda_ml_ridge(x, ...)

## Default S3 method:
cuda_ml_ridge(x, ...)

## S3 method for class 'data.frame'
cuda_ml_ridge(
  x,
  y,
  alpha = 1,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)

## S3 method for class 'matrix'
cuda_ml_ridge(
  x,
  y,
  alpha = 1,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)

## S3 method for class 'formula'
cuda_ml_ridge(
  formula,
  data,
  alpha = 1,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)

## S3 method for class 'recipe'
cuda_ml_ridge(
  x,
  data,
  alpha = 1,
  fit_intercept = TRUE,
  normalize_input = FALSE,
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`alpha`	Multiplier of the L2 penalty term (i.e., the result would become and Ordinary Least Square model if `alpha` were set to 0). Default: 1.
`fit_intercept`	If TRUE, then the model tries to correct for the global mean of the response variable. If FALSE, then the model expects data to be centered. Default: TRUE.
`normalize_input`	Ignored when `fit_intercept` is FALSE. If TRUE, then the predictors will be normalized to have a L2 norm of 1. Default: FALSE.
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A ridge regressor that can be used with the ’predict’ S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

model <- cuda_ml_ridge(formula = mpg ~ ., data = mtcars, alpha = 1e-3)
cuda_ml_predictions <- predict(model, mtcars[names(mtcars) != "mpg"])

# predictions will be comparable to those from a `glmnet` model with `lambda`
# set to 2e-3 and `alpha` set to 0
# (in `glmnet`, `lambda` is the weight of the penalty term, and `alpha` is
#  the elastic mixing parameter between L1 and L2 penalties.

library(glmnet)

glmnet_model <- glmnet(
  x = as.matrix(mtcars[names(mtcars) != "mpg"]), y = mtcars$mpg,
  alpha = 0, lambda = 2e-3, nlambda = 1, standardize = FALSE
)

glmnet_predictions <- predict(
  glmnet_model, as.matrix(mtcars[names(mtcars) != "mpg"]),
  s = 0
)
```

```
print(
  all.equal(
    as.numeric(glmnet_predictions),
    cuda_ml_predictions$.pred,
    tolerance = 1e-3
  )
)
```

`cuda_ml_serialize` *Serialize a CuML model*

Description

Given a CuML model, serialize its state into a connection.

Usage

```
cuda_ml_serialize(model, connection = NULL, ...)

cuda_ml_serialise(model, connection = NULL, ...)
```

Arguments

`model`	The model object.
`connection`	An open connection or `NULL`. If `NULL`, then the model state is serialized to a raw vector. Default: NULL.
`...`	Additional arguments to `base::serialize()`.

Value

`NULL` unless `connection` is `NULL`, in which case the serialized model state is returned as a raw vector.

See Also

`serialize`

`cuda_ml_sgd` *Train a MBSGD linear model.*

Description

Train a linear model using mini-batch stochastic gradient descent.

Usage

```
cuda_ml_sgd(x, ...)

## Default S3 method:
cuda_ml_sgd(x, ...)

## S3 method for class 'data.frame'
cuda_ml_sgd(
  x,
  y,
  fit_intercept = TRUE,
  loss = c("squared_loss", "log", "hinge"),
  penalty = c("none", "l1", "l2", "elasticnet"),
  alpha = 1e-04,
  l1_ratio = 0.5,
  epochs = 1000L,
  tol = 0.001,
  shuffle = TRUE,
  learning_rate = c("constant", "invscaling", "adaptive"),
  eta0 = 0.001,
  power_t = 0.5,
  batch_size = 32L,
  n_iters_no_change = 5L,
  ...
)

## S3 method for class 'matrix'
cuda_ml_sgd(
  x,
  y,
  fit_intercept = TRUE,
  loss = c("squared_loss", "log", "hinge"),
  penalty = c("none", "l1", "l2", "elasticnet"),
  alpha = 1e-04,
  l1_ratio = 0.5,
  epochs = 1000L,
  tol = 0.001,
  shuffle = TRUE,
  learning_rate = c("constant", "invscaling", "adaptive"),
```

```
  eta0 = 0.001,
  power_t = 0.5,
  batch_size = 32L,
  n_iters_no_change = 5L,
  ...
)

## S3 method for class 'formula'
cuda_ml_sgd(
  formula,
  data,
  fit_intercept = TRUE,
  loss = c("squared_loss", "log", "hinge"),
  penalty = c("none", "l1", "l2", "elasticnet"),
  alpha = 1e-04,
  l1_ratio = 0.5,
  epochs = 1000L,
  tol = 0.001,
  shuffle = TRUE,
  learning_rate = c("constant", "invscaling", "adaptive"),
  eta0 = 0.001,
  power_t = 0.5,
  batch_size = 32L,
  n_iters_no_change = 5L,
  ...
)

## S3 method for class 'recipe'
cuda_ml_sgd(
  x,
  data,
  fit_intercept = TRUE,
  loss = c("squared_loss", "log", "hinge"),
  penalty = c("none", "l1", "l2", "elasticnet"),
  alpha = 1e-04,
  l1_ratio = 0.5,
  epochs = 1000L,
  tol = 0.001,
  shuffle = TRUE,
  learning_rate = c("constant", "invscaling", "adaptive"),
  eta0 = 0.001,
  power_t = 0.5,
  batch_size = 32L,
  n_iters_no_change = 5L,
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`fit_intercept`	If TRUE, then the model tries to correct for the global mean of the response variable. If FALSE, then the model expects data to be centered. Default: TRUE.
`loss`	Loss function, must be one of "squared_loss", "log", "hinge".
`penalty`	Type of regularization to perform, must be one of "none", "l1", "l2", "elasticnet". - "none": no regularization. - "l1": perform regularization based on the L1-norm (LASSO) which tries to minimize the sum of the absolute values of the coefficients. - "l2": perform regularization based on the L2 norm (Ridge) which tries to minimize the sum of the square of the coefficients. - "elasticnet": perform the Elastic Net regularization which is based on the weighted averable of L1 and L2 norms. Default: "none".
`alpha`	Multiplier of the penalty term. Default: 1e-4.
`l1_ratio`	The ElasticNet mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an L2 penalty. For l1_ratio = 1 it is an L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2. The penalty term is computed using the following formula: penalty = `alpha` * `l1_ratio` * ||w||_1 + 0.5 * `alpha` * (1 - `l1_ratio`) * ||w||^2_2 where ||w||_1 is the L1 norm of the coefficients, and ||w||_2 is the L2 norm of the coefficients.
`epochs`	The number of times the model should iterate through the entire dataset during training. Default: 1000L.
`tol`	Threshold for stopping training. Training will stop if (loss in current epoch) > (loss in previous epoch) - `tol`. Default: 1e-3.
`shuffle`	Whether to shuffles the training data after each epoch. Default: True.
`learning_rate`	Must be one of "constant", "invscaling", "adaptive". - "constant": the learning rate will be kept constant. - "invscaling": (learning rate) = (initial learning rate) / pow(t, power_t) where `t` is the number of epochs and `power_t` is a tunable parameter of this model. - "adaptive": (learning rate) = (initial learning rate) as long as the training loss keeps decreasing. Each time the last `n_iter_no_change` consecutive epochs fail to decrease the training loss by `tol`, the current learning rate is divided by 5. Default: "constant".
`eta0`	The initial learning rate. Default: 1e-3.
`power_t`	The exponent used in the invscaling learning rate calculations.
`batch_size`	The number of samples that will be included in each batch. Default: 32L.
`n_iters_no_change`	The maximum number of epochs to train if there is no imporvement in the model. Default: 5.

`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A linear model that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

model <- cuda_ml_sgd(
  mpg ~ ., mtcars,
  batch_size = 4L, epochs = 50000L,
  learning_rate = "adaptive", eta0 = 1e-5,
  penalty = "l2", alpha = 1e-5, tol = 1e-6,
  n_iters_no_change = 10L
)

preds <- predict(model, mtcars[names(mtcars) != "mpg"])
print(all.equal(preds$.pred, mtcars$mpg, tolerance = 0.09))
```

`cuda_ml_svm` *Train a SVM model.*

Description

Train a Support Vector Machine model for classification or regression tasks.

Usage

```
cuda_ml_svm(x, ...)

## Default S3 method:
cuda_ml_svm(x, ...)

## S3 method for class 'data.frame'
cuda_ml_svm(
  x,
  y,
  cost = 1,
  kernel = c("rbf", "tanh", "polynomial", "linear"),
  gamma = NULL,
  coef0 = 0,
  degree = 3L,
```

```
  tol = 0.001,
  max_iter = NULL,
  nochange_steps = 1000L,
  cache_size = 1024,
  epsilon = 0.1,
  sample_weights = NULL,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
  ...
)

## S3 method for class 'matrix'
cuda_ml_svm(
  x,
  y,
  cost = 1,
  kernel = c("rbf", "tanh", "polynomial", "linear"),
  gamma = NULL,
  coef0 = 0,
  degree = 3L,
  tol = 0.001,
  max_iter = NULL,
  nochange_steps = 1000L,
  cache_size = 1024,
  epsilon = 0.1,
  sample_weights = NULL,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
  ...
)

## S3 method for class 'formula'
cuda_ml_svm(
  formula,
  data,
  cost = 1,
  kernel = c("rbf", "tanh", "polynomial", "linear"),
  gamma = NULL,
  coef0 = 0,
  degree = 3L,
  tol = 0.001,
  max_iter = NULL,
  nochange_steps = 1000L,
  cache_size = 1024,
  epsilon = 0.1,
  sample_weights = NULL,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
  ...
)
```

```
## S3 method for class 'recipe'
cuda_ml_svm(
  x,
  data,
  cost = 1,
  kernel = c("rbf", "tanh", "polynomial", "linear"),
  gamma = NULL,
  coef0 = 0,
  degree = 3L,
  tol = 0.001,
  max_iter = NULL,
  nochange_steps = 1000L,
  cache_size = 1024,
  epsilon = 0.1,
  sample_weights = NULL,
 cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
  ...
)
```

Arguments

`x`	Depending on the context: * A __data frame__ of predictors. * A __matrix__ of predictors. * A __recipe__ specifying a set of preprocessing steps * created from [recipes::recipe()]. * A __formula__ specifying the predictors and the outcome.
`...`	Optional arguments; currently unused.
`y`	A numeric vector (for regression) or factor (for classification) of desired responses.
`cost`	A positive number for the cost of predicting a sample within or on the wrong side of the margin. Default: 1.
`kernel`	Type of the SVM kernel function (must be one of "rbf", "tanh", "polynomial", or "linear"). Default: "rbf".
`gamma`	The gamma coefficient (only relevant to polynomial, RBF, and tanh kernel functions, see explanations below). Default: 1 / (num features). The following kernels are implemented: - RBF K(x_1, x_2) = exp(-gamma |x_1-x_2|^2) - TANH K(x_1, x_2) = tanh(gamma <x_1,x_2> + coef0) - POLYNOMIAL K(x_1, x_2) = (gamma <x_1,x_2> + coef0)^degree - LINEAR K(x_1,x_2) = <x_1,x_2>, where < , > denotes the dot product.
`coef0`	The 0th coefficient (only applicable to polynomial and tanh kernel functions, see explanations below). Default: 0. The following kernels are implemented: - RBF K(x_1, x_2) = exp(-gamma |x_1-x_2|^2) - TANH K(x_1, x_2) = tanh(gamma <x_1,x_2> + coef0) - POLYNOMIAL K(x_1, x_2) = (gamma <x_1,x_2> + coef0)^degree - LINEAR K(x_1,x_2) = <x_1,x_2>, where < , > denotes the dot product.
`degree`	Degree of the polynomial kernel function (note: not applicable to other kernel types, see explanations below). Default: 3.

	The following kernels are implemented: - RBF K(x_1, x_2) = exp(-gamma |x_1-x_2|^2) - TANH K(x_1, x_2) = tanh(gamma <x_1,x_2> + coef0) - POLYNOMIAL K(x_1, x_2) = (gamma <x_1,x_2> + coef0)^degree - LINEAR K(x_1,x_2) = <x_1,x_2>, where < , > denotes the dot product.
`tol`	Tolerance to stop fitting. Default: 1e-3.
`max_iter`	Maximum number of outer iterations in SmoSolver. Default: 100 * (num samples).
`nochange_steps`	Number of steps with no change w.r.t convergence. Default: 1000.
`cache_size`	Size of kernel cache (MiB) in device memory. Default: 1024.
`epsilon`	Espsilon parameter of the epsilon-SVR model. There is no penalty for points that are predicted within the epsilon-tube around the target values. Please note this parameter is only relevant for regression tasks. Default: 0.1.
`sample_weights`	Optional weight assigned to each input data point.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.
`formula`	A formula specifying the outcome terms on the left-hand side, and the predictor terms on the right-hand side.
`data`	When a __recipe__ or __formula__ is used, `data` is specified as a __data frame__ containing the predictors and (if applicable) the outcome.

Value

A SVM classifier / regressor object that can be used with the 'predict' S3 generic to make predictions on new data points.

Examples

```
library(cuda.ml)

# Classification

model <- cuda_ml_svm(
  formula = Species ~ .,
  data = iris,
  kernel = "rbf"
)

predictions <- predict(model, iris[names(iris) != "Species"])

# Regression

model <- cuda_ml_svm(
  formula = mpg ~ .,
  data = mtcars,
  kernel = "rbf"
)

predictions <- predict(model, mtcars)
```

`cuda_ml_transform` *Transform data using a trained cuML model.*

Description

Given a trained cuML model, transform an input dataset using that model.

Usage

```
cuda_ml_transform(model, x, ...)
```

Arguments

`model`	A model object.
`x`	The dataset to be transformed.
`...`	Additional model-specific parameters (if any).

Value

The transformed data points.

`cuda_ml_tsne` *t-distributed Stochastic Neighbor Embedding.*

Description

t-distributed Stochastic Neighbor Embedding (TSNE) for visualizing high- dimensional data.

Usage

```
cuda_ml_tsne(
  x,
  n_components = 2L,
  n_neighbors = ceiling(3 * perplexity),
  method = c("barnes_hut", "fft", "exact"),
  angle = 0.5,
  n_iter = 1000L,
  learning_rate = 200,
  learning_rate_method = c("adaptive", "none"),
  perplexity = 30,
  perplexity_max_iter = 100L,
  perplexity_tol = 1e-05,
  early_exaggeration = 12,
  late_exaggeration = 1,
  exaggeration_iter = 250L,
```

```
  min_grad_norm = 1e-07,
  pre_momentum = 0.5,
  post_momentum = 0.8,
  square_distances = TRUE,
  seed = NULL,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace")
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`n_components`	Dimension of the embedded space.
`n_neighbors`	The number of datapoints to use in the attractive forces. Default: ceiling(3 * perplexity).
`method`	T-SNE method, must be one of "barnes_hut", "fft", "exact". The "exact" method will be more accurate but slower. Both "barnes_hut" and "fft" methods are fast approximations.
`angle`	Valid values are between 0.0 and 1.0, which trade off speed and accuracy, respectively. Generally, these values are set between 0.2 and 0.8. (Barnes-Hut only.)
`n_iter`	Maximum number of iterations for the optimization. Should be at least 250. Default: 1000L.
`learning_rate`	Learning rate of the t-SNE algorithm, usually between (10, 1000). If the learning rate is too high, then t-SNE result could look like a cloud / ball of points.
`learning_rate_method`	Must be one of "adaptive", "none". If "adaptive", then learning rate, early exaggeration, and perplexity are automatically tuned based on input size. Default: "adaptive".
`perplexity`	The target value of the conditional distribution's perplexity (see https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding for details).
`perplexity_max_iter`	The number of epochs the best Gaussian bands are found for. Default: 100L.
`perplexity_tol`	Stop optimizing the Gaussian bands when the conditional distribution's perplexity is within this desired tolerance compared to its taget value. Default: 1e-5.
`early_exaggeration`	Controls the space between clusters. Not critical to tune this. Default: 12.0.
`late_exaggeration`	Controls the space between clusters. It may be beneficial to increase this slightly to improve cluster separation. This will be applied after 'exaggeration_iter' iterations (FFT only).
`exaggeration_iter`	Number of exaggeration iterations. Default: 250L.
`min_grad_norm`	If the gradient norm is below this threshold, the optimization will be stopped. Default: 1e-7.

`pre_momentum`	During the exaggeration iteration, more forcefully apply gradients. Default: 0.5.
`post_momentum`	During the late phases, less forcefully apply gradients. Default: 0.8.
`square_distances`	Whether TSNE should square the distance values.
`seed`	Seed to the psuedorandom number generator. Setting this can make repeated runs look more similar. Note, however, that this highly parallelized t-SNE implementation is not completely deterministic between runs, even with the same `seed` being used for each run. Default: NULL.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.

Value

A matrix containing the embedding of the input data in a low- dimensional space, with each row representing an embedded data point.

Examples

```
library(cuda.ml)

embedding <- cuda_ml_tsne(iris[1:4], method = "exact")

set.seed(0L)
print(kmeans(embedding, centers = 3))
```

`cuda_ml_tsvd` *Truncated SVD.*

Description

Dimensionality reduction using Truncated Singular Value Decomposition.

Usage

```
cuda_ml_tsvd(
  x,
  n_components = 2L,
  eig_algo = c("dq", "jacobi"),
  tol = 1e-07,
  n_iters = 15L,
  transform_input = TRUE,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace")
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`n_components`	Desired dimensionality of output data. Must be strictly less than `ncol(x)` (i.e., the number of features in input data). Default: 2.
`eig_algo`	Eigen decomposition algorithm to be applied to the covariance matrix. Valid choices are "dq" (divid-and-conquer method for symmetric matrices) and "jacobi" (the Jacobi method for symmetric matrices). Default: "dq".
`tol`	Tolerance for singular values computed by the Jacobi method. Default: 1e-7.
`n_iters`	Maximum number of iterations for the Jacobi method. Default: 15.
`transform_input`	If TRUE, then compute an approximate representation of the input data. Default: TRUE.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.

Value

A TSVD model object with the following attributes: - "components": a matrix of `n_components` rows to be used for dimensionalitiy reduction on new data points. - "explained_variance": (only present if "transform_input" is set to TRUE) amount of variance within the input data explained by each component. - "explained_variance_ratio": (only present if "transform_input" is set to TRUE) fraction of variance within the input data explained by each component. - "singular_values": The singular values corresponding to each component. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space. - "tsvd_params": opaque pointer to TSVD parameters which will be used for performing inverse transforms.

Examples

```
library(cuda.ml)

iris.tsvd <- cuda_ml_tsvd(iris[1:4], n_components = 2)
print(iris.tsvd)
```

`cuda_ml_umap` *Uniform Manifold Approximation and Projection (UMAP) for dimension reduction.*

Description

Run the Uniform Manifold Approximation and Projection (UMAP) algorithm to find a low dimensional embedding of the input data that approximates an underlying manifold.

Usage

```
cuda_ml_umap(
  x,
  y = NULL,
  n_components = 2L,
  n_neighbors = 15L,
  n_epochs = 500L,
  learning_rate = 1,
  init = c("spectral", "random"),
  min_dist = 0.1,
  spread = 1,
  set_op_mix_ratio = 1,
  local_connectivity = 1L,
  repulsion_strength = 1,
  negative_sample_rate = 5L,
  transform_queue_size = 4,
  a = NULL,
  b = NULL,
  target_n_neighbors = n_neighbors,
  target_metric = c("categorical", "euclidean"),
  target_weight = 0.5,
  transform_input = TRUE,
  seed = NULL,
 cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace")
)
```

Arguments

`x`	The input matrix or dataframe. Each data point should be a row and should consist of numeric values only.
`y`	An optional numeric vector of target values for supervised dimension reduction. Default: NULL.
`n_components`	The dimension of the space to embed into. Default: 2.
`n_neighbors`	The size of local neighborhood (in terms of number of neighboring sample points) used for manifold approximation. Default: 15.
`n_epochs`	The number of training epochs to be used in optimizing the low dimensional embedding. Default: 500.
`learning_rate`	The initial learning rate for the embedding optimization. Default: 1.0.
`init`	Initialization mode of the low dimensional embedding. Must be one of "spectral", "random". Default: "spectral".
`min_dist`	The effective minimum distance between embedded points. Default: 0.1.
`spread`	The effective scale of embedded points. In combination with `min_dist` this determines how clustered/clumped the embedded points are. Default: 1.0.
`set_op_mix_ratio`	Interpolate between (fuzzy) union and intersection as the set operation used to combine local fuzzy simplicial sets to obtain a global fuzzy simplicial sets. Both

	fuzzy set operations use the product t-norm. The value of this parameter should be between 0.0 and 1.0; a value of 1.0 will use a pure fuzzy union, while 0.0 will use a pure fuzzy intersection. Default: 1.0.
`local_connectivity`	The local connectivity required – i.e. the number of nearest neighbors that should be assumed to be connected at a local level. Default: 1.
`repulsion_strength`	Weighting applied to negative samples in low dimensional embedding optimization. Values higher than one will result in greater weight being given to negative samples. Default: 1.0.
`negative_sample_rate`	The number of negative samples to select per positive sample in the optimization process. Default: 5.
`transform_queue_size`	For transform operations (embedding new points using a trained model this will control how aggressively to search for nearest neighbors. Default: 4.0.
`a, b`	More specific parameters controlling the embedding. If not set, then these values are set automatically as determined by `min_dist` and `spread`. Default: NULL.
`target_n_neighbors`	The number of nearest neighbors to use to construct the target simplcial set. Default: n_neighbors.
`target_metric`	The metric for measuring distance between the actual and and the target values (`y`) if using supervised dimension reduction. Must be one of "categorical", "euclidean". Default: "categorical".
`target_weight`	Weighting factor between data topology and target topology. A value of 0.0 weights entirely on data, a value of 1.0 weights entirely on target. The default of 0.5 balances the weighting equally between data and target.
`transform_input`	If TRUE, then compute an approximate representation of the input data. Default: TRUE.
`seed`	Optional seed for pseudo random number generator. Default: NULL. Setting a PRNG seed will enable consistency of trained embeddings, allowing for reproducible results to 3 digits of precision, but at the expense of potentially slower training and increased memory usage. If the PRNG seed is not set, then the trained embeddings will not be deterministic.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.

Value

A UMAP model object that can be used as input to the `cuda_ml_transform()` function. If `transform_input` is set to TRUE, then the model object will contain a "transformed_data" attribute containing the lower dimensional embedding of the input data.

Examples

```
library(cuda.ml)

model <- cuda_ml_umap(
  x = iris[1:4],
  y = iris[[5]],
  n_components = 2,
  n_epochs = 200,
  transform_input = TRUE
)

set.seed(0L)
print(kmeans(model$transformed, iter.max = 100, centers = 3))
```

`cuda_ml_unserialize` *Unserialize a CuML model state*

Description

Unserialize a CuML model state into a CuML model object.

Usage

```
cuda_ml_unserialize(connection, ...)

cuda_ml_unserialise(connection, ...)
```

Arguments

`connection`	An open connection or a raw vector.
`...`	Additional arguments to `base::unserialize()`.

Value

A unserialized CuML model.

See Also

`unserialize`

`cuML_major_version` *Get the major version of the RAPIDS cuML shared library cuda.ml was linked to.*

Description

Get the major version of the RAPIDS cuML shared library cuda.ml was linked to.

Usage

```
cuML_major_version()
```

Value

The major version of the RAPIDS cuML shared library cuda.ml was linked to in a character vector, or `NA_character_` if cuda.ml was not linked to any version of RAPIDS cuML.

Examples

```
library(cuda.ml)

print(cuML_major_version())
```

`cuML_minor_version` *Get the minor version of the RAPIDS cuML shared library cuda.ml was linked to.*

Description

Get the minor version of the RAPIDS cuML shared library cuda.ml was linked to.

Usage

```
cuML_minor_version()
```

Value

The minor version of the RAPIDS cuML shared library cuda.ml was linked to in a character vector, or `NA_character_` if cuda.ml was not linked to any version of RAPIDS cuML.

Examples

```
library(cuda.ml)

print(cuML_minor_version())
```

`has_cuML` *Determine whether cuda.ml was linked to a valid version of the RAPIDS cuML shared library.*

Description

Determine whether cuda.ml was linked to a valid version of the RAPIDS cuML shared library.

Usage

```
has_cuML()
```

Value

A logical value indicating whether the current installation cuda.ml was linked to a valid version of the RAPIDS cuML shared library.

Examples

```
library(cuda.ml)

if (!has_cuML()) {
  warning(
    "Please install the RAPIDS cuML shared library first, and then re-",
    "install {cuda.ml}."
  )
}
```

`predict.cuda_ml_fil` *Make predictions on new data points.*

Description

Make predictions on new data points using a FIL model.

Usage

```
## S3 method for class 'cuda_ml_fil'
predict(object, x, output_class_probabilities = FALSE, ...)
```

Arguments

`object`	A trained CuML model.
`x`	A matrix or dataframe containing new data points.
`output_class_probabilities`	Whether to output class probabilities. NOTE: setting `output_class_probabilities` to `TRUE` is only valid when the model being applied is a classification model and supports class probabilities output. CuML classification models supporting class probabilities include `knn`, `fil`, and `rand_forest`. A warning message will be emitted if `output_class_probabilities` is set to `TRUE` or `FALSE` but the model being applied does not support class probabilities output.
`...`	Additional arguments to `predict()`. Currently unused.

Value

Predictions on new data points.

`predict.cuda_ml_knn` *Make predictions on new data points.*

Description

Make predictions on new data points using a CuML KNN model.

Usage

```
## S3 method for class 'cuda_ml_knn'
predict(object, x, output_class_probabilities = NULL, ...)
```

Arguments

`object`	A trained CuML model.
`x`	A matrix or dataframe containing new data points.
`output_class_probabilities`	Whether to output class probabilities. NOTE: setting `output_class_probabilities` to `TRUE` is only valid when the model being applied is a classification model and supports class probabilities output. CuML classification models supporting class probabilities include `knn`, `fil`, and `rand_forest`. A warning message will be emitted if `output_class_probabilities` is set to `TRUE` or `FALSE` but the model being applied does not support class probabilities output.
`...`	Additional arguments to `predict()`. Currently unused.

Value

Predictions on new data points.

`predict.cuda_ml_linear_model`

Make predictions on new data points.

Description

Make predictions on new data points using a linear model.

Usage

```
## S3 method for class 'cuda_ml_linear_model'
predict(object, x, ...)
```

Arguments

`object`	A trained CuML model.
`x`	A matrix or dataframe containing new data points.
`...`	Additional arguments to `predict()`. Currently unused.

Value

Predictions on new data points.

`predict.cuda_ml_logistic_reg`

Make predictions on new data points.

Description

Make predictions on new data points using a CuML logistic regression model.

Usage

```
## S3 method for class 'cuda_ml_logistic_reg'
predict(object, x, ...)
```

Arguments

`object`	A trained CuML model.
`x`	A matrix or dataframe containing new data points.
`...`	Additional arguments to `predict()`. Currently unused.

Value

Predictions on new data points.

`predict.cuda_ml_rand_forest` *Make predictions on new data points.*

Description

Make predictions on new data points using a CuML random forest model.

Usage

```
## S3 method for class 'cuda_ml_rand_forest'
predict(
  object,
  x,
  output_class_probabilities = NULL,
  cuML_log_level = c("off", "critical", "error", "warn", "info", "debug", "trace"),
  ...
)
```

Arguments

`object`	A trained CuML model.
`x`	A matrix or dataframe containing new data points.
`output_class_probabilities`	Whether to output class probabilities. NOTE: setting `output_class_probabilities` to `TRUE` is only valid when the model being applied is a classification model and supports class probabilities output. CuML classification models supporting class probabilities include `knn`, `fil`, and `rand_forest`. A warning message will be emitted if `output_class_probabilities` is set to `TRUE` or `FALSE` but the model being applied does not support class probabilities output.
`cuML_log_level`	Log level within cuML library functions. Must be one of "off", "critical", "error", "warn", "info", "debug", "trace". Default: off.
`...`	Additional arguments to `predict()`. Currently unused.

Value

Predictions on new data points.

`predict.cuda_ml_svm` *Make predictions on new data points.*

Description

Make predictions on new data points using a CuML SVM model.

Usage

```
## S3 method for class 'cuda_ml_svm'
predict(object, x, ...)
```

Arguments

`object`	A trained CuML model.
`x`	A matrix or dataframe containing new data points.
`...`	Additional arguments to `predict()`. Currently unused.

Value

Predictions on new data points.

Index