Package: mlpack (via r-universe)

October 28, 2026

Title 'Rcpp' Integration for the 'mlpack' Library

Version 4.8.0-1

Date 2026-07-20

Description A fast, flexible machine learning library, written in C++, that aims to provide fast, extensible implementations of cutting-edge machine learning algorithms. See also Curtin et al. (2023) <doi:10.21105/joss.05026>.

License BSD_3_clause + file LICENSE

Depends R (>= 4.3.0)

Imports Rcpp (>= 0.12.12)

LinkingTo Rcpp, RcppArmadillo (>= 0.10.8.2), RcppEnsmallen (>= 0.2.10.0)

Suggests testthat (>= 2.1.0)

URL https://www.mlpack.org/doc/user/bindings/r.html, https://github.com/mlpack/mlpack

BugReports https://github.com/mlpack/mlpack/issues

Encoding UTF-8

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Yashwant Singh Parihar [aut, ctb, cph], Ryan Curtin [aut, ctb, cph, cre], Dirk Eddelbuettel [aut, ctb, cph], James Balamuta [aut, ctb, cph], Bill March [ctb, cph], Dongryeol Lee [ctb, cph], Nishant Mehta [ctb, cph], Parikshit Ram [ctb, cph], James Cline [ctb, cph], Sterling Peet [ctb, cph], Matthew Amidon [ctb, cph], Neil Slagle [ctb, cph], Ajinkya Kale [ctb, cph], Vlad Grantcharov [ctb, cph], Noah Kauffman [ctb, cph], Rajendran Mohan [ctb, cph], Trironk Kiatkungwanglai [ctb, cph], Patrick Mason [ctb, cph], Marcus Edel [ctb, cph], Mudit Raj Gupta [ctb, cph], Sumedh Ghaisas [ctb, cph], Michael Fox [ctb, cph], Siddharth Agrawal [ctb, cph], Saheb Motiani [ctb, cph], Yash Vadalia [ctb, cph], Abhishek Laddha [ctb, cph], Vahab

Akbarzadeh [ctb, cph], Andrew Wells [ctb, cph], Zhihao Lou [ctb, cph], Udit Saxena [ctb, cph], Stephen Tu [ctb, cph], Jaskaran Singh [ctb, cph], Hritik Jain [ctb, cph], Vladimir Glazachev [ctb, cph], QiaoAn Chen [ctb, cph], Janzen Brewer [ctb, cph], Trung Dinh [ctb, cph], Tham Ngap Wei [ctb, cph], Grzegorz Krajewski [ctb, cph], Joseph Mariadassou [ctb, cph], Pavel Zhigulin [ctb, cph], Andy Fang [ctb, cph], Barak Pearlmutter [ctb, cph], Ivari Horm [ctb, cph], Dhawal Arora [ctb, cph], Alexander Leinoff [ctb, cph], Palash Ahuja [ctb, cph], Yannis Mentekidis [ctb, cph], Ranjan Mondal [ctb, cph], Mikhail Lozhnikov [ctb, cph], Marcos Pividori [ctb, cph], Keon Kim [ctb, cph], Nilay Jain [ctb, cph], Peter Lehner [ctb, cph], Anuraj Kanodia [ctb, cph], Ivan Georgiev [ctb, cph], Shikhar Bhardwaj [ctb, cph], Yashu Seth [ctb, cph], Mike Izbicki [ctb, cph], Sudhanshu Ranjan [ctb, cph], Piyush Jaiswal [ctb, cph], Dinesh Raj [ctb, cph], Vivek Pal [ctb, cph], Prasanna Patil [ctb, cph], Lakshya Agrawal [ctb, cph], Praveen Ch [ctb, cph], Kirill Mishchenko [ctb, cph], Abhinav Moudgil [ctb, cph], Thyrix Yang [ctb, cph], Sagar B Hathwar [ctb, cph], Nishanth Hegde [ctb, cph], Parminder Singh [ctb, cph], CodeAi [ctb, cph], Franciszek Stokowacki [ctb, cph], Samikshya Chand [ctb, cph], N Rajiv Vaidyanathan [ctb, cph], Kartik Nighania [ctb, cph], Eugene Freyman [ctb, cph], Manish Kumar [ctb, cph], Haritha Sreedharan Nair [ctb, cph], Sourabh Varshney [ctb, cph], Projyal Dev [ctb, cph], Nikhil Goel [ctb, cph], Shikhar Jaiswal [ctb, cph], B Kartheek Reddy [ctb, cph], Atharva Khandait [ctb, cph], Wenhao Huang [ctb, cph], Roberto Hueso [ctb, cph], Prabhat Sharma [ctb, cph], Tan Jun An [ctb, cph], Moksh Jain [ctb, cph], Manthan-R-Sheth [ctb, cph], Namrata Mukhija [ctb, cph], Conrad Sanderson [ctb, cph], Thanasis Mattas [ctb, cph], Shashank Shekhar [ctb, cph], Yasmine Dumouchel [ctb, cph], German Lancioni [ctb, cph], Arash Abghari [ctb, cph], Ayush Chamoli [ctb, cph], Tommi Laivamaa [ctb, cph], Kim SangYeon [ctb, cph], Niteya Shah [ctb, cph], Toshal Agrawal [ctb, cph], Dan Timson [ctb, cph], Miguel Canteras [ctb, cph], Bishwa Karki [ctb, cph], Mehul Kumar Nirala [ctb, cph], Heet Sankesara [ctb, cph], Jeffin Sam [ctb, cph], Vikas S Shetty [ctb, cph], Khizir Siddiqui [ctb, cph], Tejasvi Tomar [ctb, cph], Jai Agarwal [ctb, cph], Ziyang Jiang [ctb, cph], Rohit Kartik [ctb, cph], Aditya Viki [ctb, cph], Kartik Dutt [ctb, cph], Suryoday Basak [ctb, cph], Sriram S K [ctb, cph], Manoranjan Kumar Bharti (Nakul Bharti) [ctb, cph], Saraansh Tandon [ctb, cph], Gaurav Singh [ctb, cph], Lakshya Ojha [ctb, cph], Bisakh Mondal [ctb, cph], Benson Muite [ctb, cph], Sarthak Bhardwaj [ctb, cph], Aakash Kaushik [ctb, cph], Anush Kini [ctb, cph], Nippun Sharma [ctb, cph], Rishabh Garg [ctb, cph], Sudhakar Brar [ctb, cph], Alex Nguyen [ctb, cph], Gaurav Ghati [ctb, cph], Anmolpreet Singh [ctb, cph], Anjishnu

Mukherjee [ctb, cph], Omar Shrit [ctb, cph], Tru Hoang [ctb, cph], Mark Fischinger [ctb, cph], Muhammad Fawwaz Mayda [ctb, cph], Roshan Nrusing Swain [ctb, cph], Suvarsha Chennareddy [ctb, cph], Shubham Agrawal [ctb, cph], Sri Madhan M [ctb, cph], Zhuojin Liu [ctb, cph], Richèl Bilderbeek [ctb, cph], Chetan Pandey [ctb, cph], Nikolay Apanasov [ctb, cph], Martin Lambertsen [ctb, cph], Andrea Novellini [ctb, cph], Felix Patschkowski [ctb, cph], Benjamin A. Beasley [ctb, cph], Maksym Prots [ctb, cph], Zachary Ng [ctb, cph], Ranjodh Singh [ctb, cph], Mohammad Mundiwala [ctb, cph]

Maintainer Ryan Curtin <ryan@ratml.org>

Repository https://cran.r-universe.dev

Date/Publication 2026-07-22 18:38:37 UTC

RemoteUrl https://github.com/cran/mlpack

RemoteRef HEAD

RemoteSha 5f6a34cf268cac9c8854968ce97b0a901523eaa2

Contents

adaboost . . . 5
adaboost_classify . . . 7
adaboost_probabilities . . . 8
adaboost_train . . . 9
approx_kfn . . . 10
bayesian_linear_regression . . . 12
bayesian_linear_regression_predict . . . 15
bayesian_linear_regression_train . . . 16
cf . . . 17
dbscan . . . 20
decision_tree . . . 22
decision_tree_classify . . . 24
decision_tree_probabilities . . . 25
decision_tree_train . . . 26
det . . . 27
emst . . . 29
fastmks . . . 31
gmm_generate . . . 33
gmm_probability . . . 34
gmm_train . . . 35
hmm_generate . . . 37
hmm_loglik . . . 39
hmm_train . . . 40
hmm_viterbi . . . 41
hoeffding_tree . . . 42
image_converter . . . 45
kde . . . 46

kernel_pca . . . 49
kfn . . . 51
kmeans . . . 53
knn . . . 56
krann . . . 58
lars . . . 60
lars_train . . . 62
linear_regression . . . 64
linear_regression_train . . . 66
linear_svm . . . 67
lmnn . . . 70
local_coordinate_coding . . . 73
logistic_regression . . . 75
logistic_regression_probabilities . . . 78
logistic_regression_train . . . 79
lsh . . . 81
mean_shift . . . 83
mlpack . . . 84
nbc . . . 89
nca . . . 91
nmf . . . 93
pca . . . 95
perceptron . . . 96
predict.mlpack_lars . . . 98
predict.mlpack_linear_regression . . . 99
predict.mlpack_logistic_regression . . . 100
predict.mlpack_random_forest . . . 101
preprocess_binarize . . . 104
preprocess_describe . . . 105
preprocess_one_hot_encoding . . . 107
preprocess_scale . . . 108
preprocess_split . . . 110
radical . . . 112
random_forest_classify . . . 114
random_forest_probabilities . . . 115
random_forest_train . . . 116
Serialize . . . 118
softmax_regression . . . 118
sparse_coding . . . 120
test_r_binding . . . 122

Index **125**

adaboost *AdaBoost*

Description

An implementation of the AdaBoost.MH (Adaptive Boosting) algorithm for classification. This can be used to train an AdaBoost model on labeled data or use an existing AdaBoost model to predict the classes of new points.

Usage

```
adaboost(
  input_model = NA,
  iterations = 1000,
  labels = NA,
  test = NA,
  tolerance = 1e-10,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE),
  weak_learner = "decision_stump"
)
```

Arguments

input_model	Input AdaBoost model (AdaBoostModel).
iterations	The maximum number of boosting iterations to be run (0 will run until convergence.. Default value "1000" (integer).
labels	Labels for the training set (integer row).
test	Test dataset (numeric matrix).
tolerance	The tolerance for change in values of the weighted error during training. Default value "1e-10" (numeric).
training	Dataset for training AdaBoost (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
weak_learner	The type of weak learner to use: 'decision_stump', or 'perceptron'. Default value "decision_stump" (character).

Details

This program implements the AdaBoost (or Adaptive Boosting) algorithm. The variant of AdaBoost implemented here is AdaBoost.MH. It uses a weak learner, either decision stumps or perceptrons, and over many iterations, creates a strong learner that is a weighted ensemble of weak learners. It runs these iterations until a tolerance value is crossed for change in the value of the weighted training error.

For more information about the algorithm, see the paper "Improved Boosting Algorithms Using Confidence-Rated Predictions", by R.E. Schapire and Y. Singer.

This program allows training of an AdaBoost model, and then application of that model to a test dataset. To train a model, a dataset must be passed with the "training" option. Labels can be given with the "labels" option; if no labels are specified, the labels will be assumed to be the last column of the input dataset. Alternately, an AdaBoost model may be loaded with the "input_model" option.

Once a model is trained or loaded, it may be used to provide class predictions for a given test dataset. A test dataset may be specified with the "test" parameter. The predicted classes for each point in the test dataset are output to the "predictions" output parameter. The AdaBoost model itself is output to the "output_model" output parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output trained AdaBoost model (AdaBoostModel).
`predictions`	Predicted labels for the test set (integer row).
`probabilities`	Predicted class probabilities for each point in the test set (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to run AdaBoost on an input dataset "data" with labels
# "labels"and perceptrons as the weak learner type, storing the trained model
# in "model", one could use the following command:
#
# \dontrun{
# output <- adaboost(training=data, labels=labels, weak_learner="perceptron")
# model <- output$output_model
# }
#
# Similarly, an already-trained model in "model" can be used to provide class
# predictions from test data "test_data" and store the output in
# "predictions" with the following command:
#
# \dontrun{
# output <- adaboost(input_model=model, test=test_data)
# predictions <- output$predictions
# }
```

adaboost_classify *AdaBoost Prediction*

Description

Class predictions from model.

Usage

```
adaboost_classify(
  input_model,
  test,
  verbose = getOption("mlpack.verbose", FALSE)
)

## S3 method for class 'mlpack_adaboost'
predict(object, newdata, type = c("predictions", "probabilities"), ...)
```

Arguments

input_model	Input AdaBoost model (AdaBoostModel).
test	Test dataset (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
object	An instantiated model object for which prediction is desired
newdata	A test data set
type	A character value selection predictions or probabilities
...	Additional optional arguments affecting the prediction

Value

A list with several components defining the class attributes:

predictions	Predicted labels for the test set (integer row).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

adaboost_probabilities *AdaBoost Probability Prediction*

Description

Class probabilities from model.

Usage

```
adaboost_probabilities(
  input_model,
  test,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input_model	Input AdaBoost model (AdaBoostModel).
test	Test dataset (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

probabilities	Predicted class probabilities for each point in the test set (numeric matrix).

Author(s)

mlpack developers

Examples

```
# \dontrun{ prob <- predict(model, newdata=X_test, type="probabilities") }
```

adaboost_train *AdaBoost*

Description

Training AdaBoost model.

Usage

```
adaboost_train(
  training,
  iterations = 1000,
  labels = NA,
  tolerance = 1e-10,
  verbose = getOption("mlpack.verbose", FALSE),
  weak_learner = "decision_stump"
)
```

Arguments

training	Dataset for training AdaBoost (numeric matrix).
iterations	The maximum number of boosting iterations to be run (0 will run until convergence.. Default value "1000" (integer).
labels	Labels for the training set (integer row).
tolerance	The tolerance for change in values of the weighted error during training. Default value "1e-10" (numeric).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
weak_learner	The type of weak learner to use: 'decision_stump', or 'perceptron'. Default value "decision_stump" (character).

Details

This program implements the AdaBoost (or Adaptive Boosting) algorithm. The variant of AdaBoost implemented here is AdaBoost.MH. It uses a weak learner, either decision stumps or perceptrons, and over many iterations, creates a strong learner that is a weighted ensemble of weak learners. It runs these iterations until a tolerance value is crossed for change in the value of the weighted training error.

For more information about the algorithm, see the paper "Improved Boosting Algorithms Using Confidence-Rated Predictions", by R.E. Schapire and Y. Singer.

Value

A list with several components defining the class attributes:

output_model	Output trained AdaBoost model (AdaBoostModel).

Author(s)

mlpack developers

Examples

```
#
# \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <- as.matrix(read.csv("http://datasets.mlpack.org/iris.csv",
# header=FALSE))
# y <- as.matrix(read.csv("http://datasets.mlpack.org/iris_labels.csv",
# header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- adaboost_train(training=X_train, labels=y_train)
# }
```

---

approx_kfn *Approximate furthest neighbor search*

Description

An implementation of two strategies for furthest neighbor search. This can be used to compute the furthest neighbor of query point(s) from a set of points; furthest neighbor models can be saved and reused with future query point(s).

Usage

```
approx_kfn(
  algorithm = "ds",
  calculate_error = FALSE,
  exact_distances = NA,
  input_model = NA,
  k = 0,
  num_projections = 5,
  num_tables = 5,
  query = NA,
  reference = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

algorithm	Algorithm to use: 'ds' or 'qdafn'. Default value "ds" (character).
calculate_error	If set, calculate the average distance error for the first furthest neighbor only. Default value "FALSE" (logical).
exact_distances	Matrix containing exact distances to furthest neighbors; this can be used to avoid explicit calculation when –calculate_error is set (numeric matrix).
input_model	File containing input model (ApproxKFNModel).
k	Number of furthest neighbors to search for. Default value "0" (integer).
num_projections	Number of projections to use in each hash table. Default value "5" (integer).
num_tables	Number of hash tables to use. Default value "5" (integer).
query	Matrix containing query points (numeric matrix).
reference	Matrix containing the reference dataset (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program implements two strategies for furthest neighbor search. These strategies are:

- The 'qdafn' algorithm from "Approximate Furthest Neighbor in High Dimensions" by R. Pagh, F. Silvestri, J. Sivertsen, and M. Skala, in Similarity Search and Applications 2015 (SISAP). - The 'DrusillaSelect' algorithm from "Fast approximate furthest neighbors with data-dependent candidate selection", by R.R. Curtin and A.B. Gardner, in Similarity Search and Applications 2016 (SISAP).

These two strategies give approximate results for the furthest neighbor search problem and can be used as fast replacements for other furthest neighbor techniques such as those found in the mlpack_kfn program. Note that typically, the 'ds' algorithm requires far fewer tables and projections than the 'qdafn' algorithm.

Specify a reference set (set to search in) with "reference", specify a query set with "query", and specify algorithm parameters with "num_tables" and "num_projections" (or don't and defaults will be used). The algorithm to be used (either 'ds'—the default—or 'qdafn') may be specified with "algorithm". Also specify the number of neighbors to search for with "k".

Note that for 'qdafn' in lower dimensions, "num_projections" may need to be set to a high value in order to return results for each query point.

If no query set is specified, the reference set will be used as the query set. The "output_model" output parameter may be used to store the built model, and an input model may be loaded instead of specifying a reference set with the "input_model" option.

Results for each query point can be stored with the "neighbors" and "distances" output parameters. Each row of these output matrices holds the k distances or neighbor indices for each query point.

Value

A list with several components defining the class attributes:

`distances`	Matrix to save furthest neighbor distances to (numeric matrix).
`neighbors`	Matrix to save neighbor indices to (integer matrix).
`output_model`	File to save output model to (ApproxKFNModel).

Author(s)

mlpack developers

Examples

```
# For example, to find the 5 approximate furthest neighbors with
# "reference_set" as the reference set and "query_set" as the query set using
# DrusillaSelect, storing the furthest neighbor indices to "neighbors" and
# the furthest neighbor distances to "distances", one could call
#
# \dontrun{
# output <- approx_kfn(query=query_set, reference=reference_set, k=5,
#   algorithm="ds")
# neighbors <- output$neighbors
# distances <- output$distances
# }
#
# and to perform approximate all-furthest-neighbors search with k=1 on the
# set "data" storing only the furthest neighbor distances to "distances", one
# could call
#
# \dontrun{
# output <- approx_kfn(reference=reference_set, k=1)
# distances <- output$distances
# }
#
# A trained model can be re-used.  If a model has been previously saved to
# "model", then we may find 3 approximate furthest neighbors on a query set
# "new_query_set" using that model and store the furthest neighbor indices
# into "neighbors" by calling
#
# \dontrun{
# output <- approx_kfn(input_model=model, query=new_query_set, k=3)
# neighbors <- output$neighbors
# }
```

`bayesian_linear_regression` *BayesianLinearRegression*

Description

An implementation of the Bayesian linear regression.

Usage

```
bayesian_linear_regression(
  center = FALSE,
  input = NA,
  input_model = NA,
  responses = NA,
  scale = FALSE,
  test = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)

## S3 method for class 'mlpack_bayesian_linear_regression'
predict(object, newdata, stddevs = FALSE, ...)
```

Arguments

center	Center the data and fit the intercept if enabled. Default value "FALSE" (logical).
input	Matrix of covariates (X) (numeric matrix).
input_model	Trained BayesianLinearRegression model to use (BayesianLinearRegression).
responses	Matrix of responses/observations (y) (numeric row).
scale	Scale each feature by their standard deviations if enabled. Default value "FALSE" (logical).
test	Matrix containing points to regress on (test points) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
object	An instantiated model object for which prediction is desired
newdata	A test data set
stddevs	A flag selecting standard deviation estimation returned along with point estimates
...	Additional optional arguments affecting the prediction

Details

An implementation of the Bayesian linear regression. This model is a probabilistic view and implementation of the linear regression. The final solution is obtained by computing a posterior distribution from gaussian likelihood and a zero mean gaussian isotropic prior distribution on the solution. Optimization is AUTOMATIC and does not require cross validation. The optimization is performed by maximization of the evidence function. Parameters are tuned during the maximization of the marginal likelihood. This procedure includes the Ockham's razor that penalizes over complex solutions.

This program is able to train a Bayesian linear regression model or load a model from file, output regression predictions for a test set, and save the trained model to a file.

To train a BayesianLinearRegression model, the "input" and "responses"parameters must be given. The "center"and "scale" parameters control the centering and the normalizing options. A trained model can be saved with the "output_model". If no training is desired at all, a model can be passed via the "input_model" parameter.

The program can also provide predictions for test data using either the trained model or the given input model. Test points can be specified with the "test" parameter. Predicted responses to the test points can be saved with the "predictions" output parameter. The corresponding standard deviation can be save by precising the "stds" parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output BayesianLinearRegression model (BayesianLinearRegression).
`predictions`	If –test_file is specified, this file is where the predicted responses will be saved (numeric matrix).
`stds`	If specified, this is where the standard deviations of the predictive distribution will be saved (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, the following command trains a model on the data "data" and
# responses "responses"with center set to true and scale set to false (so,
# Bayesian linear regression is being solved, and then the model is saved to
# "blr_model":
#
# \dontrun{
# output <- bayesian_linear_regression(input=data, responses=responses,
#   center=1, scale=0)
# blr_model <- output$output_model
# }
#
# The following command uses the "blr_model" to provide predicted  responses
# for the data "test" and save those  responses to "test_predictions":
#
# \dontrun{
# output <- bayesian_linear_regression(input_model=blr_model, test=test)
# test_predictions <- output$predictions
# }
#
# Because the estimator computes a predictive distribution instead of a
# simple point estimate, the "stds" parameter allows one to save the
# prediction uncertainties:
#
```

```
# \dontrun{
# output <- bayesian_linear_regression(input_model=blr_model, test=test)
# test_predictions <- output$predictions
# stds <- output$stds
# }
```

`bayesian_linear_regression_predict` *BayesianLinearRegression Prediction*

Description

An implementation of the Bayesian linear regression prediction: Given a pre-trained model and a test data set, it provides model predictions.

Usage

```
bayesian_linear_regression_predict(
  input_model,
  test,
  stddevs = FALSE,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Trained BayesianLinearRegression model to use (BayesianLinearRegression).
`test`	Matrix containing points to regress on (test points) (numeric matrix).
`stddevs`	Return standard deviations along with predictions. Default value "FALSE" (logical).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

`predictions`	Matrix of predicted responses, with associated standard deviations if option selected (numeric matrix).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

`bayesian_linear_regression_train` *BayesianLinearRegression Training*

Description

An implementation of the Bayesian linear regression training.

Usage

```
bayesian_linear_regression_train(
  input,
  responses,
  center = FALSE,
  scale = FALSE,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Matrix of covariates (X) (numeric matrix).
`responses`	Matrix of responses/observations (y) (numeric row).
`center`	Center the data and fit the intercept if enabled. Default value "FALSE" (logical).
`scale`	Scale each feature by their standard deviations if enabled. Default value "FALSE" (logical).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of the Bayesian linear regression. This model is a probabilistic view and implementation of the linear regression. The final solution is obtained by computing a posterior distribution from gaussian likelihood and a zero mean gaussian isotropic prior distribution on the solution. Optimization is AUTOMATIC and does not require cross validation. The optimization is performed by maximization of the evidence function. Parameters are tuned during the maximization of the marginal likelihood. This procedure includes the Ockham's razor that penalizes over complex solutions.

To train a BayesianLinearRegression model, the "input" and "responses" parameters must be given. The "center" and "scale" parameters control the centering and the normalizing options. A trained model is returned.

Value

A list with several components defining the class attributes:

`output_model`	Output BayesianLinearRegression model (BayesianLinearRegression).

Author(s)

mlpack developers

Examples

```
#
# #' # \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <- as.matrix(read.csv("http://datasets.mlpack.org/admission_predict.csv",
# header=FALSE))
# y <-
# as.matrix(read.csv("http://datasets.mlpack.org/admission_predict.responses.
# csv", header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- bayesian_linear_regression_train(input=X_train, responses=y_train,
#   center=1, scale=0)
#   }
```

---

cf *Collaborative Filtering*

Description

An implementation of several collaborative filtering (CF) techniques for recommender systems. This can be used to train a new CF model, or use an existing CF model to compute recommendations.

Usage

```
cf(
  algorithm = "NMF",
  all_user_recommendations = FALSE,
  input_model = NA,
  interpolation = "average",
  iteration_only_termination = FALSE,
  max_iterations = 1000,
  min_residue = 1e-05,
  neighbor_search = "euclidean",
  neighborhood = 5,
  normalization = "none",
  query = NA,
```

```
  rank = 0,
  recommendations = 5,
  seed = 0,
  test = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

algorithm	Algorithm used for matrix factorization. Default value "NMF" (character).
all_user_recommendations	Generate recommendations for all users. Default value "FALSE" (logical).
input_model	Trained CF model to load (CFModel).
interpolation	Algorithm used for weight interpolation. Default value "average" (character).
iteration_only_termination	Terminate only when the maximum number of iterations is reached. Default value "FALSE" (logical).
max_iterations	Maximum number of iterations. If set to zero, there is no limit on the number of iterations. Default value "1000" (integer).
min_residue	Residue required to terminate the factorization (lower values generally mean better fits). Default value "1e-05" (numeric).
neighbor_search	Algorithm used for neighbor search. Default value "euclidean" (character).
neighborhood	Size of the neighborhood of similar users to consider for each query user. Default value "5" (integer).
normalization	Normalization performed on the ratings. Default value "none" (character).
query	List of query users for which recommendations should be generated (integer matrix).
rank	Rank of decomposed matrices (if 0, a heuristic is used to estimate the rank). Default value "0" (integer).
recommendations	Number of recommendations to generate for each query user. Default value "5" (integer).
seed	Set the random seed (0 uses std::time(NULL)). Default value "0" (integer).
test	Test set to calculate RMSE on (numeric matrix).
training	Input dataset to perform CF on (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs collaborative filtering (CF) on the given dataset. Given a list of user, item and preferences (the "training" parameter), the program will perform a matrix decomposition and then can perform a series of actions related to collaborative filtering. Alternately, the program can load an existing saved CF model with the "input_model" parameter and then use that model to provide recommendations or predict values.

The input matrix should be a 3-dimensional matrix of ratings, where the first dimension is the user, the second dimension is the item, and the third dimension is that user's rating of that item. Both the users and items should be numeric indices, not names. The indices are assumed to start from 0.

A set of query users for which recommendations can be generated may be specified with the "query" parameter; alternately, recommendations may be generated for every user in the dataset by specifying the "all_user_recommendations" parameter. In addition, the number of recommendations per user to generate can be specified with the "recommendations" parameter, and the number of similar users (the size of the neighborhood) to be considered when generating recommendations can be specified with the "neighborhood" parameter.

For performing the matrix decomposition, the following optimization algorithms can be specified via the "algorithm" parameter:

- 'RegSVD' – Regularized SVD using a SGD optimizer - 'NMF' – Non-negative matrix factorization with alternating least squares update rules - 'BatchSVD' – SVD batch learning - 'SVDIncompleteIncremental' – SVD incomplete incremental learning - 'SVDCompleteIncremental' – SVD complete incremental learning - 'BiasSVD' – Bias SVD using a SGD optimizer - 'SVDPP' – SVD++ using a SGD optimizer - 'RandSVD' – RandomizedSVD learning - 'QSVD' – QuicSVD learning - 'BKSVD' – Block Krylov SVD learning

The following neighbor search algorithms can be specified via the "neighbor_search" parameter:

- 'cosine' – Cosine Search Algorithm - 'euclidean' – Euclidean Search Algorithm - 'pearson' – Pearson Search Algorithm

The following weight interpolation algorithms can be specified via the "interpolation" parameter:

- 'average' – Average Interpolation Algorithm - 'regression' – Regression Interpolation Algorithm - 'similarity' – Similarity Interpolation Algorithm

The following ranking normalization algorithms can be specified via the "normalization" parameter:

- 'none' – No Normalization - 'item_mean' – Item Mean Normalization - 'overall_mean' – Overall Mean Normalization - 'user_mean' – User Mean Normalization - 'z_score' – Z-Score Normalization

A trained model may be saved to with the "output_model" output parameter.

Value

A list with several components defining the class attributes:

`output`	Matrix that will store output recommendations (integer matrix).
`output_model`	Output for trained CF model (CFModel).

Author(s)

mlpack developers

Examples

```
# To train a CF model on a dataset "training_set" using NMF for decomposition
# and saving the trained model to "model", one could call:
#
# \dontrun{
# output <- cf(training=training_set, algorithm="NMF")
# model <- output$output_model
# }
#
# Then, to use this model to generate recommendations for the list of users
# in the query set "users", storing 5 recommendations in "recommendations",
# one could call
#
# \dontrun{
# output <- cf(input_model=model, query=users, recommendations=5)
# recommendations <- output$output
# }
```

`dbscan` *DBSCAN clustering*

Description

An implementation of DBSCAN clustering. Given a dataset, this can compute and return a clustering of that dataset.

Usage

```
dbscan(
  input,
  epsilon = 1,
  min_size = 5,
  naive = FALSE,
  selection_type = "ordered",
  single_mode = FALSE,
  tree_type = "kd",
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Input dataset to cluster (numeric matrix).
`epsilon`	Radius of each range search. Default value "1" (numeric).
`min_size`	Minimum number of points for a cluster. Default value "5" (integer).
`naive`	If set, brute-force range search (not tree-based) will be used. Default value "FALSE" (logical).

`selection_type`	If using point selection policy, the type of selection to use ('ordered', 'random'). Default value "ordered" (character).
`single_mode`	If set, single-tree range search (not dual-tree) will be used. Default value "FALSE" (logical).
`tree_type`	If using single-tree or dual-tree search, the type of tree to use ('kd', 'r', 'r-star', 'x', 'hilbert-r', 'r-plus', 'r-plus-plus', 'cover', 'ball'). Default value "kd" (character).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program implements the DBSCAN algorithm for clustering using accelerated tree-based range search. The type of tree that is used may be parameterized, or brute-force range search may also be used.

The input dataset to be clustered may be specified with the "input" parameter; the radius of each range search may be specified with the "epsilon" parameters, and the minimum number of points in a cluster may be specified with the "min_size" parameter.

The "assignments" and "centroids" output parameters may be used to save the output of the clustering. "assignments" contains the cluster assignments of each point, and "centroids" contains the centroids of each cluster.

The range search may be controlled with the "tree_type", "single_mode", and "naive" parameters. "tree_type" can control the type of tree used for range search; this can take a variety of values: 'kd', 'r', 'r-star', 'x', 'hilbert-r', 'r-plus', 'r-plus-plus', 'cover', 'ball'. The "single_mode" parameter will force single-tree search (as opposed to the default dual-tree search), and '"naive" will force brute-force range search.

Value

A list with several components defining the class attributes:

`assignments`	Output matrix for assignments of each point (integer row).
`centroids`	Matrix to save output centroids to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# An example usage to run DBSCAN on the dataset in "input" with a radius of
# 0.5 and a minimum cluster size of 5 is given below:
#
# \dontrun{
# dbscan(input=input, epsilon=0.5, min_size=5)
# }
```

`decision_tree` *Decision tree*

Description

An implementation of an ID3-style decision tree for classification, which supports categorical data. Given labeled data with numeric or categorical features, a decision tree can be trained and saved; or, an existing decision tree can be used for classification on new points.

Usage

```
decision_tree(
  input_model = NA,
  labels = NA,
  maximum_depth = 0,
  minimum_gain_split = 1e-07,
  minimum_leaf_size = 20,
  print_training_accuracy = FALSE,
  test = NA,
  test_labels = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE),
  weights = NA
)
```

Arguments

`input_model`	Pre-trained decision tree, to be used with test points (DecisionTreeModel).
`labels`	Training labels (integer row).
`maximum_depth`	Maximum depth of the tree (0 means no limit). Default value "0" (integer).
`minimum_gain_split`	Minimum gain for node splitting. Default value "1e-07" (numeric).
`minimum_leaf_size`	Minimum number of points in a leaf. Default value "20" (integer).
`print_training_accuracy`	Print the training accuracy. Default value "FALSE" (logical).
`test`	Testing dataset (may be categorical) (numeric matrix/data.frame with info).
`test_labels`	Test point labels, if accuracy calculation is desired (integer row).
`training`	Training dataset (may be categorical) (numeric matrix/data.frame with info).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
`weights`	The weight of label (numeric matrix).

Details

Train and evaluate using a decision tree. Given a dataset containing numeric or categorical features, and associated labels for each point in the dataset, this program can train a decision tree on that data.

The training set and associated labels are specified with the "training" and "labels" parameters, respectively. The labels should be in the range ‘[0, num_classes - 1]‘. Optionally, if "labels" is not specified, the labels are assumed to be the last dimension of the training dataset.

When a model is trained, the "output_model" output parameter may be used to save the trained model. A model may be loaded for predictions with the "input_model" parameter. The "input_model" parameter may not be specified when the "training" parameter is specified. The "minimum_leaf_size" parameter specifies the minimum number of training points that must fall into each leaf for it to be split. The "minimum_gain_split" parameter specifies the minimum gain that is needed for the node to split. The "maximum_depth" parameter specifies the maximum depth of the tree. If "print_training_accuracy" is specified, the training accuracy will be printed.

Test data may be specified with the "test" parameter, and if performance numbers are desired for that test set, labels may be specified with the "test_labels" parameter. Predictions for each test point may be saved via the "predictions" output parameter. Class probabilities for each prediction may be saved with the "probabilities" output parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output for trained decision tree (DecisionTreeModel).
`predictions`	Class predictions for each test point (integer row).
`probabilities`	Class probabilities for each test point (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to train a decision tree with a minimum leaf size of 20 on the
# dataset contained in "data" with labels "labels", saving the output model
# to "tree" and printing the training error, one could call
#
# \dontrun{
# output <- decision_tree(training=data, labels=labels, minimum_leaf_size=20,
#   minimum_gain_split=0.001, print_training_accuracy=TRUE)
# tree <- output$output_model
# }
#
# Then, to use that model to classify points in "test_set" and print the test
# error given the labels "test_labels" using that model, while saving the
# predictions for each point to "predictions", one could call
#
# \dontrun{
# output <- decision_tree(input_model=tree, test=test_set,
```

```
#   test_labels=test_labels)
# predictions <- output$predictions
# }
```

`decision_tree_classify` *Decision tree Prediction*

Description

Class predictions from train decision tree model.

Usage

```
decision_tree_classify(
  input_model,
  test,
  test_labels = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Pre-trained decision tree, to be used with test points (DecisionTreeModel).
`test`	Testing dataset (may contain categorical variables) (numeric matrix/data.frame with info).
`test_labels`	Test point labels, if accuracy calculation is desired (integer row).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

`predictions`	Class predictions for each test point (integer row).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

`decision_tree_probabilities` *Decision tree Prediction*

Description

Class predictions from train decision tree model.

Usage

```
decision_tree_probabilities(
  input_model,
  test,
  test_labels = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Pre-trained decision tree, to be used with test points (DecisionTreeModel).
`test`	Testing dataset (may contain categorical variables) (numeric matrix/data.frame with info).
`test_labels`	Test point labels, if accuracy calculation is desired (integer row).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

`probabilities`	Class probabilities for each test point if probabilities has been selected (numeric matrix).

Author(s)

mlpack developers

Examples

```
# \dontrun{ prob <- predict(model, newdata=X_test, type="probabilities") }
```

`decision_tree_train` *Decision tree training*

Description

Training ID3-style decision tree model.

Usage

```
decision_tree_train(
  training,
  labels = NA,
  maximum_depth = 0,
  minimum_gain_split = 1e-07,
  minimum_leaf_size = 20,
  print_training_accuracy = FALSE,
  verbose = getOption("mlpack.verbose", FALSE),
  weights = NA
)
```

Arguments

`training`	Training dataset (may contain categorical variables) (numeric matrix/data.frame with info).
`labels`	Training labels (integer row).
`maximum_depth`	Maximum depth of the tree (0 means no limit). Default value "0" (integer).
`minimum_gain_split`	Minimum gain for node splitting. Default value "1e-07" (numeric).
`minimum_leaf_size`	Minimum number of points in a leaf. Default value "20" (integer).
`print_training_accuracy`	Print the training accuracy. Default value "FALSE" (logical).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
`weights`	The weight of label (numeric matrix).

Details

Train using a decision tree. Given a dataset containing numeric or categorical features, and associated labels for each point in the dataset, this program can train a decision tree on that data.

The training set and associated labels are specified with the "training" and "labels" parameters, respectively. The labels should be in the range ‘[0, num_classes - 1]‘. Optionally, if "labels" is not specified, the labels are assumed to be the last dimension of the training dataset.

The trained model is returned, and can then be used for prediction. The "minimum_leaf_size" parameter specifies the minimum number of training points that must fall into each leaf for it to be split. The "minimum_gain_split" parameter specifies the minimum gain that is needed for the node to split. The "maximum_depth" parameter specifies the maximum depth of the tree. If "print_training_accuracy" is specified, the training accuracy will be printed.

Value

A list with several components defining the class attributes:

`output_model` Output for trained decision tree (DecisionTreeModel).

Author(s)

mlpack developers

Examples

```
#
# #' # \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <- as.matrix(read.csv("http://datasets.mlpack.org/iris.csv",
# header=FALSE))
# y <- as.matrix(read.csv("http://datasets.mlpack.org/iris_labels.csv",
# header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- decision_tree_train(training=X_train, labels=y_train,
#   minimum_leaf_size=20, minimum_gain_split=0.001)
#   }
```

---

det *Density Estimation With Density Estimation Trees*

Description

An implementation of density estimation trees for the density estimation task. Density estimation trees can be trained or used to predict the density at locations given by query points.

Usage

```
det(
  folds = 10,
  input_model = NA,
  max_leaf_size = 10,
  min_leaf_size = 5,
  path_format = "lr",
  skip_pruning = FALSE,
  test = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

folds	The number of folds of cross-validation to perform for the estimation (0 is LOOCV. Default value "10" (integer).
input_model	Trained density estimation tree to load (DTree).
max_leaf_size	The maximum size of a leaf in the unpruned, fully grown DET. Default value "10" (integer).
min_leaf_size	The minimum size of a leaf in the unpruned, fully grown DET. Default value "5" (integer).
path_format	The format of path printing: 'lr', 'id-lr', or 'lr-id'. Default value "lr" (character).
skip_pruning	Whether to bypass the pruning process and output the unpruned tree only. Default value "FALSE" (logical).
test	A set of test points to estimate the density of (numeric matrix).
training	The data set on which to build a density estimation tree (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs a number of functions related to Density Estimation Trees. The optimal Density Estimation Tree (DET) can be trained on a set of data (specified by "training") using cross-validation (with number of folds specified with the "folds" parameter). This trained density estimation tree may then be saved with the "output_model" output parameter.

The variable importances (that is, the feature importance values for each dimension) may be saved with the "vi" output parameter, and the density estimates for each training point may be saved with the "training_set_estimates" output parameter.

Enabling path printing for each node outputs the path from the root node to a leaf for each entry in the test set, or training set (if a test set is not provided). Strings like 'LRLRLR' (indicating that traversal went to the left child, then the right child, then the left child, and so forth) will be output. If 'lr-id' or 'id-lr' are given as the "path_format" parameter, then the ID (tag) of every node along the path will be printed after or before the L or R character indicating the direction of traversal, respectively.

This program also can provide density estimates for a set of test points, specified in the "test" parameter. The density estimation tree used for this task will be the tree that was trained on the given training points, or a tree given as the parameter "input_model". The density estimates for the test points may be saved using the "test_set_estimates" output parameter.

Value

A list with several components defining the class attributes:

`output_model` Output to save trained density estimation tree to (DTree).

`tag_counters_file`
: The file to output the number of points that went to each leaf. Default value "" (character).

`tag_file` The file to output the tags (and possibly paths) for each sample in the test set. Default value "" (character).

`test_set_estimates`
: The output estimates on the test set from the final optimally pruned tree (numeric matrix).

`training_set_estimates`
: The output density estimates on the training set from the final optimally pruned tree (numeric matrix).

`vi` The output variable importance values for each feature (numeric matrix).

Author(s)

mlpack developers

---

emst *Fast Euclidean Minimum Spanning Tree*

Description

An implementation of the Dual-Tree Boruvka algorithm for computing the Euclidean minimum spanning tree of a set of input points.

Usage

```
emst(
  input,
  leaf_size = 1,
  naive = FALSE,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Input data matrix (numeric matrix).
leaf_size	Leaf size in the kd-tree. One-element leaves give the empirically best performance, but at the cost of greater memory requirements. Default value "1" (integer).
naive	Compute the MST using O(n^2) naive algorithm. Default value "FALSE" (logical).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program can compute the Euclidean minimum spanning tree of a set of input points using the dual-tree Boruvka algorithm.

The set to calculate the minimum spanning tree of is specified with the "input" parameter, and the output may be saved with the "output" output parameter.

The "leaf_size" parameter controls the leaf size of the kd-tree that is used to calculate the minimum spanning tree, and if the "naive" option is given, then brute-force search is used (this is typically much slower in low dimensions). The leaf size does not affect the results, but it may have some effect on the runtime of the algorithm.

Value

A list with several components defining the class attributes:

output	Output data. Stored as an edge list (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, the minimum spanning tree of the input dataset "data" can be
# calculated with a leaf size of 20 and stored as "spanning_tree" using the
# following command:
#
# \dontrun{
# spanning_tree <- emst(input=data, leaf_size=20)
# }
# #' #
# The output matrix is a three-dimensional matrix, where each row indicates
# an edge.  The first dimension corresponds to the lesser index of the edge;
# the second dimension corresponds to the greater index of the edge; and the
# third column corresponds to the distance between the two points.
```

`fastmks` *FastMKS (Fast Max-Kernel Search)*

Description

An implementation of the single-tree and dual-tree fast max-kernel search (FastMKS) algorithm. Given a set of reference points and a set of query points, this can find the reference point with maximum kernel value for each query point; trained models can be reused for future queries.

Usage

```
fastmks(
  bandwidth = 1,
  base = 2,
  degree = 2,
  input_model = NA,
  k = 0,
  kernel = "linear",
  naive = FALSE,
  offset = 0,
  query = NA,
  reference = NA,
  scale = 1,
  single = FALSE,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`bandwidth`	Bandwidth (for Gaussian, Epanechnikov, and triangular kernels). Default value "1" (numeric).
`base`	Base to use during cover tree construction. Default value "2" (numeric).
`degree`	Degree of polynomial kernel. Default value "2" (numeric).
`input_model`	Input FastMKS model to use (FastMKSModel).
`k`	Number of maximum kernels to find. Default value "0" (integer).
`kernel`	Kernel type to use: 'linear', 'polynomial', 'cosine', 'gaussian', 'epanechnikov', 'triangular', 'hyptan'. Default value "linear" (character).
`naive`	If true, O(n^2) naive mode is used for computation. Default value "FALSE" (logical).
`offset`	Offset of kernel (for polynomial and hyptan kernels). Default value "0" (numeric).
`query`	The query dataset (numeric matrix).
`reference`	The reference dataset (numeric matrix).
`scale`	Scale of kernel (for hyptan kernel). Default value "1" (numeric).

single	If true, single-tree search is used (as opposed to dual-tree search. Default value "FALSE" (logical).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program will find the k maximum kernels of a set of points, using a query set and a reference set (which can optionally be the same set). More specifically, for each point in the query set, the k points in the reference set with maximum kernel evaluations are found. The kernel function used is specified with the "kernel" parameter.

Value

A list with several components defining the class attributes:

indices	Output matrix of indices (integer matrix).
kernels	Output matrix of kernels (numeric matrix).
output_model	Output for FastMKS model (FastMKSModel).

Author(s)

mlpack developers

Examples

```
# For example, the following command will calculate, for each point in the
# query set "query", the five points in the reference set "reference" with
# maximum kernel evaluation using the linear kernel.  The kernel evaluations
# may be saved with the  "kernels" output parameter and the indices may be
# saved with the "indices" output parameter.
#
# \dontrun{
# output <- fastmks(k=5, reference=reference, query=query, kernel="linear")
# indices <- output$indices
# kernels <- output$kernels
# }
# #' #
# The output matrices are organized such that row i and column j in the
# indices matrix corresponds to the index of the point in the reference set
# that has j'th largest kernel evaluation with the point in the query set
# with index i.  Row i and column j in the kernels matrix corresponds to the
# kernel evaluation between those two points.
#
# This program performs FastMKS using a cover tree.  The base used to build
# the cover tree can be specified with the "base" parameter.
```

`gmm_generate` *GMM Sample Generator*

Description

A sample generator for pre-trained GMMs. Given a pre-trained GMM, this can sample new points randomly from that distribution.

Usage

```
gmm_generate(
  input_model,
  samples,
  seed = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Input GMM model to generate samples from (GMM).
`samples`	Number of samples to generate (integer).
`seed`	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program is able to generate samples from a pre-trained GMM (use gmm_train to train a GMM). The pre-trained GMM must be specified with the "input_model" parameter. The number of samples to generate is specified by the "samples" parameter. Output samples may be saved with the "output" output parameter.

Value

A list with several components defining the class attributes:

`output`	Matrix to save output samples in (numeric matrix).

Author(s)

mlpack developers

Examples

```
# The following command can be used to generate 100 samples from the
# pre-trained GMM "gmm" and store those generated samples in "samples":
#
# \dontrun{
# samples <- gmm_generate(input_model=gmm, samples=100)
# }
```

---

gmm_probability *GMM Probability Calculator*

Description

A probability calculator for GMMs. Given a pre-trained GMM and a set of points, this can compute the probability that each point is from the given GMM.

Usage

```
gmm_probability(
  input,
  input_model,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Input matrix to calculate probabilities of (numeric matrix).
input_model	Input GMM to use as model (GMM).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program calculates the probability that given points came from a given GMM (that is, P(X | gmm)). The GMM is specified with the "input_model" parameter, and the points are specified with the "input" parameter. The output probabilities may be saved via the "output" output parameter.

Value

A list with several components defining the class attributes:

output	Matrix to store calculated probabilities in (numeric matrix).

Author(s)

mlpack developers

Examples

```
# So, for example, to calculate the probabilities of each point in "points"
# coming from the pre-trained GMM "gmm", while storing those probabilities in
# "probs", the following command could be used:
#
# \dontrun{
# probs <- gmm_probability(input_model=gmm, input=points)
# }
```

`gmm_train` *Gaussian Mixture Model (GMM) Training*

Description

An implementation of the EM algorithm for training Gaussian mixture models (GMMs). Given a dataset, this can train a GMM for future use with other tools.

Usage

```
gmm_train(
  gaussians,
  input,
  diagonal_covariance = FALSE,
  input_model = NA,
  kmeans_max_iterations = 1000,
  max_iterations = 250,
  no_force_positive = FALSE,
  noise = 0,
  percentage = 0.02,
  refined_start = FALSE,
  samplings = 100,
  seed = 0,
  tolerance = 1e-10,
  trials = 1,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`gaussians`	Number of Gaussians in the GMM (integer).
`input`	The training data on which the model will be fit (numeric matrix).
`diagonal_covariance`	Force the covariance of the Gaussians to be diagonal. This can accelerate training time significantly. Default value "FALSE" (logical).
`input_model`	Initial input GMM model to start training with (GMM).

`kmeans_max_iterations`
: Maximum number of iterations for the k-means algorithm (used to initialize EM). Default value "1000" (integer).

`max_iterations`
: Maximum number of iterations of EM algorithm (passing 0 will run until convergence). Default value "250" (integer).

`no_force_positive`
: Do not force the covariance matrices to be positive definite. Default value "FALSE" (logical).

`noise`
: Variance of zero-mean Gaussian noise to add to data. Default value "0" (numeric).

`percentage`
: If using –refined_start, specify the percentage of the dataset used for each sampling (should be between 0.0 and 1.0). Default value "0.02" (numeric).

`refined_start`
: During the initialization, use refined initial positions for k-means clustering (Bradley and Fayyad, 1998). Default value "FALSE" (logical).

`samplings`
: If using –refined_start, specify the number of samplings used for initial points. Default value "100" (integer).

`seed`
: Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).

`tolerance`
: Tolerance for convergence of EM. Default value "1e-10" (numeric).

`trials`
: Number of trials to perform in training GMM. Default value "1" (integer).

`verbose`
: Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program takes a parametric estimate of a Gaussian mixture model (GMM) using the EM algorithm to find the maximum likelihood estimate. The model may be saved and reused by other mlpack GMM tools.

The input data to train on must be specified with the "input" parameter, and the number of Gaussians in the model must be specified with the "gaussians" parameter. Optionally, many trials with different random initializations may be run, and the result with highest log-likelihood on the training data will be taken. The number of trials to run is specified with the "trials" parameter. By default, only one trial is run.

The tolerance for convergence and maximum number of iterations of the EM algorithm are specified with the "tolerance" and "max_iterations" parameters, respectively. The GMM may be initialized for training with another model, specified with the "input_model" parameter. Otherwise, the model is initialized by running k-means on the data. The k-means clustering initialization can be controlled with the "kmeans_max_iterations", "refined_start", "samplings", and "percentage" parameters. If "refined_start" is specified, then the Bradley-Fayyad refined start initialization will be used. This can often lead to better clustering results.

The 'diagonal_covariance' flag will cause the learned covariances to be diagonal matrices. This significantly simplifies the model itself and causes training to be faster, but restricts the ability to fit more complex GMMs.

If GMM training fails with an error indicating that a covariance matrix could not be inverted, make sure that the "no_force_positive" parameter is not specified. Alternately, adding a small amount

of Gaussian noise (using the "noise" parameter) to the entire dataset may help prevent Gaussians with zero variance in a particular dimension, which is usually the cause of non-invertible covariance matrices.

The "no_force_positive" parameter, if set, will avoid the checks after each iteration of the EM algorithm which ensure that the covariance matrices are positive definite. Specifying the flag can cause faster runtime, but may also cause non-positive definite covariance matrices, which will cause the program to crash.

Value

A list with several components defining the class attributes:

`output_model` Output for trained GMM model (GMM).

Author(s)

mlpack developers

Examples

```
# As an example, to train a 6-Gaussian GMM on the data in "data" with a
# maximum of 100 iterations of EM and 3 trials, saving the trained GMM to
# "gmm", the following command can be used:
#
# \dontrun{
# gmm <- gmm_train(input=data, gaussians=6, trials=3)
# }
#
# To re-train that GMM on another set of data "data2", the following command
# may be used:
#
# \dontrun{
# new_gmm <- gmm_train(input_model=gmm, input=data2, gaussians=6)
# }
```

`hmm_generate` *Hidden Markov Model (HMM) Sequence Generator*

Description

A utility to generate random sequences from a pre-trained Hidden Markov Model (HMM). The length of the desired sequence can be specified, and a random sequence of observations is returned.

Usage

```
hmm_generate(
  length,
  model,
  seed = 0,
  start_state = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

length	Length of sequence to generate (integer).
model	Trained HMM to generate sequences with (HMMModel).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
start_state	Starting state of sequence. Default value "0" (integer).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes an already-trained HMM, specified as the "model" parameter, and generates a random observation sequence and hidden state sequence based on its parameters. The observation sequence may be saved with the "output" output parameter, and the internal state sequence may be saved with the "state" output parameter.

The state to start the sequence in may be specified with the "start_state" parameter.

Value

A list with several components defining the class attributes:

output	Matrix to save observation sequence to (numeric matrix).
state	Matrix to save hidden state sequence to (integer matrix).

Author(s)

mlpack developers

Examples

```
# For example, to generate a sequence of length 150 from the HMM "hmm" and
# save the observation sequence to "observations" and the hidden state
# sequence to "states", the following command may be used:
#
# \dontrun{
# output <- hmm_generate(model=hmm, length=150)
# observations <- output$output
# states <- output$state
# }
```

hmm_loglik *Hidden Markov Model (HMM) Sequence Log-Likelihood*

Description

A utility for computing the log-likelihood of a sequence for Hidden Markov Models (HMMs). Given a pre-trained HMM and an observation sequence, this computes and returns the log-likelihood of that sequence being observed from that HMM.

Usage

```
hmm_loglik(input, input_model, verbose = getOption("mlpack.verbose", FALSE))
```

Arguments

input	File containing observations (numeric matrix).
input_model	File containing HMM (HMMModel).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes an already-trained HMM, specified with the "input_model" parameter, and evaluates the log-likelihood of a sequence of observations, given with the "input" parameter. The computed log-likelihood is given as output.

Value

A list with several components defining the class attributes:

log_likelihood	Log-likelihood of the sequence. Default value "0" (numeric).

Author(s)

mlpack developers

Examples

```
# For example, to compute the log-likelihood of the sequence "seq" with the
# pre-trained HMM "hmm", the following command may be used:
#
# \dontrun{
# hmm_loglik(input=seq, input_model=hmm)
# }
```

`hmm_train` *Hidden Markov Model (HMM) Training*

Description

An implementation of training algorithms for Hidden Markov Models (HMMs). Given labeled or unlabeled data, an HMM can be trained for further use with other mlpack HMM tools.

Usage

```
hmm_train(
  input_file,
  batch = FALSE,
  gaussians = 0,
  input_model = NA,
  labels_file = "",
  seed = 0,
  states = 0,
  tolerance = 1e-05,
  type = "gaussian",
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_file`	File containing input observations (character).
`batch`	If true, input_file (and if passed, labels_file) are expected to contain a list of files to use as input observation sequences (and label sequences). Default value "FALSE" (logical).
`gaussians`	Number of gaussians in each GMM (necessary when type is 'gmm'). Default value "0" (integer).
`input_model`	Pre-existing HMM model to initialize training with (HMMModel).
`labels_file`	Optional file of hidden states, used for labeled training. Default value "" (character).
`seed`	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
`states`	Number of hidden states in HMM (necessary, unless model_file is specified). Default value "0" (integer).
`tolerance`	Tolerance of the Baum-Welch algorithm. Default value "1e-05" (numeric).
`type`	Type of HMM: discrete | gaussian | diag_gmm | gmm. Default value "gaussian" (character).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program allows a Hidden Markov Model to be trained on labeled or unlabeled data. It supports four types of HMMs: Discrete HMMs, Gaussian HMMs, GMM HMMs, or Diagonal GMM HMMs

Either one input sequence can be specified (with "input_file"), or, a file containing files in which input sequences can be found (when "input_file"and"batch" are used together). In addition, labels can be provided in the file specified by "labels_file", and if "batch" is used, the file given to "labels_file" should contain a list of files of labels corresponding to the sequences in the file given to "input_file".

The HMM is trained with the Baum-Welch algorithm if no labels are provided. The tolerance of the Baum-Welch algorithm can be set with the "tolerance"option. By default, the transition matrix is randomly initialized and the emission distributions are initialized to fit the extent of the data.

Optionally, a pre-created HMM model can be used as a guess for the transition matrix and emission probabilities; this is specifiable with "output_model".

Value

A list with several components defining the class attributes:

`output_model` Output for trained HMM (HMMModel).

Author(s)

mlpack developers

`hmm_viterbi` *Hidden Markov Model (HMM) Viterbi State Prediction*

Description

A utility for computing the most probable hidden state sequence for Hidden Markov Models (HMMs). Given a pre-trained HMM and an observed sequence, this uses the Viterbi algorithm to compute and return the most probable hidden state sequence.

Usage

```
hmm_viterbi(input, input_model, verbose = getOption("mlpack.verbose", FALSE))
```

Arguments

`input`	Matrix containing observations (numeric matrix).
`input_model`	Trained HMM to use (HMMModel).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes an already-trained HMM, specified as "input_model", and evaluates the most probable hidden state sequence of a given sequence of observations (specified as '"input", using the Viterbi algorithm. The computed state sequence may be saved using the "output" output parameter.

Value

A list with several components defining the class attributes:

`output`	File to save predicted state sequence to (integer matrix).

Author(s)

mlpack developers

Examples

```
# For example, to predict the state sequence of the observations "obs" using
# the HMM "hmm", storing the predicted state sequence to "states", the
# following command could be used:
#
# \dontrun{
# states <- hmm_viterbi(input=obs, input_model=hmm)
# }
```

---

hoeffding_tree *Hoeffding trees*

Description

An implementation of Hoeffding trees, a form of streaming decision tree for classification. Given labeled data, a Hoeffding tree can be trained and saved for later use, or a pre-trained Hoeffding tree can be used for predicting the classifications of new points.

Usage

```
hoeffding_tree(
  batch_mode = FALSE,
  bins = 10,
  confidence = 0.95,
  info_gain = FALSE,
  input_model = NA,
  labels = NA,
  max_samples = 5000,
  min_samples = 100,
  numeric_split_strategy = "binary",
  observations_before_binning = 100,
  passes = 1,
```

```
  test = NA,
  test_labels = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

batch_mode	If true, samples will be considered in batch instead of as a stream. This generally results in better trees but at the cost of memory usage and runtime. Default value "FALSE" (logical).
bins	If the 'domingos' split strategy is used, this specifies the number of bins for each numeric split. Default value "10" (integer).
confidence	Confidence before splitting (between 0 and 1). Default value "0.95" (numeric).
info_gain	If set, information gain is used instead of Gini impurity for calculating Hoeffding bounds. Default value "FALSE" (logical).
input_model	Input trained Hoeffding tree model (HoeffdingTreeModel).
labels	Labels for training dataset (integer row).
max_samples	Maximum number of samples before splitting. Default value "5000" (integer).
min_samples	Minimum number of samples before splitting. Default value "100" (integer).
numeric_split_strategy	The splitting strategy to use for numeric features: 'domingos' or 'binary'. Default value "binary" (character).
observations_before_binning	If the 'domingos' split strategy is used, this specifies the number of samples observed before binning is performed. Default value "100" (integer).
passes	Number of passes to take over the dataset. Default value "1" (integer).
test	Testing dataset (may be categorical) (numeric matrix/data.frame with info).
test_labels	Labels of test data (integer row).
training	Training dataset (may be categorical) (numeric matrix/data.frame with info).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program implements Hoeffding trees, a form of streaming decision tree suited best for large (or streaming) datasets. This program supports both categorical and numeric data. Given an input dataset, this program is able to train the tree with numerous training options, and save the model to a file. The program is also able to use a trained model or a model from file in order to predict classes for a given test set.

The training file and associated labels are specified with the "training" and "labels" parameters, respectively. Optionally, if "labels" is not specified, the labels are assumed to be the last dimension of the training dataset.

The training may be performed in batch mode (like a typical decision tree algorithm) by specifying the "batch_mode" option, but this may not be the best option for large datasets.

When a model is trained, it may be saved via the "output_model" output parameter. A model may be loaded from file for further training or testing with the "input_model" parameter.

Test data may be specified with the "test" parameter, and if performance statistics are desired for that test set, labels may be specified with the "test_labels" parameter. Predictions for each test point may be saved with the "predictions" output parameter, and class probabilities for each prediction may be saved with the "probabilities" output parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output for trained Hoeffding tree model (HoeffdingTreeModel).
`predictions`	Matrix to output label predictions for test data into (integer row).
`probabilities`	In addition to predicting labels, provide rediction probabilities in this matrix (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to train a Hoeffding tree with confidence 0.99 with data
# "dataset", saving the trained tree to "tree", the following command may be
# used:
#
# \dontrun{
# output <- hoeffding_tree(training=dataset, confidence=0.99)
# tree <- output$output_model
# }
#
# Then, this tree may be used to make predictions on the test set "test_set",
# saving the predictions into "predictions" and the class probabilities into
# "class_probs" with the following command:
#
# \dontrun{
# output <- hoeffding_tree(input_model=tree, test=test_set)
# predictions <- output$predictions
# class_probs <- output$probabilities
# }
```

`image_converter` *Image Converter*

Description

A utility to load an image or set of images into a single dataset that can then be used by other mlpack methods and utilities. This can also unpack an image dataset into individual files, for instance after mlpack methods have been used.

Usage

```
image_converter(
  input,
  channels = 0,
  dataset = NA,
  height = 0,
  quality = 90,
  save = FALSE,
  verbose = getOption("mlpack.verbose", FALSE),
  width = 0
)
```

Arguments

`input`	Image filenames which have to be loaded/saved (character vector).
`channels`	Number of channels in the image. Default value "0" (integer).
`dataset`	Input matrix to save as images (numeric matrix).
`height`	Height of the images. Default value "0" (integer).
`quality`	Compression of the image if saved as jpg (0-100). Default value "90" (integer).
`save`	Save a dataset as images. Default value "FALSE" (logical).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
`width`	Width of the image. Default value "0" (integer).

Details

This utility takes an image or an array of images and loads them to a matrix. You can optionally specify the height "height" width "width" and channel "channels" of the images that needs to be loaded; otherwise, these parameters will be automatically detected from the image. There are other options too, that can be specified such as "quality".

You can also provide a dataset and save them as images using "dataset" and "save" as an parameter.

Value

A list with several components defining the class attributes:

output	Matrix to save images data to, Onlyneeded if you are specifying 'save' option (numeric matrix).

Author(s)

mlpack developers

Examples

```
#  An example to load an image :
#
# \dontrun{
# Y <- image_converter(input=X, height=256, width=256, channels=3)
# }
#
#  An example to save an image is :
#
# \dontrun{
# image_converter(input=X, height=256, width=256, channels=3, dataset=Y,
#   save=TRUE)
# }
```

---

kde *Kernel Density Estimation*

Description

An implementation of kernel density estimation with dual-tree algorithms. Given a set of reference points and query points and a kernel function, this can estimate the density function at the location of each query point using trees; trees that are built can be saved for later use.

Usage

```
kde(
  abs_error = 0,
  algorithm = "dual-tree",
  bandwidth = 1,
  initial_sample_size = 100,
  input_model = NA,
  kernel = "gaussian",
  mc_break_coef = 0.4,
  mc_entry_coef = 3,
  mc_probability = 0.95,
  monte_carlo = FALSE,
  query = NA,
```

```
  reference = NA,
  rel_error = 0.05,
  tree = "kd-tree",
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

abs_error	Relative error tolerance for the prediction. Default value "0" (numeric).
algorithm	Algorithm to use for the prediction.('dual-tree', 'single-tree'). Default value "dual-tree" (character).
bandwidth	Bandwidth of the kernel. Default value "1" (numeric).
initial_sample_size	Initial sample size for Monte Carlo estimations. Default value "100" (integer).
input_model	Contains pre-trained KDE model (KDEModel).
kernel	Kernel to use for the prediction.('gaussian', 'epanechnikov', 'laplacian', 'spherical', 'triangular'). Default value "gaussian" (character).
mc_break_coef	Controls what fraction of the amount of node's descendants is the limit for the sample size before it recurses. Default value "0.4" (numeric).
mc_entry_coef	Controls how much larger does the amount of node descendants has to be compared to the initial sample size in order to be a candidate for Monte Carlo estimations. Default value "3" (numeric).
mc_probability	Probability of the estimation being bounded by relative error when using Monte Carlo estimations. Default value "0.95" (numeric).
monte_carlo	Whether to use Monte Carlo estimations when possible. Default value "FALSE" (logical).
query	Query dataset to KDE on (numeric matrix).
reference	Input reference dataset use for KDE (numeric matrix).
rel_error	Relative error tolerance for the prediction. Default value "0.05" (numeric).
tree	Tree to use for the prediction.('kd-tree', 'ball-tree', 'cover-tree', 'octree', 'r-tree'). Default value "kd-tree" (character).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs a Kernel Density Estimation. KDE is a non-parametric way of estimating probability density function. For each query point the program will estimate its probability density by applying a kernel function to each reference point. The computational complexity of this is O(N^2) where there are N query points and N reference points, but this implementation will typically see better performance as it uses an approximate dual or single tree algorithm for acceleration.

Dual or single tree optimization avoids many barely relevant calculations (as kernel function values decrease with distance), so it is an approximate computation. You can specify the maximum relative error tolerance for each query value with "rel_error" as well as the maximum absolute error

tolerance with the parameter "abs_error". This program runs using an Euclidean metric. Kernel function can be selected using the "kernel" option. You can also choose what which type of tree to use for the dual-tree algorithm with "tree". It is also possible to select whether to use dual-tree algorithm or single-tree algorithm using the "algorithm" option.

Monte Carlo estimations can be used to accelerate the KDE estimate when the Gaussian Kernel is used. This provides a probabilistic guarantee on the the error of the resulting KDE instead of an absolute guarantee.To enable Monte Carlo estimations, the "monte_carlo" flag can be used, and success probability can be set with the "mc_probability" option. It is possible to set the initial sample size for the Monte Carlo estimation using "initial_sample_size". This implementation will only consider a node, as a candidate for the Monte Carlo estimation, if its number of descendant nodes is bigger than the initial sample size. This can be controlled using a coefficient that will multiply the initial sample size and can be set using "mc_entry_coef". To avoid using the same amount of computations an exact approach would take, this program recurses the tree whenever a fraction of the amount of the node's descendant points have already been computed. This fraction is set using "mc_break_coef".

Value

A list with several components defining the class attributes:

`output_model`	If specified, the KDE model will be saved here (KDEModel).
`predictions`	Vector to store density predictions (numeric column).

Author(s)

mlpack developers

Examples

```
# For example, the following will run KDE using the data in "ref_data" for
# training and the data in "qu_data" as query data. It will apply an
# Epanechnikov kernel with a 0.2 bandwidth to each reference point and use a
# KD-Tree for the dual-tree optimization. The returned predictions will be
# within 5% of the real KDE value for each query point.
#
# \dontrun{
# output <- kde(reference=ref_data, query=qu_data, bandwidth=0.2,
#   kernel="epanechnikov", tree="kd-tree", rel_error=0.05)
# out_data <- output$predictions
# }
#
# the predicted density estimations will be stored in "out_data".
# If no "query" is provided, then KDE will be computed on the "reference"
# dataset.
# It is possible to select either a reference dataset or an input model but
# not both at the same time. If an input model is selected and parameter
# values are not set (e.g. "bandwidth") then default parameter values will be
# used.
#
# In addition to the last program call, it is also possible to activate Monte
# Carlo estimations if a Gaussian kernel is used. This can provide faster
```

```
# results, but the KDE will only have a probabilistic guarantee of meeting
# the desired error bound (instead of an absolute guarantee). The following
# example will run KDE using a Monte Carlo estimation when possible. The
# results will be within a 5% of the real KDE value with a 95% probability.
# Initial sample size for the Monte Carlo estimation will be 200 points and a
# node will be a candidate for the estimation only when it contains 700 (i.e.
# 3.5*200) points. If a node contains 700 points and 420 (i.e. 0.6*700) have
# already been sampled, then the algorithm will recurse instead of keep
# sampling.
#
# \dontrun{
# output <- kde(reference=ref_data, query=qu_data, bandwidth=0.2,
#   kernel="gaussian", tree="kd-tree", rel_error=0.05, monte_carlo=,
#   mc_probability=0.95, initial_sample_size=200, mc_entry_coef=3.5,
#   mc_break_coef=0.6)
# out_data <- output$predictions
# }
```

---

kernel_pca *Kernel Principal Components Analysis*

Description

An implementation of Kernel Principal Components Analysis (KPCA). This can be used to perform nonlinear dimensionality reduction or preprocessing on a given dataset.

Usage

```
kernel_pca(
  input,
  kernel,
  bandwidth = 1,
  center = FALSE,
  degree = 1,
  kernel_scale = 1,
  new_dimensionality = 0,
  nystroem_method = FALSE,
  offset = 0,
  sampling = "kmeans",
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Input dataset to perform KPCA on (numeric matrix).
kernel	The kernel to use; see the above documentation for the list of usable kernels (character).
bandwidth	Bandwidth, for 'gaussian' and 'laplacian' kernels. Default value "1" (numeric).

`center` If set, the transformed data will be centered about the origin. Default value "FALSE" (logical).

`degree` Degree of polynomial, for 'polynomial' kernel. Default value "1" (numeric).

`kernel_scale` Scale, for 'hyptan' kernel. Default value "1" (numeric).

`new_dimensionality` If not 0, reduce the dimensionality of the output dataset by ignoring the dimensions with the smallest eigenvalues. Default value "0" (integer).

`nystroem_method` If set, the Nystroem method will be used. Default value "FALSE" (logical).

`offset` Offset, for 'hyptan' and 'polynomial' kernels. Default value "0" (numeric).

`sampling` Sampling scheme to use for the Nystroem method: 'kmeans', 'random', 'ordered. Default value "kmeans" (character).

`verbose` Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs Kernel Principal Components Analysis (KPCA) on the specified dataset with the specified kernel. This will transform the data onto the kernel principal components, and optionally reduce the dimensionality by ignoring the kernel principal components with the smallest eigenvalues.

For the case where a linear kernel is used, this reduces to regular PCA.

The kernels that are supported are listed below:

* 'linear': the standard linear dot product (same as normal PCA): 'K(x, y) = x^T y'

* 'gaussian': a Gaussian kernel; requires bandwidth: 'K(x, y) = exp(-(|| x - y || ^ 2) / (2 * (bandwidth ^ 2)))'

* 'polynomial': polynomial kernel; requires offset and degree: 'K(x, y) = (x^T y + offset) ^ degree'

* 'hyptan': hyperbolic tangent kernel; requires scale and offset: 'K(x, y) = tanh(scale * (x^T y) + offset)'

* 'laplacian': Laplacian kernel; requires bandwidth: 'K(x, y) = exp(-(|| x - y ||) / bandwidth)'

* 'epanechnikov': Epanechnikov kernel; requires bandwidth: 'K(x, y) = max(0, 1 - || x - y ||^2 / bandwidth^2)'

* 'cosine': cosine distance: 'K(x, y) = 1 - (x^T y) / (|| x || * || y ||)'

The parameters for each of the kernels should be specified with the options "bandwidth", "kernel_scale", "offset", or "degree" (or a combination of those parameters).

Optionally, the Nystroem method ("Using the Nystroem method to speed up kernel machines", 2001) can be used to calculate the kernel matrix by specifying the "nystroem_method" parameter. This approach works by using a subset of the data as basis to reconstruct the kernel matrix; to specify the sampling scheme, the "sampling" parameter is used. The sampling scheme for the Nystroem method can be chosen from the following list: 'kmeans', 'random', 'ordered'.

Value

A list with several components defining the class attributes:

output	Matrix to save modified dataset to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, the following command will perform KPCA on the dataset "input"
# using the Gaussian kernel, and saving the transformed data to
# "transformed":
#
# \dontrun{
# transformed <- kernel_pca(input=input, kernel="gaussian")
# }
```

---

kfn *k-Furthest-Neighbors Search*

Description

An implementation of k-furthest-neighbor search using single-tree and dual-tree algorithms. Given a set of reference points and query points, this can find the k furthest neighbors in the reference set of each query point using trees; trees that are built can be saved for future use.

Usage

```
kfn(
  algorithm = "dual_tree",
  epsilon = 0,
  input_model = NA,
  k = 0,
  leaf_size = 20,
  percentage = 1,
  query = NA,
  random_basis = FALSE,
  reference = NA,
  seed = 0,
  tree_type = "kd",
  true_distances = NA,
  true_neighbors = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

algorithm	Type of neighbor search: 'naive', 'single_tree', 'dual_tree', 'greedy'. Default value "dual_tree" (character).
epsilon	If specified, will do approximate furthest neighbor search with given relative error. Must be in the range [0,1). Default value "0" (numeric).
input_model	Pre-trained kFN model (KFNModel).
k	Number of furthest neighbors to find. Default value "0" (integer).
leaf_size	Leaf size for tree building (used for kd-trees, vp trees, random projection trees, UB trees, R trees, R* trees, X trees, Hilbert R trees, R+ trees, R++ trees, and octrees). Default value "20" (integer).
percentage	If specified, will do approximate furthest neighbor search. Must be in the range (0,1] (decimal form). Resultant neighbors will be at least (p*100) Default value "1" (numeric).
query	Matrix containing query points (optional) (numeric matrix).
random_basis	Before tree-building, project the data onto a random orthogonal basis. Default value "FALSE" (logical).
reference	Matrix containing the reference dataset (numeric matrix).
seed	Random seed (if 0, std::time(NULL) is used). Default value "0" (integer).
tree_type	Type of tree to use: 'kd', 'vp', 'rp', 'max-rp', 'ub', 'cover', 'r', 'r-star', 'x', 'ball', 'hilbert-r', 'r-plus', 'r-plus-plus', 'oct'. Default value "kd" (character).
true_distances	Matrix of true distances to compute the effective error (average relative error) (it is printed when -v is specified) (numeric matrix).
true_neighbors	Matrix of true neighbors to compute the recall (it is printed when -v is specified) (integer matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program will calculate the k-furthest-neighbors of a set of points. You may specify a separate set of reference points and query points, or just a reference set which will be used as both the reference and query set.

Value

A list with several components defining the class attributes:

distances	Matrix to output distances into (numeric matrix).
neighbors	Matrix to output neighbors into (integer matrix).
output_model	If specified, the kFN model will be output here (KFNModel).

Author(s)

mlpack developers

Examples

```
# For example, the following will calculate the 5 furthest neighbors of
# eachpoint in "input" and store the distances in "distances" and the
# neighbors in "neighbors":
#
# \dontrun{
# output <- kfn(k=5, reference=input)
# distances <- output$distances
# neighbors <- output$neighbors
# }
# #' #
# The output files are organized such that row i and column j in the
# neighbors output matrix corresponds to the index of the point in the
# reference set which is the j'th furthest neighbor from the point in the
# query set with index i.  Row i and column j in the distances output file
# corresponds to the distance between those two points.
```

---

kmeans *K-Means Clustering*

Description

An implementation of several strategies for efficient k-means clustering. Given a dataset and a value of k, this computes and returns a k-means clustering on that data.

Usage

```
kmeans(
  clusters,
  input,
  algorithm = "naive",
  allow_empty_clusters = FALSE,
  in_place = FALSE,
  initial_centroids = NA,
  kill_empty_clusters = FALSE,
  kmeans_plus_plus = FALSE,
  labels_only = FALSE,
  max_iterations = 1000,
  percentage = 0.02,
  refined_start = FALSE,
  samplings = 100,
  seed = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

clusters	Number of clusters to find (0 autodetects from initial centroids) (integer).
input	Input dataset to perform clustering on (numeric matrix).
algorithm	Algorithm to use for the Lloyd iteration ('naive', 'pelleg-moore', 'elkan', 'hamerly', 'dualtree', or 'dualtree-covertree'). Default value "naive" (character).
allow_empty_clusters	Allow empty clusters to be persist. Default value "FALSE" (logical).
in_place	If specified, a column containing the learned cluster assignments will be added to the input dataset file. In this case, –output_file is overridden. (Do not use in Python.. Default value "FALSE" (logical).
initial_centroids	Start with the specified initial centroids (numeric matrix).
kill_empty_clusters	Remove empty clusters when they occur. Default value "FALSE" (logical).
kmeans_plus_plus	Use the k-means++ initialization strategy to choose initial points. Default value "FALSE" (logical).
labels_only	Only output labels into output file. Default value "FALSE" (logical).
max_iterations	Maximum number of iterations before k-means terminates. Default value "1000" (integer).
percentage	Percentage of dataset to use for each refined start sampling (use when –refined_start is specified). Default value "0.02" (numeric).
refined_start	Use the refined initial point strategy by Bradley and Fayyad to choose initial points. Default value "FALSE" (logical).
samplings	Number of samplings to perform for refined start (use when –refined_start is specified). Default value "100" (integer).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs K-Means clustering on the given dataset. It can return the learned cluster assignments, and the centroids of the clusters. Empty clusters are not allowed by default; when a cluster becomes empty, the point furthest from the centroid of the cluster with maximum variance is taken to fill that cluster.

Optionally, the strategy to choose initial centroids can be specified. The k-means++ algorithm can be used to choose initial centroids with the "kmeans_plus_plus" parameter. The Bradley and Fayyad approach ("Refining initial points for k-means clustering", 1998) can be used to select initial points by specifying the "refined_start" parameter. This approach works by taking random samplings of the dataset; to specify the number of samplings, the "samplings" parameter is used, and to specify the percentage of the dataset to be used in each sample, the "percentage" parameter is used (it should be a value between 0.0 and 1.0).

There are several options available for the algorithm used for each Lloyd iteration, specified with the "algorithm" option. The standard O(kN) approach can be used ('naive'). Other options include the Pelleg-Moore tree-based algorithm ('pelleg-moore'), Elkan's triangle-inequality based algorithm ('elkan'), Hamerly's modification to Elkan's algorithm ('hamerly'), the dual-tree k-means algorithm ('dualtree'), and the dual-tree k-means algorithm using the cover tree ('dualtree-covertree').

The behavior for when an empty cluster is encountered can be modified with the "allow_empty_clusters" option. When this option is specified and there is a cluster owning no points at the end of an iteration, that cluster's centroid will simply remain in its position from the previous iteration. If the "kill_empty_clusters" option is specified, then when a cluster owns no points at the end of an iteration, the cluster centroid is simply filled with DBL_MAX, killing it and effectively reducing k for the rest of the computation. Note that the default option when neither empty cluster option is specified can be time-consuming to calculate; therefore, specifying either of these parameters will often accelerate runtime.

Initial clustering assignments may be specified using the "initial_centroids" parameter, and the maximum number of iterations may be specified with the "max_iterations" parameter.

Value

A list with several components defining the class attributes:

`centroid`	If specified, the centroids of each cluster will be written to the given file (numeric matrix).
`output`	Matrix to store output labels or labeled data to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# As an example, to use Hamerly's algorithm to perform k-means clustering
# with k=10 on the dataset "data", saving the centroids to "centroids" and
# the assignments for each point to "assignments", the following command
# could be used:
#
# \dontrun{
# output <- kmeans(input=data, clusters=10)
# assignments <- output$output
# centroids <- output$centroid
# }
#
# To run k-means on that same dataset with initial centroids specified in
# "initial" with a maximum of 500 iterations, storing the output centroids in
# "final" the following command may be used:
#
# \dontrun{
# output <- kmeans(input=data, initial_centroids=initial, clusters=10,
#   max_iterations=500)
# final <- output$centroid
# }
```

knn *k-Nearest-Neighbors Search*

Description

An implementation of k-nearest-neighbor search using single-tree and dual-tree algorithms. Given a set of reference points and query points, this can find the k nearest neighbors in the reference set of each query point using trees; trees that are built can be saved for future use.

Usage

```
knn(
  algorithm = "dual_tree",
  epsilon = 0,
  input_model = NA,
  k = 0,
  leaf_size = 20,
  query = NA,
  random_basis = FALSE,
  reference = NA,
  rho = 0.7,
  seed = 0,
  tau = 0,
  tree_type = "kd",
  true_distances = NA,
  true_neighbors = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

algorithm	Type of neighbor search: 'naive', 'single_tree', 'dual_tree', 'greedy'. Default value "dual_tree" (character).
epsilon	If specified, will do approximate nearest neighbor search with given relative error. Default value "0" (numeric).
input_model	Pre-trained kNN model (KNNModel).
k	Number of nearest neighbors to find. Default value "0" (integer).
leaf_size	Leaf size for tree building (used for kd-trees, vp trees, random projection trees, UB trees, R trees, R* trees, X trees, Hilbert R trees, R+ trees, R++ trees, spill trees, and octrees). Default value "20" (integer).
query	Matrix containing query points (optional) (numeric matrix).
random_basis	Before tree-building, project the data onto a random orthogonal basis. Default value "FALSE" (logical).
reference	Matrix containing the reference dataset (numeric matrix).
rho	Balance threshold (only valid for spill trees). Default value "0.7" (numeric).

seed	Random seed (if 0, std::time(NULL) is used). Default value "0" (integer).
tau	Overlapping size (only valid for spill trees). Default value "0" (numeric).
tree_type	Type of tree to use: 'kd', 'vp', 'rp', 'max-rp', 'ub', 'cover', 'r', 'r-star', 'x', 'ball', 'hilbert-r', 'r-plus', 'r-plus-plus', 'spill', 'oct'. Default value "kd" (character).
true_distances	Matrix of true distances to compute the effective error (average relative error) (it is printed when -v is specified) (numeric matrix).
true_neighbors	Matrix of true neighbors to compute the recall (it is printed when -v is specified) (integer matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program will calculate the k-nearest-neighbors of a set of points using kd-trees or cover trees (cover tree support is experimental and may be slow). You may specify a separate set of reference points and query points, or just a reference set which will be used as both the reference and query set.

Value

A list with several components defining the class attributes:

distances	Matrix to output distances into (numeric matrix).
neighbors	Matrix to output neighbors into (integer matrix).
output_model	If specified, the kNN model will be output here (KNNModel).

Author(s)

mlpack developers

Examples

```
# For example, the following command will calculate the 5 nearest neighbors
# of each point in "input" and store the distances in "distances" and the
# neighbors in "neighbors":
#
# \dontrun{
# output <- knn(k=5, reference=input)
# neighbors <- output$neighbors
# distances <- output$distances
# }
# #' #
# The output is organized such that row i and column j in the neighbors
# output matrix corresponds to the index of the point in the reference set
# which is the j'th nearest neighbor from the point in the query set with
# index i.  Row j and column i in the distances output matrix corresponds to
# the distance between those two points.
```

krann *K-Rank-Approximate-Nearest-Neighbors (kRANN)*

Description

An implementation of rank-approximate k-nearest-neighbor search (kRANN) using single-tree and dual-tree algorithms. Given a set of reference points and query points, this can find the k nearest neighbors in the reference set of each query point using trees; trees that are built can be saved for future use.

Usage

```
krann(
  alpha = 0.95,
  first_leaf_exact = FALSE,
  input_model = NA,
  k = 0,
  leaf_size = 20,
  naive = FALSE,
  query = NA,
  random_basis = FALSE,
  reference = NA,
  sample_at_leaves = FALSE,
  seed = 0,
  single_mode = FALSE,
  single_sample_limit = 20,
  tau = 5,
  tree_type = "kd",
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

alpha	The desired success probability. Default value "0.95" (numeric).
first_leaf_exact	The flag to trigger sampling only after exactly exploring the first leaf. Default value "FALSE" (logical).
input_model	Pre-trained kNN model (RAModel).
k	Number of nearest neighbors to find. Default value "0" (integer).
leaf_size	Leaf size for tree building (used for kd-trees, UB trees, R trees, R* trees, X trees, Hilbert R trees, R+ trees, R++ trees, and octrees). Default value "20" (integer).
naive	If true, sampling will be done without using a tree. Default value "FALSE" (logical).
query	Matrix containing query points (optional) (numeric matrix).

`random_basis` Before tree-building, project the data onto a random orthogonal basis. Default value "FALSE" (logical).

`reference` Matrix containing the reference dataset (numeric matrix).

`sample_at_leaves` The flag to trigger sampling at leaves. Default value "FALSE" (logical).

`seed` Random seed (if 0, std::time(NULL) is used). Default value "0" (integer).

`single_mode` If true, single-tree search is used (as opposed to dual-tree search. Default value "FALSE" (logical).

`single_sample_limit` The limit on the maximum number of samples (and hence the largest node you can approximate). Default value "20" (integer).

`tau` The allowed rank-error in terms of the percentile of the data. Default value "5" (numeric).

`tree_type` Type of tree to use: 'kd', 'ub', 'cover', 'r', 'x', 'r-star', 'hilbert-r', 'r-plus', 'r-plus-plus', 'oct'. Default value "kd" (character).

`verbose` Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program will calculate the k rank-approximate-nearest-neighbors of a set of points. You may specify a separate set of reference points and query points, or just a reference set which will be used as both the reference and query set. You must specify the rank approximation (in success probability).

Value

A list with several components defining the class attributes:

`distances` Matrix to output distances into (numeric matrix).

`neighbors` Matrix to output neighbors into (integer matrix).

`output_model` If specified, the kNN model will be output here (RAModel).

Author(s)

mlpack developers

Examples

```
# For example, the following will return 5 neighbors from the top 0.1% of the
# data (with probability 0.95) for each point in "input" and store the
# distances in "distances" and the neighbors in "neighbors.csv":
#
# \dontrun{
# output <- krann(reference=input, k=5, tau=0.1)
# distances <- output$distances
```

```
# neighbors <- output$neighbors
# }
# #' #
# Note that tau must be set such that the number of points in the
# corresponding percentile of the data is greater than k.  Thus, if we choose
# tau = 0.1 with a dataset of 1000 points and k = 5, then we are attempting
# to choose 5 nearest neighbors out of the closest 1 point -- this is invalid
# and the program will terminate with an error message.
#
# The output matrices are organized such that row i and column j in the
# neighbors output file corresponds to the index of the point in the
# reference set which is the i'th nearest neighbor from the point in the
# query set with index j.  Row i and column j in the distances output file
# corresponds to the distance between those two points.
```

`lars` *LARS*

Description

An implementation of Least Angle Regression (Stagewise/laSso), also known as LARS. This can train a LARS/LASSO/Elastic Net model and use that model or a pre-trained model to output regression predictions for a test set.

Usage

```
lars(
  input = NA,
  input_model = NA,
  lambda1 = 0,
  lambda2 = 0,
  no_intercept = FALSE,
  no_normalize = FALSE,
  responses = NA,
  test = NA,
  use_cholesky = FALSE,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Matrix of covariates (X) (numeric matrix).
`input_model`	Trained LARS model to use (LARS).
`lambda1`	Regularization parameter for l1-norm penalty. Default value "0" (numeric).
`lambda2`	Regularization parameter for l2-norm penalty. Default value "0" (numeric).
`no_intercept`	Do not fit an intercept in the model. Default value "FALSE" (logical).

`no_normalize`	Do not normalize data to unit variance before modeling. Default value "FALSE" (logical).
`responses`	Matrix of responses/observations (y) (numeric matrix).
`test`	Matrix containing points to regress on (test points) (numeric matrix).
`use_cholesky`	Use Cholesky decomposition during computation rather than explicitly computing the full Gram matrix. Default value "FALSE" (logical).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of LARS: Least Angle Regression (Stagewise/laSso). This is a stage-wise homotopy-based algorithm for L1-regularized linear regression (LASSO) and L1+L2-regularized linear regression (Elastic Net).

This program is able to train a LARS/LASSO/Elastic Net model or load a model from file, output regression predictions for a test set, and save the trained model to a file. The LARS algorithm is described in more detail below:

Let X be a matrix where each row is a point and each column is a dimension, and let y be a vector of targets.

The Elastic Net problem is to solve

min_beta 0.5 || X * beta - y ||_2^2 + lambda_1 ||beta||_1 + 0.5 lambda_2 ||beta||_2^2

If lambda1 > 0 and lambda2 = 0, the problem is the LASSO. If lambda1 > 0 and lambda2 > 0, the problem is the Elastic Net. If lambda1 = 0 and lambda2 > 0, the problem is ridge regression. If lambda1 = 0 and lambda2 = 0, the problem is unregularized linear regression.

For efficiency reasons, it is not recommended to use this algorithm with "lambda1" = 0. In that case, use the 'linear_regression' program, which implements both unregularized linear regression and ridge regression.

To train a LARS/LASSO/Elastic Net model, the "input" and "responses" parameters must be given. The "lambda1", "lambda2", and "use_cholesky" parameters control the training options. A trained model can be saved with the "output_model". If no training is desired at all, a model can be passed via the "input_model" parameter.

The program can also provide predictions for test data using either the trained model or the given input model. Test points can be specified with the "test" parameter. Predicted responses to the test points can be saved with the "output_predictions" output parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output LARS model (LARS).
`output_predictions`	If –test_file is specified, this file is where the predicted responses will be saved (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, the following command trains a model on the data "data" and
# responses "responses" with lambda1 set to 0.4 and lambda2 set to 0 (so,
# LASSO is being solved), and then the model is saved to "lasso_model":
#
# \dontrun{
# output <- lars(input=data, responses=responses, lambda1=0.4, lambda2=0)
# lasso_model <- output$output_model
# }
#
# The following command uses the "lasso_model" to provide predicted responses
# for the data "test" and save those responses to "test_predictions":
#
# \dontrun{
# output <- lars(input_model=lasso_model, test=test)
# test_predictions <- output$output_predictions
# }
```

---

lars_train *LARS Training*

Description

An implementation of Least Angle Regression (stagewise/lasso), also known as LARS. This can train a LARS/LASSO/Elastic Net model, and save the pre-trained model for later use to output regression predictions from a test set.

Usage

```
lars_train(
  input,
  responses,
  lambda1 = 0,
  lambda2 = 0,
  no_intercept = FALSE,
  no_normalize = FALSE,
  use_cholesky = FALSE,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Matrix of covariates (X) (numeric matrix).
responses	Row vector of responses/observations (y) (numeric row).

`lambda1`	Regularization parameter for l1-norm penalty. Default value "0" (numeric).
`lambda2`	Regularization parameter for l2-norm penalty. Default value "0" (numeric).
`no_intercept`	Do not fit an intercept in the model. Default value "FALSE" (logical).
`no_normalize`	Do not normalize data to unit variance before modeling. Default value "FALSE" (logical).
`use_cholesky`	Use Cholesky decomposition during computation rather than explicitly computing the full Gram matrix. Default value "FALSE" (logical).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of LARS: Least Angle Regression (stagewise/lasso). This is a stage-wise homotopy-based algorithm for L1-regularized linear regression (LASSO) and L1+L2-regularized linear regression (Elastic Net).

This program is able to train a LARS/LASSO/Elastic Net model or load a model from a file, output regression predictions for a test set, and save the trained model to a file. The LARS algorithm is described in more detail below:

Let X be a matrix where each row is a point and each column is a dimension, and let y be a vector of targets.

The Elastic Net problem is to solve

min_beta 0.5 || X * beta - y ||_2^2 + lambda_1 ||beta||_1 + 0.5 lambda_2 ||beta||_2^2

If lambda1 > 0 and lambda2 = 0, the problem is the LASSO. If lambda1 > 0 and lambda2 > 0, the problem is the Elastic Net. If lambda1 = 0 and lambda2 > 0, the problem is ridge regression. If lambda1 = 0 and lambda2 = 0, the problem is unregularized linear regression.

For efficiency reasons, it is not recommended to use this algorithm with "lambda1" = 0. In that case, use the 'linear_regression' program, which implements both unregularized linear regression and ridge regression.

To train a LARS/LASSO/Elastic Net model, the "input" and "responses" parameters must be given. The "lambda1", "lambda2", and "use_cholesky" parameters control the training options. A trained model can be saved with the "output_model". If no training is desired at all, a model can be passed via the "input_model" parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output LARS model (LARS).

Author(s)

mlpack developers

Examples

```
#
# #' # \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <- as.matrix(read.csv("http://datasets.mlpack.org/admission_predict.csv",
# header=FALSE))
# y <-
# as.matrix(read.csv("http://datasets.mlpack.org/admission_predict.responses.
# csv", header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- lars_train(input=X_train, responses=y_train, lambda1=1e-05,
#   lambda2=1e-06)
#   }
```

---

linear_regression *Simple Linear Regression and Prediction*

Description

An implementation of simple linear regression and ridge regression using ordinary least squares. Given a dataset and responses, a model can be trained and saved for later use, or a pre-trained model can be used to output regression predictions for a test set.

Usage

```
linear_regression(
  input_model = NA,
  lambda = 0,
  test = NA,
  training = NA,
  training_responses = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input_model	Existing LinearRegression model to use (LinearRegression).
lambda	Tikhonov regularization for ridge regression. If 0, the method reduces to linear regression. Default value "0" (numeric).
test	Matrix containing X' (test regressors) (numeric matrix).

`training` Matrix containing training set X (regressors) (numeric matrix).

`training_responses`
Optional vector containing y (responses). If not given, the responses are assumed to be the last row of the input file (numeric row).

`verbose` Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of simple linear regression and simple ridge regression using ordinary least squares. This solves the problem

y = X * b + e

where X (specified by "training") and y (specified either as the last column of the input matrix "training" or via the "training_responses" parameter) are known and b is the desired variable. If the covariance matrix (X'X) is not invertible, or if the solution is overdetermined, then specify a Tikhonov regularization constant (with "lambda") greater than 0, which will regularize the covariance matrix to make it invertible. The calculated b may be saved with the "output_predictions" output parameter.

Optionally, the calculated value of b is used to predict the responses for another matrix X' (specified by the "test" parameter):

y' = X' * b

and the predicted responses y' may be saved with the "output_predictions" output parameter. This type of regression is related to least-angle regression, which mlpack implements as the 'lars' program.

Value

A list with several components defining the class attributes:

`output_model` Output LinearRegression model (LinearRegression).

`output_predictions`
If –test_file is specified, this matrix is where the predicted responses will be saved (numeric row).

Author(s)

mlpack developers

Examples

```
# For example, to run a linear regression on the dataset "X" with responses
# "y", saving the trained model to "lr_model", the following command could be
# used:
#
# \dontrun{
# output <- linear_regression(training=X, training_responses=y)
# lr_model <- output$output_model
```

```
# }
#
# Then, to use "lr_model" to predict responses for a test set "X_test",
# saving the predictions to "X_test_responses", the following command could
# be used:
#
# \dontrun{
# output <- linear_regression(input_model=lr_model, test=X_test)
# X_test_responses <- output$output_predictions
# }
```

`linear_regression_train` *Simple Linear Regression*

Description

Train a linear regression model.

Usage

```
linear_regression_train(
  training,
  lambda = 0,
  training_responses = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`training`	Matrix containing training set X (regressors) (numeric matrix).
`lambda`	Tikhonov regularization for ridge regression. If 0, the method reduces to linear regression. Default value "0" (numeric).
`training_responses`	Optional vector containing y (responses). If not given, the responses are assumed to be the last row of the input file (numeric row).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of simple linear regression and simple ridge regression using ordinary least squares. This solves the problem

y = X * b + e

Value

A list with several components defining the class attributes:

`output_model` Output LinearRegression model (LinearRegression).

Author(s)

mlpack developers

Examples

```
#
# #' # \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <-
# as.matrix(read.csv("https://datasets.mlpack.org/admission_predict.csv",
# header=FALSE))
# y <-
# as.matrix(read.csv("https://datasets.mlpack.org/admission_predict.responses
# .csv", header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- linear_regression_train(training=X_train,
# training_responses=y_train)
#   }
```

`linear_svm` *Linear SVM is an L2-regularized support vector machine.*

Description

An implementation of linear SVM for multiclass classification. Given labeled data, a model can be trained and saved for future use; or, a pre-trained model can be used to classify new points.

Usage

```
linear_svm(
  delta = 1,
  epochs = 50,
  input_model = NA,
  labels = NA,
  lambda = 1e-04,
  max_iterations = 10000,
```

```
  no_intercept = FALSE,
  num_classes = 0,
  optimizer = "lbfgs",
  seed = 0,
  shuffle = FALSE,
  step_size = 0.01,
  test = NA,
  test_labels = NA,
  tolerance = 1e-10,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

delta	Margin of difference between correct class and other classes. Default value "1" (numeric).
epochs	Maximum number of full epochs over dataset for psg. Default value "50" (integer).
input_model	Existing model (parameters) (LinearSVMModel).
labels	A matrix containing labels (0 or 1) for the points in the training set (y) (integer row).
lambda	L2-regularization parameter for training. Default value "0.0001" (numeric).
max_iterations	Maximum iterations for optimizer (0 indicates no limit). Default value "10000" (integer).
no_intercept	Do not add the intercept term to the model. Default value "FALSE" (logical).
num_classes	Number of classes for classification; if unspecified (or 0), the number of classes found in the labels will be used. Default value "0" (integer).
optimizer	Optimizer to use for training ('lbfgs' or 'psgd'). Default value "lbfgs" (character).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
shuffle	Don't shuffle the order in which data points are visited for parallel SGD. Default value "FALSE" (logical).
step_size	Step size for parallel SGD optimizer. Default value "0.01" (numeric).
test	Matrix containing test dataset (numeric matrix).
test_labels	Matrix containing test labels (integer row).
tolerance	Convergence tolerance for optimizer. Default value "1e-10" (numeric).
training	A matrix containing the training set (the matrix of predictors, X) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of linear SVMs that uses either L-BFGS or parallel SGD (stochastic gradient descent) to train the model.

This program allows loading a linear SVM model (via the "input_model" parameter) or training a linear SVM model given training data (specified with the "training" parameter), or both those things at once. In addition, this program allows classification on a test dataset (specified with the "test" parameter) and the classification results may be saved with the "predictions" output parameter. The trained linear SVM model may be saved using the "output_model" output parameter.

The training data, if specified, may have class labels as its last dimension. Alternately, the "labels" parameter may be used to specify a separate vector of labels.

When a model is being trained, there are many options. L2 regularization (to prevent overfitting) can be specified with the "lambda" option, and the number of classes can be manually specified with the "num_classes"and if an intercept term is not desired in the model, the "no_intercept" parameter can be specified.Margin of difference between correct class and other classes can be specified with the "delta" option.The optimizer used to train the model can be specified with the "optimizer" parameter. Available options are 'psgd' (parallel stochastic gradient descent) and 'lbfgs' (the L-BFGS optimizer). There are also various parameters for the optimizer; the "max_iterations" parameter specifies the maximum number of allowed iterations, and the "tolerance" parameter specifies the tolerance for convergence. For the parallel SGD optimizer, the "step_size" parameter controls the step size taken at each iteration by the optimizer and the maximum number of epochs (specified with "epochs"). If the objective function for your data is oscillating between Inf and 0, the step size is probably too large. There are more parameters for the optimizers, but the C++ interface must be used to access these.

Optionally, the model can be used to predict the labels for another matrix of data points, if "test" is specified. The "test" parameter can be specified without the "training" parameter, so long as an existing linear SVM model is given with the "input_model" parameter. The output predictions from the linear SVM model may be saved with the "predictions" parameter.

Value

A list with several components defining the class attributes:

`output_model`	Output for trained linear svm model (LinearSVMModel).
`predictions`	If test data is specified, this matrix is where the predictions for the test set will be saved (integer row).
`probabilities`	If test data is specified, this matrix is where the class probabilities for the test set will be saved (numeric matrix).

Author(s)

mlpack developers

Examples

```
# As an example, to train a LinaerSVM on the data '"data"' with labels
# '"labels"' with L2 regularization of 0.1, saving the model to
# '"lsvm_model"', the following command may be used:
```

```
#
# \dontrun{
# output <- linear_svm(training=data, labels=labels, lambda=0.1, delta=1,
#   num_classes=0)
# lsvm_model <- output$output_model
# }
#
# Then, to use that model to predict classes for the dataset '"test"',
# storing the output predictions in '"predictions"', the following command
# may be used:
#
# \dontrun{
# output <- linear_svm(input_model=lsvm_model, test=test)
# predictions <- output$predictions
# }
```

---

lmnn *Large Margin Nearest Neighbors (LMNN)*

Description

An implementation of Large Margin Nearest Neighbors (LMNN), a distance learning technique. Given a labeled dataset, this learns a transformation of the data that improves k-nearest-neighbor performance; this can be useful as a preprocessing step.

Usage

```
lmnn(
  input,
  batch_size = 50,
  center = FALSE,
  distance = NA,
  k = 1,
  labels = NA,
  linear_scan = FALSE,
  max_iterations = 1e+05,
  normalize = FALSE,
  optimizer = "amsgrad",
  passes = 50,
  print_accuracy = FALSE,
  rank = 0,
  regularization = 0.5,
  seed = 0,
  step_size = 0.01,
  tolerance = 1e-07,
  update_interval = 1,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Input dataset to run LMNN on (numeric matrix).
batch_size	Batch size for mini-batch SGD. Default value "50" (integer).
center	Perform mean-centering on the dataset. It is useful when the centroid of the data is far from the origin. Default value "FALSE" (logical).
distance	Initial distance matrix to be used as starting poin (numeric matrix).
k	Number of target neighbors to use for each datapoint. Default value "1" (integer).
labels	Labels for input dataset (integer row).
linear_scan	Don't shuffle the order in which data points are visited for SGD or mini-batch SGD. Default value "FALSE" (logical).
max_iterations	Maximum number of iterations for L-BFGS (0 indicates no limit). Default value "100000" (integer).
normalize	Use a normalized starting point for optimization. Itis useful for when points are far apart, or when SGD is returning NaN. Default value "FALSE" (logical).
optimizer	Optimizer to use; 'amsgrad', 'bbsgd', 'sgd', or 'lbfgs'. Default value "amsgrad" (character).
passes	Maximum number of full passes over dataset for AMSGrad, BB_SGD and SGD. Default value "50" (integer).
print_accuracy	Print accuracies on initial and transformed datase. Default value "FALSE" (logical).
rank	Rank of distance matrix to be optimized.. Default value "0" (integer).
regularization	Regularization for LMNN objective function. Default value "0.5" (numeric).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
step_size	Step size for AMSGrad, BB_SGD and SGD (alpha). Default value "0.01" (numeric).
tolerance	Maximum tolerance for termination of AMSGrad, BB_SGD, SGD or L-BFGS. Default value "1e-07" (numeric).
update_interval	Number of iterations after which impostors need to be recalculated. Default value "1" (integer).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program implements Large Margin Nearest Neighbors, a distance learning technique. The method seeks to improve k-nearest-neighbor classification on a dataset. The method employes the strategy of reducing distance between similar labeled data points (a.k.a target neighbors) and increasing distance between differently labeled points (a.k.a impostors) using standard optimization techniques over the gradient of the distance between data points.

To work, this algorithm needs labeled data. It can be given as the last row of the input dataset (specified with "input"), or alternatively as a separate matrix (specified with "labels"). Additionally, a starting point for optimization (specified with "distance"can be given, having (r x d) dimensionality. Here r should satisfy 1 <= r <= d, Consequently a Low-Rank matrix will be optimized. Alternatively, Low-Rank distance can be learned by specifying the "rank"parameter (A Low-Rank matrix with uniformly distributed values will be used as initial learning point).

The program also requires number of targets neighbors to work with (specified with "k"), A regularization parameter can also be passed, It acts as a trade of between the pulling and pushing terms (specified with "regularization"), In addition, this implementation of LMNN includes a parameter to decide the interval after which impostors must be re-calculated (specified with "update_interval").

Output can either be the learned distance matrix (specified with "output"), or the transformed dataset (specified with "transformed_data"), or both. Additionally mean-centered dataset (specified with "centered_data") can be accessed given mean-centering (specified with "center") is performed on the dataset. Accuracy on initial dataset and final transformed dataset can be printed by specifying the "print_accuracy"parameter.

This implementation of LMNN uses AdaGrad, BigBatch_SGD, stochastic gradient descent, mini-batch stochastic gradient descent, or the L_BFGS optimizer.

AdaGrad, specified by the value 'adagrad' for the parameter "optimizer", uses maximum of past squared gradients. It primarily on six parameters: the step size (specified with "step_size"), the batch size (specified with "batch_size"), the maximum number of passes (specified with "passes"). Inaddition, a normalized starting point can be used by specifying the "normalize" parameter.

BigBatch_SGD, specified by the value 'bbsgd' for the parameter "optimizer", depends primarily on four parameters: the step size (specified with "step_size"), the batch size (specified with "batch_size"), the maximum number of passes (specified with "passes"). In addition, a normalized starting point can be used by specifying the "normalize" parameter.

Stochastic gradient descent, specified by the value 'sgd' for the parameter "optimizer", depends primarily on three parameters: the step size (specified with "step_size"), the batch size (specified with "batch_size"), and the maximum number of passes (specified with "passes"). In addition, a normalized starting point can be used by specifying the "normalize" parameter. Furthermore, mean-centering can be performed on the dataset by specifying the "center"parameter.

The L-BFGS optimizer, specified by the value 'lbfgs' for the parameter "optimizer", uses a backtracking line search algorithm to minimize a function. The following parameters are used by L-BFGS: "max_iterations", "tolerance"(the optimization is terminated when the gradient norm is below this value). For more details on the L-BFGS optimizer, consult either the mlpack L-BFGS documentation (in lbfgs.hpp) or the vast set of published literature on L-BFGS. In addition, a normalized starting point can be used by specifying the "normalize" parameter.

By default, the AMSGrad optimizer is used.

Value

A list with several components defining the class attributes:

`centered_data`	Output matrix for mean-centered dataset (numeric matrix).
`output`	Output matrix for learned distance matrix (numeric matrix).
`transformed_data`	Output matrix for transformed dataset (numeric matrix).

Author(s)

mlpack developers

Examples

```
# Example - Let's say we want to learn distance on iris dataset with number
# of targets as 3 using BigBatch_SGD optimizer. A simple call for the same
# will look like:
#
# \dontrun{
# output <- lmnn(input=iris, labels=iris_labels, k=3, optimizer="bbsgd")
# output <- output$output
# }
#
# Another program call making use of update interval & regularization
# parameter with dataset having labels as last column can be made as:
#
# \dontrun{
# output <- lmnn(input=letter_recognition, k=5, update_interval=10,
#   regularization=0.4)
# output <- output$output
# }
```

---

local_coordinate_coding *Local Coordinate Coding*

Description

An implementation of Local Coordinate Coding (LCC), a data transformation technique. Given input data, this transforms each point to be expressed as a linear combination of a few points in the dataset; once an LCC model is trained, it can be used to transform points later also.

Usage

```
local_coordinate_coding(
  atoms = 0,
  initial_dictionary = NA,
  input_model = NA,
  lambda = 0,
  max_iterations = 0,
  normalize = FALSE,
  seed = 0,
  test = NA,
  tolerance = 0.01,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

atoms	Number of atoms in the dictionary. Default value "0" (integer).
initial_dictionary	Optional initial dictionary (numeric matrix).
input_model	Input LCC model (LocalCoordinateCoding).
lambda	Weighted l1-norm regularization parameter. Default value "0" (numeric).
max_iterations	Maximum number of iterations for LCC (0 indicates no limit). Default value "0" (integer).
normalize	If set, the input data matrix will be normalized before coding. Default value "FALSE" (logical).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
test	Test points to encode (numeric matrix).
tolerance	Tolerance for objective function. Default value "0.01" (numeric).
training	Matrix of training data (X) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of Local Coordinate Coding (LCC), which codes data that approximately lives on a manifold using a variation of l1-norm regularized sparse coding. Given a dense data matrix X with n points and d dimensions, LCC seeks to find a dense dictionary matrix D with k atoms in d dimensions, and a coding matrix Z with n points in k dimensions. Because of the regularization method used, the atoms in D should lie close to the manifold on which the data points lie.

The original data matrix X can then be reconstructed as D * Z. Therefore, this program finds a representation of each point in X as a sparse linear combination of atoms in the dictionary D.

The coding is found with an algorithm which alternates between a dictionary step, which updates the dictionary D, and a coding step, which updates the coding matrix Z.

To run this program, the input matrix X must be specified (with -i), along with the number of atoms in the dictionary (-k). An initial dictionary may also be specified with the "initial_dictionary" parameter. The l1-norm regularization parameter is specified with the "lambda" parameter.

Value

A list with several components defining the class attributes:

codes	Output codes matrix (numeric matrix).
dictionary	Output dictionary matrix (numeric matrix).
output_model	Output for trained LCC model (LocalCoordinateCoding).

Author(s)

mlpack developers

Examples

```
# For example, to run LCC on the dataset "data" using 200 atoms and an
# l1-regularization parameter of 0.1, saving the dictionary "dictionary" and
# the codes into "codes", use
#
# \dontrun{
# output <- local_coordinate_coding(training=data, atoms=200, lambda=0.1)
# dict <- output$dictionary
# codes <- output$codes
# }
#
# The maximum number of iterations may be specified with the "max_iterations"
# parameter. Optionally, the input data matrix X can be normalized before
# coding with the "normalize" parameter.
#
# An LCC model may be saved using the "output_model" output parameter.  Then,
# to encode new points from the dataset "points" with the previously saved
# model "lcc_model", saving the new codes to "new_codes", the following
# command can be used:
#
# \dontrun{
# output <- local_coordinate_coding(input_model=lcc_model, test=points)
# new_codes <- output$codes
# }
```

---

logistic_regression *L2-regularized Logistic Regression and Prediction*

Description

An implementation of L2-regularized logistic regression for two-class classification. Given labeled data, a model can be trained and saved for future use; or, a pre-trained model can be used to classify new points.

Usage

```
logistic_regression(
  batch_size = 64,
  decision_boundary = 0.5,
  input_model = NA,
  labels = NA,
  lambda = 0,
  max_iterations = 10000,
  optimizer = "lbfgs",
  print_training_accuracy = FALSE,
  step_size = 0.01,
  test = NA,
  tolerance = 1e-10,
```

```
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

batch_size	Batch size for SGD. Default value "64" (integer).
decision_boundary	Decision boundary for prediction; if the logistic function for a point is less than the boundary, the class is taken to be 0; otherwise, the class is 1. Default value "0.5" (numeric).
input_model	Existing model (parameters) (LogisticRegression).
labels	A matrix containing labels (0 or 1) for the points in the training set (y) (integer row).
lambda	L2-regularization parameter for training. Default value "0" (numeric).
max_iterations	Maximum iterations for optimizer (0 indicates no limit). Default value "10000" (integer).
optimizer	Optimizer to use for training ('lbfgs' or 'sgd'). Default value "lbfgs" (character).
print_training_accuracy	If set, then the accuracy of the model on the training set will be printed (verbose must also be specified). Default value "FALSE" (logical).
step_size	Step size for SGD optimizer. Default value "0.01" (numeric).
test	Matrix containing test dataset (numeric matrix).
tolerance	Convergence tolerance for optimizer. Default value "1e-10" (numeric).
training	A matrix containing the training set (the matrix of predictors, X) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of L2-regularized logistic regression using either the L-BFGS optimizer or SGD (stochastic gradient descent). This solves the regression problem

y = (1 / 1 + e^-(X * b)).

In this setting, y corresponds to class labels and X corresponds to data.

This program allows loading a logistic regression model (via the "input_model" parameter) or training a logistic regression model given training data (specified with the "training" parameter), or both those things at once. In addition, this program allows classification on a test dataset (specified with the "test" parameter) and the classification results may be saved with the "predictions" output parameter. The trained logistic regression model may be saved using the "output_model" output parameter.

The training data, if specified, may have class labels as its last dimension. Alternately, the "labels" parameter may be used to specify a separate matrix of labels.

When a model is being trained, there are many options. L2 regularization (to prevent overfitting) can be specified with the "lambda" option, and the optimizer used to train the model can be specified with the "optimizer" parameter. Available options are 'sgd' (stochastic gradient descent) and 'lbfgs' (the L-BFGS optimizer). There are also various parameters for the optimizer; the "max_iterations" parameter specifies the maximum number of allowed iterations, and the "tolerance" parameter specifies the tolerance for convergence. For the SGD optimizer, the "step_size" parameter controls the step size taken at each iteration by the optimizer. The batch size for SGD is controlled with the "batch_size" parameter. If the objective function for your data is oscillating between Inf and 0, the step size is probably too large. There are more parameters for the optimizers, but the C++ interface must be used to access these.

For SGD, an iteration refers to a single point. So to take a single pass over the dataset with SGD, "max_iterations" should be set to the number of points in the dataset.

Optionally, the model can be used to predict the responses for another matrix of data points, if "test" is specified. The "test" parameter can be specified without the "training" parameter, so long as an existing logistic regression model is given with the "input_model" parameter. The output predictions from the logistic regression model may be saved with the "predictions" parameter.

This implementation of logistic regression does not support the general multi-class case but instead only the two-class case. Any labels must be either 0 or 1. For more classes, see the softmax regression implementation.

Value

A list with several components defining the class attributes:

`output_model`	Output for trained logistic regression model (LogisticRegression).
`predictions`	If test data is specified, this matrix is where the predictions for the test set will be saved (integer row).
`probabilities`	If test data is specified, this matrix is where the class probabilities for the test set will be saved (numeric matrix).

Author(s)

mlpack developers

Examples

```
# As an example, to train a logistic regression model on the data '"data"'
# with labels '"labels"' with L2 regularization of 0.1, saving the model to
# '"lr_model"', the following command may be used:
#
# \dontrun{
# output <- logistic_regression(training=data, labels=labels, lambda=0.1,
#   print_training_accuracy=TRUE)
# lr_model <- output$output_model
# }
#
# Then, to use that model to predict classes for the dataset '"test"',
# storing the output predictions in '"predictions"', the following command
# may be used:
```

```
#
# \dontrun{
# output <- logistic_regression(input_model=lr_model, test=test)
# predictions <- output$predictions
# }
```

`logistic_regression_probabilities` *L2-regularized Logistic Regression Probabilities*

Description

An implementation of L2-regularized logistic regression for two-class classification. Uses a trained model to classify new points and provide classification probabilities.

Usage

```
logistic_regression_probabilities(
  input_model,
  test,
  decision_boundary = 0.5,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Existing model (parameters) (LogisticRegression).
`test`	Matrix containing test dataset (numeric matrix).
`decision_boundary`	Decision boundary for prediction; if the logistic function for a point is less than the boundary, the class is taken to be 0; otherwise, the class is 1. Default value "0.5" (numeric).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

`probabilities`	Predicted class probabilities for each point in the test set (numeric matrix).

Author(s)

mlpack developers

Examples

```
# \dontrun{ prob <- predict(model, newdata=X_test, type="probabilities") }
```

`logistic_regression_train` *L2-regularized Logistic Regression Training and Prediction*

Description

An implementation of L2-regularized logistic regression for two-class classification. Given labeled data, a model is trained and saved for future use; or, a pre-trained model can be used to classify new points.

Usage

```
logistic_regression_train(
  training,
  batch_size = 64,
  labels = NA,
  lambda = 0,
  max_iterations = 10000,
  optimizer = "lbfgs",
  print_training_accuracy = FALSE,
  step_size = 0.01,
  tolerance = 1e-10,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`training`	A matrix containing the training set (the matrix of predictors, X) (numeric matrix).
`batch_size`	Batch size for SGD. Default value "64" (integer).
`labels`	A matrix containing labels (0 or 1) for the points in the training set (y) (integer row).
`lambda`	L2-regularization parameter for training. Default value "0" (numeric).
`max_iterations`	Maximum iterations for optimizer (0 indicates no limit). Default value "10000" (integer).
`optimizer`	Optimizer to use for training ('lbfgs' or 'sgd'). Default value "lbfgs" (character).
`print_training_accuracy`	If set, then the accuracy of the model on the training set will be printed (verbose must also be specified). Default value "FALSE" (logical).
`step_size`	Step size for SGD optimizer. Default value "0.01" (numeric).
`tolerance`	Convergence tolerance for optimizer. Default value "1e-10" (numeric).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of L2-regularized logistic regression using either the L-BFGS optimizer or SGD (stochastic gradient descent). This solves the regression problem

y = (1 / 1 + e^-(X * b)).

In this setting, y corresponds to class labels and X corresponds to data.

This implementation can train a logistic regression model given training data (specified with the "training" parameter). A trained logistic regression model can then be used to perform classification on a test dataset (specified with the "test" parameter). Alternatively, classification probabilities can be computed and saved with the "probabilities" parameter.

The training data, if specified, may have class labels as its last dimension. Alternately, the "labels" parameter may be used to specify a separate matrix of labels.

When a model is being trained, there are many options. L2 regularization (to prevent overfitting) can be specified with the "lambda" option, and the optimizer used to train the model can be specified with the "optimizer" parameter. Available options are 'sgd' (stochastic gradient descent) and 'lbfgs' (the L-BFGS optimizer). There are also various parameters for the optimizer; the "max_iterations" parameter specifies the maximum number of allowed iterations, and the "tolerance" parameter specifies the tolerance for convergence. For the SGD optimizer, the "step_size" parameter controls the step size taken at each iteration by the optimizer. The batch size for SGD is controlled with the "batch_size" parameter. If the objective function for your data is oscillating between Inf and 0, the step size is probably too large. There are more parameters for the optimizers, but the C++ interface must be used to access these.

For SGD, an iteration refers to a single point. So to take a single pass over the dataset with SGD, "max_iterations" should be set to the number of points in the dataset.

This implementation of logistic regression does not support the general multi-class case but instead only the two-class case. Any labels must be either 0 or 1. For more classes, see the softmax regression implementation.

Value

A list with several components defining the class attributes:

`output_model` Output for trained logistic regression model (LogisticRegression).

Author(s)

mlpack developers

Examples

```
#
# #' # \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <- as.matrix(read.csv("http://datasets.mlpack.org/iris.csv",
# header=FALSE))
# y <- as.matrix(read.csv("http://datasets.mlpack.org/iris_labels.csv",
# header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
```

```
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- logistic_regression_train(training=X_train, labels=y_train,
#   lambda=0.1)
#   }
```

---

lsh *K-Approximate-Nearest-Neighbor Search with LSH*

Description

An implementation of approximate k-nearest-neighbor search with locality-sensitive hashing (LSH). Given a set of reference points and a set of query points, this will compute the k approximate nearest neighbors of each query point in the reference set; models can be saved for future use.

Usage

```
lsh(
  bucket_size = 500,
  hash_width = 0,
  input_model = NA,
  k = 0,
  num_probes = 0,
  projections = 10,
  query = NA,
  reference = NA,
  second_hash_size = 99901,
  seed = 0,
  tables = 30,
  true_neighbors = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

bucket_size	The size of a bucket in the second level hash. Default value "500" (integer).
hash_width	The hash width for the first-level hashing in the LSH preprocessing. By default, the LSH class automatically estimates a hash width for its use. Default value "0" (numeric).
input_model	Input LSH model (LSHSearch).
k	Number of nearest neighbors to find. Default value "0" (integer).

num_probes	Number of additional probes for multiprobe LSH; if 0, traditional LSH is used. Default value "0" (integer).
projections	The number of hash functions for each tabl. Default value "10" (integer).
query	Matrix containing query points (optional) (numeric matrix).
reference	Matrix containing the reference dataset (numeric matrix).
second_hash_size	The size of the second level hash table. Default value "99901" (integer).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
tables	The number of hash tables to be used. Default value "30" (integer).
true_neighbors	Matrix of true neighbors to compute recall with (the recall is printed when -v is specified) (integer matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program will calculate the k approximate-nearest-neighbors of a set of points using locality-sensitive hashing. You may specify a separate set of reference points and query points, or just a reference set which will be used as both the reference and query set.

Value

A list with several components defining the class attributes:

distances	Matrix to output distances into (numeric matrix).
neighbors	Matrix to output neighbors into (integer matrix).
output_model	Output for trained LSH model (LSHSearch).

Author(s)

mlpack developers

Examples

```
# For example, the following will return 5 neighbors from the data for each
# point in "input" and store the distances in "distances" and the neighbors
# in "neighbors":
#
# \dontrun{
# output <- lsh(k=5, reference=input)
# distances <- output$distances
# neighbors <- output$neighbors
# }
# #' #
# The output is organized such that row i and column j in the neighbors
# output corresponds to the index of the point in the reference set which is
# the j'th nearest neighbor from the point in the query set with index i.
```

```
# Row j and column i in the distances output file corresponds to the distance
# between those two points.
#
# Because this is approximate-nearest-neighbors search, results may be
# different from run to run.  Thus, the "seed" parameter can be specified to
# set the random seed.
#
# This program also has many other parameters to control its functionality;
# see the parameter-specific documentation for more information.
```

`mean_shift` *Mean Shift Clustering*

Description

A fast implementation of mean-shift clustering using dual-tree range search. Given a dataset, this uses the mean shift algorithm to produce and return a clustering of the data.

Usage

```
mean_shift(
  input,
  force_convergence = FALSE,
  in_place = FALSE,
  labels_only = FALSE,
  max_iterations = 1000,
  radius = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Input dataset to perform clustering on (numeric matrix).
`force_convergence`	If specified, the mean shift algorithm will continue running regardless of max_iterations until the clusters converge. Default value "FALSE" (logical).
`in_place`	If specified, a column containing the learned cluster assignments will be added to the input dataset file. In this case, –output_file is overridden. (Do not use with Python.. Default value "FALSE" (logical).
`labels_only`	If specified, only the output labels will be written to the file specified by –output_file. Default value "FALSE" (logical).
`max_iterations`	Maximum number of iterations before mean shift terminates. Default value "1000" (integer).
`radius`	If the distance between two centroids is less than the given radius, one will be removed. A radius of 0 or less means an estimate will be calculated and used for the radius. Default value "0" (numeric).

verbose — Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs mean shift clustering on the given dataset, storing the learned cluster assignments either as a column of labels in the input dataset or separately.

The input dataset should be specified with the "input" parameter, and the radius used for search can be specified with the "radius" parameter. The maximum number of iterations before algorithm termination is controlled with the "max_iterations" parameter.

The output labels may be saved with the "output" output parameter and the centroids of each cluster may be saved with the "centroid" output parameter.

Value

A list with several components defining the class attributes:

centroid — If specified, the centroids of each cluster will be written to the given matrix (numeric matrix).

output — Matrix to write output labels or labeled data to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to run mean shift clustering on the dataset "data" and store
# the centroids to "centroids", the following command may be used:
#
# \dontrun{
# output <- mean_shift(input=data)
# centroids <- output$centroid
# }
```

---

mlpack *mlpack*

Description

mlpack is a fast, flexible machine learning library, written in C++, that aims to provide fast, extensible implementations of cutting-edge machine learning algorithms. mlpack provides these algorithms as simple command-line programs, C++ classes and bindings for : Python, Julia, Go and R which can then be integrated into larger-scale machine learning solutions.

Author(s)

Maintainer: Ryan Curtin <ryan@ratml.org> [contributor, copyright holder]

Authors:

- Ryan Curtin <ryan@ratml.org> [contributor, copyright holder]
- Yashwant Singh Parihar <yashwantsingh.sngh@gmail.com> [contributor, copyright holder]
- Dirk Eddelbuettel <edd@debian.org> [contributor, copyright holder]
- James Balamuta <james.balamuta@gmail.com> [contributor, copyright holder]

Other contributors:

- Bill March <march@gatech.edu> [contributor, copyright holder]
- Dongryeol Lee <dongryel@cc.gatech.edu> [contributor, copyright holder]
- Nishant Mehta <niche@cc.gatech.edu> [contributor, copyright holder]
- Parikshit Ram <p.ram@gatech.edu> [contributor, copyright holder]
- James Cline <james.cline@gatech.edu> [contributor, copyright holder]
- Sterling Peet <sterling.peet@gatech.edu> [contributor, copyright holder]
- Matthew Amidon <mamidon@gatech.edu> [contributor, copyright holder]
- Neil Slagle <npslagle@gmail.com> [contributor, copyright holder]
- Ajinkya Kale <kaleajinkya@gmail.com> [contributor, copyright holder]
- Vlad Grantcharov <vlad321@gatech.edu> [contributor, copyright holder]
- Noah Kauffman <notoriousnoah@gmail.com> [contributor, copyright holder]
- Rajendran Mohan <rmohan88@gatech.edu> [contributor, copyright holder]
- Trironk Kiatkungwanglai <trironk@gmail.com> [contributor, copyright holder]
- Patrick Mason <patrick.s.mason@gmail.com> [contributor, copyright holder]
- Marcus Edel <marcus.edel@fu-berlin.de> [contributor, copyright holder]
- Mudit Raj Gupta <mudit.raaj.gupta@gmail.com> [contributor, copyright holder]
- Sumedh Ghaisas <sumedhghaisas@gmail.com> [contributor, copyright holder]
- Michael Fox <michaelfox99@gmail.com> [contributor, copyright holder]
- Siddharth Agrawal <siddharth.950@gmail.com> [contributor, copyright holder]
- Saheb Motiani <saheb210692@gmail.com> [contributor, copyright holder]
- Yash Vadalia <yashdv@gmail.com> [contributor, copyright holder]
- Abhishek Laddha <laddhaabhishek11@gmail.com> [contributor, copyright holder]
- Vahab Akbarzadeh <v.akbarzadeh@gmail.com> [contributor, copyright holder]
- Andrew Wells <andrewmw94@gmail.com> [contributor, copyright holder]
- Zhihao Lou <lzh1984@gmail.com> [contributor, copyright holder]
- Udit Saxena <saxenda.udit@gmail.com> [contributor, copyright holder]
- Stephen Tu <tu.stephenl@gmail.com> [contributor, copyright holder]
- Jaskaran Singh <jaskaranvirdi@ymail.com> [contributor, copyright holder]

- Hritik Jain <hritik.jain.cse13@itbhu.ac.in> [contributor, copyright holder]
- Vladimir Glazachev <glazachev.vladimir@gmail.com> [contributor, copyright holder]
- QiaoAn Chen <kazenoyumechen@gmail.com> [contributor, copyright holder]
- Janzen Brewer <jahabrewer@gmail.com> [contributor, copyright holder]
- Trung Dinh <dinhanhtrung@gmail.com> [contributor, copyright holder]
- Tham Ngap Wei <thamngapwei@gmail.com> [contributor, copyright holder]
- Grzegorz Krajewski <krajekg@gmail.com> [contributor, copyright holder]
- Joseph Mariadassou <joe.mariadassou@gmail.com> [contributor, copyright holder]
- Pavel Zhigulin <pashaworking@gmail.com> [contributor, copyright holder]
- Andy Fang <AndyFang.DZ@gmail.com> [contributor, copyright holder]
- Barak Pearlmutter <barak+git@pearlmutter.net> [contributor, copyright holder]
- Ivari Horm <ivari@risk.ee> [contributor, copyright holder]
- Dhawal Arora <d.p.arora1@gmail.com> [contributor, copyright holder]
- Alexander Leinoff <alexander-leinoff@uiowa.edu> [contributor, copyright holder]
- Palash Ahuja <abhor902@gmail.com> [contributor, copyright holder]
- Yannis Mentekidis <mentekid@gmail.com> [contributor, copyright holder]
- Ranjan Mondal <ranjan.rev@gmail.com> [contributor, copyright holder]
- Mikhail Lozhnikov <lozhnikovma@gmail.com> [contributor, copyright holder]
- Marcos Pividori <marcos.pividori@gmail.com> [contributor, copyright holder]
- Keon Kim <kwk236@gmail.com> [contributor, copyright holder]
- Nilay Jain <nilayjain13@gmail.com> [contributor, copyright holder]
- Peter Lehner <peter.lehner@dlr.de> [contributor, copyright holder]
- Anuraj Kanodia <akanuraj200@gmail.com> [contributor, copyright holder]
- Ivan Georgiev <ivan@jonan.info> [contributor, copyright holder]
- Shikhar Bhardwaj <shikharbhardwaj68@gmail.com> [contributor, copyright holder]
- Yashu Seth <yashuseth2503@gmail.com> [contributor, copyright holder]
- Mike Izbicki <mike@izbicki.me> [contributor, copyright holder]
- Sudhanshu Ranjan <sranjan.sud@gmail.com> [contributor, copyright holder]
- Piyush Jaiswal <piyush.jaiswal@st.niituniversity.in> [contributor, copyright holder]
- Dinesh Raj <dinu.iota@gmail.com> [contributor, copyright holder]
- Vivek Pal <vivekpal.dtu@gmail.com> [contributor, copyright holder]
- Prasanna Patil <prasannapatil08@gmail.com> [contributor, copyright holder]
- Lakshya Agrawal <zeeshan.lakshya@gmail.com> [contributor, copyright holder]
- Praveen Ch <chvsp972911@gmail.com> [contributor, copyright holder]
- Kirill Mishchenko <ki.mishchenko@gmail.com> [contributor, copyright holder]
- Abhinav Moudgil <abhinavmoudgil95@gmail.com> [contributor, copyright holder]
- Thyrix Yang <thyrixyang@gmail.com> [contributor, copyright holder]

- Sagar B Hathwar <sagarbhathwar@gmail.com> [contributor, copyright holder]
- Nishanth Hegde <hegde.nishanth@gmail.com> [contributor, copyright holder]
- Parminder Singh <parmsingh101@gmail.com> [contributor, copyright holder]
- CodeAi <benjamin.bales@assrc.us> [contributor, copyright holder]
- Franciszek Stokowacki <franek.stokowacki@gmail.com> [contributor, copyright holder]
- Samikshya Chand <samikshya289@gmail.com> [contributor, copyright holder]
- N Rajiv Vaidyanathan <rajivvaidyanathan4@gmail.com> [contributor, copyright holder]
- Kartik Nighania <kartiknighania@gmail.com> [contributor, copyright holder]
- Eugene Freyman <evg.freyman@gmail.com> [contributor, copyright holder]
- Manish Kumar <manish887kr@gmail.com> [contributor, copyright holder]
- Haritha Sreedharan Nair <haritha1313@gmail.com> [contributor, copyright holder]
- Sourabh Varshney <sourabhvarshney111@gmail.com> [contributor, copyright holder]
- Projyal Dev <projyal@gmail.com> [contributor, copyright holder]
- Nikhil Goel <nikhilgoel199797@gmail.com> [contributor, copyright holder]
- Shikhar Jaiswal <jaiswalshikhar87@gmail.com> [contributor, copyright holder]
- B Kartheek Reddy <bkartheekreddy@gmail.com> [contributor, copyright holder]
- Atharva Khandait <akhandait45@gmail.com> [contributor, copyright holder]
- Wenhao Huang <wenhao.huang.work@gmail.com> [contributor, copyright holder]
- Roberto Hueso <robertohueso96@gmail.com> [contributor, copyright holder]
- Prabhat Sharma <prabhatsharma7298@gmail.com> [contributor, copyright holder]
- Tan Jun An <yamidarkxxx@gmail.com> [contributor, copyright holder]
- Moksh Jain <mokshjn00@gmail.com> [contributor, copyright holder]
- Manthan-R-Sheth <manthanrsheth96@gmail.com> [contributor, copyright holder]
- Namrata Mukhija <namratamukhija@gmail.com> [contributor, copyright holder]
- Conrad Sanderson [contributor, copyright holder]
- Thanasis Mattas <mattasa@auth.gr> [contributor, copyright holder]
- Shashank Shekhar <contactshashankshekhar@gmail.com> [contributor, copyright holder]
- Yasmine Dumouchel <yasmine.dumouchel@gmail.com> [contributor, copyright holder]
- German Lancioni [contributor, copyright holder]
- Arash Abghari <arash.abghari@gmail.com> [contributor, copyright holder]
- Ayush Chamoli [contributor, copyright holder]
- Tommi Laivamaa <tommi.laivamaa@protonmail.com> [contributor, copyright holder]
- Kim SangYeon <sy0814k@gmail.com> [contributor, copyright holder]
- Niteya Shah <niteya.56@gmail.com> [contributor, copyright holder]
- Toshal Agrawal <tagrawal1339@gmail.com> [contributor, copyright holder]
- Dan Timson [contributor, copyright holder]
- Miguel Canteras <mcanteras@gmail.com> [contributor, copyright holder]

- Bishwa Karki <karkeebishwa1@gmail.com> [contributor, copyright holder]
- Mehul Kumar Nirala <mehulkumarnirala@gmail.com> [contributor, copyright holder]
- Heet Sankesara <heetsankesara3@gmail.com> [contributor, copyright holder]
- Jeffin Sam <sam.jeffin@gmail.com> [contributor, copyright holder]
- Vikas S Shetty <shettyvikas209@gmail.com> [contributor, copyright holder]
- Khizir Siddiqui <khizirsiddiqui@gmail.com> [contributor, copyright holder]
- Tejasvi Tomar <tstomar@outlook.com> [contributor, copyright holder]
- Jai Agarwal <jai.bhageria@gmail.com> [contributor, copyright holder]
- Ziyang Jiang <zij004@alumni.stanford.edu> [contributor, copyright holder]
- Rohit Kartik <rohit.audrey@gmail.com> [contributor, copyright holder]
- Aditya Viki <adityaviki01@gmail.com> [contributor, copyright holder]
- Kartik Dutt <kartikdutt@live.in> [contributor, copyright holder]
- Suryoday Basak <suryodaybasak@gmail.com> [contributor, copyright holder]
- Sriram S K <sriramsk1999@gmail.com> [contributor, copyright holder]
- Manoranjan Kumar Bharti (Nakul Bharti) <knakul853@gmail.com> [contributor, copyright holder]
- Saraansh Tandon <saraanshtandon1999@gmail.com> [contributor, copyright holder]
- Gaurav Singh <gs8763076@gmail.com> [contributor, copyright holder]
- Lakshya Ojha <ojhalakshya@gmail.com> [contributor, copyright holder]
- Bisakh Mondal <bisakhmondal00@gmail.com> [contributor, copyright holder]
- Benson Muite <benson_muite@emailplus.org> [contributor, copyright holder]
- Sarthak Bhardwaj <7sarthakbhardwaj@gmail.com> [contributor, copyright holder]
- Aakash Kaushik <kaushikaakash7539@gmail.com> [contributor, copyright holder]
- Anush Kini <anushkini@gmail.com> [contributor, copyright holder]
- Nippun Sharma <inbox.nippun@gmail.com> [contributor, copyright holder]
- Rishabh Garg <rishabhgarg108@gmail.com> [contributor, copyright holder]
- Sudhakar Brar <dxhrmhall1449@tutanota.com> [contributor, copyright holder]
- Alex Nguyen <alexvn.edu@gmail.com> [contributor, copyright holder]
- Gaurav Ghati <gauravghatii@gmail.com> [contributor, copyright holder]
- Anmolpreet Singh <anmol323c@gmail.com> [contributor, copyright holder]
- Anjishnu Mukherjee <amukher6@gmu.edu> [contributor, copyright holder]
- Omar Shrit <omar@shrit.me> [contributor, copyright holder]
- Tru Hoang <trugiahoang@gmail.com> [contributor, copyright holder]
- Mark Fischinger <markfischinger@gmail.com> [contributor, copyright holder]
- Muhammad Fawwaz Mayda <maydafawwaz@gmail.com> [contributor, copyright holder]
- Roshan Nrusing Swain <swainroshan001@gmail.com> [contributor, copyright holder]
- Suvarsha Chennareddy <suvarshachennareddy@gmail.com> [contributor, copyright holder]

- Shubham Agrawal <shubham.agra1206@gmail.com> [contributor, copyright holder]
- Sri Madhan M <srimadhan11@gmail.com> [contributor, copyright holder]
- Zhuojin Liu <zhuojinliu.cs@gmail.com> [contributor, copyright holder]
- Richèl Bilderbeek <richel@richelbilderbeek.nl> [contributor, copyright holder]
- Chetan Pandey <chetanpandey1266@gmail.com> [contributor, copyright holder]
- Nikolay Apanasov <nikolay@apanasov.org> [contributor, copyright holder]
- Martin Lambertsen <github@lambertsen.one> [contributor, copyright holder]
- Andrea Novellini <andre.novellini@gmail.com> [contributor, copyright holder]
- Felix Patschkowski <felix.patschkowski@gmail.com> [contributor, copyright holder]
- Benjamin A. Beasley <code@musicinmybrain.net> [contributor, copyright holder]
- Maksym Prots <imaxprots@gmail.com> [contributor, copyright holder]
- Zachary Ng <zachn716@gmail.com> [contributor, copyright holder]
- Ranjodh Singh <ranjodhsingh1729@gmail.com> [contributor, copyright holder]
- Mohammad Mundiwala <mohammadmundiwala@gmail.com> [contributor, copyright holder]

See Also

Useful links:

- https://www.mlpack.org/doc/user/bindings/r.html
- https://github.com/mlpack/mlpack
- Report bugs at https://github.com/mlpack/mlpack/issues

---

nbc *Parametric Naive Bayes Classifier*

Description

An implementation of the Naive Bayes Classifier, used for classification. Given labeled data, an NBC model can be trained and saved, or, a pre-trained model can be used for classification.

Usage

```
nbc(
  incremental_variance = FALSE,
  input_model = NA,
  labels = NA,
  test = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

incremental_variance	The variance of each class will be calculated incrementally. Default value "FALSE" (logical).
input_model	Input Naive Bayes model (NBCModel).
labels	A file containing labels for the training set (integer row).
test	A matrix containing the test set (numeric matrix).
training	A matrix containing the training set (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program trains the Naive Bayes classifier on the given labeled training set, or loads a model from the given model file, and then may use that trained model to classify the points in a given test set.

The training set is specified with the "training" parameter. Labels may be either the last row of the training set, or alternately the "labels" parameter may be specified to pass a separate matrix of labels.

If training is not desired, a pre-existing model may be loaded with the "input_model" parameter.

The "incremental_variance" parameter can be used to force the training to use an incremental algorithm for calculating variance. This is slower, but can help avoid loss of precision in some cases.

If classifying a test set is desired, the test set may be specified with the "test" parameter, and the classifications may be saved with the "predictions"predictions parameter. If saving the trained model is desired, this may be done with the "output_model" output parameter.

Value

A list with several components defining the class attributes:

output_model	File to save trained Naive Bayes model to (NBCModel).
predictions	The matrix in which the predicted labels for the test set will be written (integer row).
probabilities	The matrix in which the predicted probability of labels for the test set will be written (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to train a Naive Bayes classifier on the dataset "data" with
# labels "labels" and save the model to "nbc_model", the following command
# may be used:
#
# \dontrun{
# output <- nbc(training=data, labels=labels)
# nbc_model <- output$output_model
# }
#
# Then, to use "nbc_model" to predict the classes of the dataset "test_set"
# and save the predicted classes to "predictions", the following command may
# be used:
#
# \dontrun{
# output <- nbc(input_model=nbc_model, test=test_set)
# predictions <- output$predictions
# }
```

---

nca *Neighborhood Components Analysis (NCA)*

Description

An implementation of neighborhood components analysis, a distance learning technique that can be used for preprocessing. Given a labeled dataset, this uses NCA, which seeks to improve the k-nearest-neighbor classification, and returns the learned distance metric.

Usage

```
nca(
  input,
  armijo_constant = 1e-04,
  batch_size = 50,
  labels = NA,
  linear_scan = FALSE,
  max_iterations = 5e+05,
  max_line_search_trials = 50,
  max_step = 1e+20,
  min_step = 1e-20,
  normalize = FALSE,
  num_basis = 5,
  optimizer = "sgd",
  seed = 0,
  step_size = 0.01,
  tolerance = 1e-07,
  verbose = getOption("mlpack.verbose", FALSE),
  wolfe = 0.9
)
```

Arguments

input	Input dataset to run NCA on (numeric matrix).
armijo_constant	Armijo constant for L-BFGS. Default value "0.0001" (numeric).
batch_size	Batch size for mini-batch SGD. Default value "50" (integer).
labels	Labels for input dataset (integer row).
linear_scan	Don't shuffle the order in which data points are visited for SGD or mini-batch SGD. Default value "FALSE" (logical).
max_iterations	Maximum number of iterations for SGD or L-BFGS (0 indicates no limit). Default value "500000" (integer).
max_line_search_trials	Maximum number of line search trials for L-BFGS. Default value "50" (integer).
max_step	Maximum step of line search for L-BFGS. Default value "1e+20" (numeric).
min_step	Minimum step of line search for L-BFGS. Default value "1e-20" (numeric).
normalize	Use a normalized starting point for optimization. This is useful for when points are far apart, or when SGD is returning NaN. Default value "FALSE" (logical).
num_basis	Number of memory points to be stored for L-BFGS. Default value "5" (integer).
optimizer	Optimizer to use; 'sgd' or 'lbfgs'. Default value "sgd" (character).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
step_size	Step size for stochastic gradient descent (alpha). Default value "0.01" (numeric).
tolerance	Maximum tolerance for termination of SGD or L-BFGS. Default value "1e-07" (numeric).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
wolfe	Wolfe condition parameter for L-BFGS. Default value "0.9" (numeric).

Details

This program implements Neighborhood Components Analysis, both a linear dimensionality reduction technique and a distance learning technique. The method seeks to improve k-nearest-neighbor classification on a dataset by scaling the dimensions. The method is nonparametric, and does not require a value of k. It works by using stochastic ("soft") neighbor assignments and using optimization techniques over the gradient of the accuracy of the neighbor assignments.

To work, this algorithm needs labeled data. It can be given as the last row of the input dataset (specified with "input"), or alternatively as a separate matrix (specified with "labels").

This implementation of NCA uses stochastic gradient descent, mini-batch stochastic gradient descent, or the L_BFGS optimizer. These optimizers do not guarantee global convergence for a nonconvex objective function (NCA's objective function is nonconvex), so the final results could depend on the random seed or other optimizer parameters.

Stochastic gradient descent, specified by the value 'sgd' for the parameter "optimizer", depends primarily on three parameters: the step size (specified with "step_size"), the batch size (specified

with "batch_size"), and the maximum number of iterations (specified with "max_iterations"). In addition, a normalized starting point can be used by specifying the "normalize" parameter, which is necessary if many warnings of the form 'Denominator of p_i is 0!' are given. Tuning the step size can be a tedious affair. In general, the step size is too large if the objective is not mostly uniformly decreasing, or if zero-valued denominator warnings are being issued. The step size is too small if the objective is changing very slowly. Setting the termination condition can be done easily once a good step size parameter is found; either increase the maximum iterations to a large number and allow SGD to find a minimum, or set the maximum iterations to 0 (allowing infinite iterations) and set the tolerance (specified by "tolerance") to define the maximum allowed difference between objectives for SGD to terminate. Be careful—setting the tolerance instead of the maximum iterations can take a very long time and may actually never converge due to the properties of the SGD optimizer. Note that a single iteration of SGD refers to a single point, so to take a single pass over the dataset, set the value of the "max_iterations" parameter equal to the number of points in the dataset.

The L-BFGS optimizer, specified by the value 'lbfgs' for the parameter "optimizer", uses a backtracking line search algorithm to minimize a function. The following parameters are used by L-BFGS: "num_basis" (specifies the number of memory points used by L-BFGS), "max_iterations", "armijo_constant", "wolfe", "tolerance" (the optimization is terminated when the gradient norm is below this value), "max_line_search_trials", "min_step", and "max_step" (which both refer to the line search routine). For more details on the L-BFGS optimizer, consult either the mlpack L-BFGS documentation (in lbfgs.hpp) or the vast set of published literature on L-BFGS.

By default, the SGD optimizer is used.

Value

A list with several components defining the class attributes:

`output` Output matrix for learned distance matrix (numeric matrix).

Author(s)

mlpack developers

`nmf` *Non-negative Matrix Factorization*

Description

An implementation of non-negative matrix factorization. This can be used to decompose an input dataset into two low-rank non-negative components.

Usage

```
nmf(
  input,
  rank,
  initial_h = NA,
  initial_w = NA,
```

```
  max_iterations = 10000,
  min_residue = 1e-05,
  seed = 0,
  update_rules = "multdist",
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Input dataset to perform NMF on (numeric matrix).
rank	Rank of the factorization (integer).
initial_h	Initial H matrix (numeric matrix).
initial_w	Initial W matrix (numeric matrix).
max_iterations	Number of iterations before NMF terminates (0 runs until convergence. Default value "10000" (integer).
min_residue	The minimum root mean square residue allowed for each iteration, below which the program terminates. Default value "1e-05" (numeric).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
update_rules	Update rules for each iteration; (multdist | multdiv | als). Default value "multdist" (character).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs non-negative matrix factorization on the given dataset, storing the resulting decomposed matrices in the specified files. For an input dataset V, NMF decomposes V into two matrices W and H such that

V = W * H

where all elements in W and H are non-negative. If V is of size (n x m), then W will be of size (n x r) and H will be of size (r x m), where r is the rank of the factorization (specified by the "rank" parameter).

Optionally, the desired update rules for each NMF iteration can be chosen from the following list:

- multdist: multiplicative distance-based update rules (Lee and Seung 1999) - multdiv: multiplicative divergence-based update rules (Lee and Seung 1999) - als: alternating least squares update rules (Paatero and Tapper 1994)

The maximum number of iterations is specified with "max_iterations", and the minimum residue required for algorithm termination is specified with the "min_residue" parameter.

Value

A list with several components defining the class attributes:

h	Matrix to save the calculated H to (numeric matrix).
w	Matrix to save the calculated W to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to run NMF on the input matrix "V" using the 'multdist' update
# rules with a rank-10 decomposition and storing the decomposed matrices into
# "W" and "H", the following command could be used:
#
# \dontrun{
# output <- nmf(input=V, rank=10, update_rules="multdist")
# W <- output$w
# H <- output$h
# }
```

---

pca *Principal Components Analysis*

Description

An implementation of several strategies for principal components analysis (PCA), a common preprocessing step. Given a dataset and a desired new dimensionality, this can reduce the dimensionality of the data using the linear transformation determined by PCA.

Usage

```
pca(
  input,
  decomposition_method = "exact",
  new_dimensionality = 0,
  scale = FALSE,
  var_to_retain = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Input dataset to perform PCA on (numeric matrix).
`decomposition_method`	Method used for the principal components analysis: 'exact', 'randomized', 'randomized-block-krylov', 'quic'. Default value "exact" (character).
`new_dimensionality`	Desired dimensionality of output dataset. If 0, no dimensionality reduction is performed. Default value "0" (integer).
`scale`	If set, the data will be scaled before running PCA, such that the variance of each feature is 1. Default value "FALSE" (logical).

var_to_retain	Amount of variance to retain; should be between 0 and 1. If 1, all variance is retained. Overrides -d. Default value "0" (numeric).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs principal components analysis on the given dataset using the exact, randomized, randomized block Krylov, or QUIC SVD method. It will transform the data onto its principal components, optionally performing dimensionality reduction by ignoring the principal components with the smallest eigenvalues.

Use the "input" parameter to specify the dataset to perform PCA on. A desired new dimensionality can be specified with the "new_dimensionality" parameter, or the desired variance to retain can be specified with the "var_to_retain" parameter. If desired, the dataset can be scaled before running PCA with the "scale" parameter.

Multiple different decomposition techniques can be used. The method to use can be specified with the "decomposition_method" parameter, and it may take the values 'exact', 'randomized', or 'quic'.

Value

A list with several components defining the class attributes:

output	Matrix to save modified dataset to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to reduce the dimensionality of the matrix "data" to 5
# dimensions using randomized SVD for the decomposition, storing the output
# matrix to "data_mod", the following command can be used:
#
# \dontrun{
# data_mod <- pca(input=data, new_dimensionality=5,
#   decomposition_method="randomized")
# }
```

---

perceptron *Perceptron*

Description

An implementation of a perceptron—a single level neural network–=for classification. Given labeled data, a perceptron can be trained and saved for future use; or, a pre-trained perceptron can be used for classification on new points.

Usage

```
perceptron(
  input_model = NA,
  labels = NA,
  max_iterations = 1000,
  test = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input_model	Input perceptron model (PerceptronModel).
labels	A matrix containing labels for the training set (integer row).
max_iterations	The maximum number of iterations the perceptron is to be ru. Default value "1000" (integer).
test	A matrix containing the test set (numeric matrix).
training	A matrix containing the training set (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program implements a perceptron, which is a single level neural network. The perceptron makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The perceptron learning rule is able to converge, given enough iterations (specified using the "max_iterations" parameter), if the data supplied is linearly separable. The perceptron is parameterized by a matrix of weight vectors that denote the numerical weights of the neural network.

This program allows loading a perceptron from a model (via the "input_model" parameter) or training a perceptron given training data (via the "training" parameter), or both those things at once. In addition, this program allows classification on a test dataset (via the "test" parameter) and the classification results on the test set may be saved with the "predictions" output parameter. The perceptron model may be saved with the "output_model" output parameter.

Value

A list with several components defining the class attributes:

output_model	Output for trained perceptron model (PerceptronModel).
predictions	The matrix in which the predicted labels for the test set will be written (integer row).

Author(s)

mlpack developers

Examples

```
# The training data given with the "training" option may have class labels as
# its last dimension (so, if the training data is in CSV format, labels
# should be the last column).  Alternately, the "labels" parameter may be
# used to specify a separate matrix of labels.
#
# All these options make it easy to train a perceptron, and then re-use that
# perceptron for later classification.  The invocation below trains a
# perceptron on "training_data" with labels "training_labels", and saves the
# model to "perceptron_model".
#
# \dontrun{
# output <- perceptron(training=training_data, labels=training_labels)
# perceptron_model <- output$output_model
# }
#
# Then, this model can be re-used for classification on the test data
# "test_data".  The example below does precisely that, saving the predicted
# classes to "predictions".
#
# \dontrun{
# output <- perceptron(input_model=perceptron_model, test=test_data)
# predictions <- output$predictions
# }
# #' #
# Note that all of the options may be specified at once: predictions may be
# calculated right after training a model, and model training can occur even
# if an existing perceptron model is passed with the "input_model" parameter.
#  However, note that the number of classes and the dimensionality of all
# data must match.  So you cannot pass a perceptron model trained on 2
# classes and then re-train with a 4-class dataset.  Similarly, attempting
# classification on a 3-dimensional dataset with a perceptron that has been
# trained on 8 dimensions will cause an error.
```

`predict.mlpack_lars` *LARS Prediction*

Description

An implementation of Least Angle Regression (stagewise/lasso), also known as LARS. This program can use a pre-trained LARS/LASSO/Elastic Net model to output regression predictions from a test set.

Usage

```
## S3 method for class 'mlpack_lars'
predict(object, newdata, ...)

lars_predict(input_model, test, verbose = getOption("mlpack.verbose", FALSE))
```

Arguments

object	An instantiated model object for which prediction is desired
newdata	A test data set
...	Additional optional arguments affecting the prediction
input_model	Trained LARS model to use (LARS).
test	Matrix containing points to regress on (test points) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

predictions	Matrix containing predicted responses (numeric matrix).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

---

predict.mlpack_linear_regression
Linear Regression Prediction

Description

Predictions from model.

Usage

```
## S3 method for class 'mlpack_linear_regression'
predict(object, newdata, ...)

linear_regression_predict(
  input_model,
  test,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

object	An instantiated model object for which prediction is desired
newdata	A test data set
...	Additional optional arguments affecting the prediction
input_model	Existing LinearRegression model to use (LinearRegression).
test	Matrix containing X' (test regressors) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

output_predictions
: Matrix containing predicted responses (numeric row).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

---

predict.mlpack_logistic_regression *L2-regularized Logistic Regression Classification*

Description

An implementation of L2-regularized logistic regression for two-class classification. Uses a trained model to classify new points.

Usage

```
## S3 method for class 'mlpack_logistic_regression'
predict(object, newdata, type = c("predictions", "probabilities"), ...)

logistic_regression_classify(
  input_model,
  test,
  decision_boundary = 0.5,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

object	An instantiated model object for which prediction is desired
newdata	A test data set
type	A character value selection predictions or probabilities
...	Additional optional arguments affecting the prediction
input_model	Existing model (parameters) (LogisticRegression).
test	Matrix containing test dataset (numeric matrix).
decision_boundary	Decision boundary for prediction; if the logistic function for a point is less than the boundary, the class is taken to be 0; otherwise, the class is 1. Default value "0.5" (numeric).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

predictions	If test data is specified, this matrix is where the predictions for the test set will be saved (integer row).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

---

predict.mlpack_random_forest
Random Forests

Description

An implementation of the standard random forest algorithm by Leo Breiman for classification. Given labeled data, a random forest can be trained and saved for future use; or, a pre-trained random forest can be used for classification.

Usage

```
## S3 method for class 'mlpack_random_forest'
predict(object, newdata, type = c("predictions", "probabilities"), ...)

random_forest(
  input_model = NA,
  labels = NA,
  maximum_depth = 0,
  minimum_gain_split = 0,
  minimum_leaf_size = 1,
  num_trees = 10,
  print_training_accuracy = FALSE,
  seed = 0,
  subspace_dim = 0,
  test = NA,
  test_labels = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE),
  warm_start = FALSE
)
```

Arguments

object	An instantiated model object for which prediction is desired
newdata	A test data set
...	Additional optional arguments affecting the prediction
input_model	Pre-trained random forest to use for classification (RandomForestModel).
labels	Labels for training dataset (integer row).
maximum_depth	Maximum depth of the tree (0 means no limit). Default value "0" (integer).
minimum_gain_split	Minimum gain needed to make a split when building a tree. Default value "0" (numeric).
minimum_leaf_size	Minimum number of points in each leaf node. Default value "1" (integer).
num_trees	Number of trees in the random forest. Default value "10" (integer).
print_training_accuracy	If set, then the accuracy of the model on the training set will be predicted (verbose must also be specified). Default value "FALSE" (logical).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
subspace_dim	Dimensionality of random subspace to use for each split. '0' will autoselect the square root of data dimensionality. Default value "0" (integer).
test	Test dataset to produce predictions for (numeric matrix).
test_labels	Test dataset labels, if accuracy calculation is desired (integer row).
training	Training dataset (numeric matrix).

verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
warm_start	If true and passed along with ‘training‘ and ‘input_model‘ then trains more trees on top of existing model. Default value "FALSE" (logical).
type	Type of predictions to return, stddevs will return standard deviation estimations along with point estimations

Details

This program is an implementation of the standard random forest classification algorithm by Leo Breiman. A random forest can be trained and saved for later use, or a random forest may be loaded and predictions or class probabilities for points may be generated.

The training set and associated labels are specified with the "training" and "labels" parameters, respectively. The labels should be in the range ‘[0, num_classes - 1]‘. Optionally, if "labels" is not specified, the labels are assumed to be the last dimension of the training dataset.

When a model is trained, the "output_model" output parameter may be used to save the trained model. A model may be loaded for predictions with the "input_model"parameter. The "input_model" parameter may not be specified when the "training" parameter is specified. The "minimum_leaf_size" parameter specifies the minimum number of training points that must fall into each leaf for it to be split. The "num_trees" controls the number of trees in the random forest. The "minimum_gain_split" parameter controls the minimum required gain for a decision tree node to split. Larger values will force higher-confidence splits. The "maximum_depth" parameter specifies the maximum depth of the tree. The "subspace_dim" parameter is used to control the number of random dimensions chosen for an individual node's split. If "print_training_accuracy" is specified, the calculated accuracy on the training set will be printed.

Test data may be specified with the "test" parameter, and if performance measures are desired for that test set, labels for the test points may be specified with the "test_labels" parameter. Predictions for each test point may be saved via the "predictions"output parameter. Class probabilities for each prediction may be saved with the "probabilities" output parameter.

Value

A list with several components defining the class attributes:

output_model	Model to save trained random forest to (RandomForestModel).
predictions	Predicted classes for each point in the test set (integer row).
probabilities	Predicted class probabilities for each point in the test set (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to train a random forest with a minimum leaf size of 20 using
# 10 trees on the dataset contained in "data"with labels "labels", saving the
# output random forest to "rf_model" and printing the training error, one
```

```
# could call
#
# \dontrun{
# output <- random_forest(training=data, labels=labels, minimum_leaf_size=20,
#   num_trees=10, print_training_accuracy=TRUE)
# rf_model <- output$output_model
# }
#
# Then, to use that model to classify points in "test_set" and print the test
# error given the labels "test_labels" using that model, while saving the
# predictions for each point to "predictions", one could call
#
# \dontrun{
# output <- random_forest(input_model=rf_model, test=test_set,
#   test_labels=test_labels)
# predictions <- output$predictions
# }
```

`preprocess_binarize` *Binarize Data*

Description

A utility to binarize a dataset. Given a dataset, this utility converts each value in the desired dimension(s) to 0 or 1; this can be a useful preprocessing step.

Usage

```
preprocess_binarize(
  input,
  dimension = 0,
  threshold = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Input data matrix (numeric matrix).
`dimension`	Dimension to apply the binarization. If not set, the program will binarize every dimension by default. Default value "0" (integer).
`threshold`	Threshold to be applied for binarization. If not set, the threshold defaults to 0.0. Default value "0" (numeric).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes a dataset and binarizes the variables into either 0 or 1 given threshold. User can apply binarization on a dimension or the whole dataset. The dimension to apply binarization to can be specified using the "dimension" parameter; if left unspecified, every dimension will be binarized. The threshold for binarization can also be specified with the "threshold" parameter; the default threshold is 0.0.

The binarized matrix may be saved with the "output" output parameter.

Value

A list with several components defining the class attributes:

`output`	Matrix in which to save the output (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, if we want to set all variables greater than 5 in the dataset
# "X" to 1 and variables less than or equal to 5.0 to 0, and save the result
# to "Y", we could run
#
# \dontrun{
# Y <- preprocess_binarize(input=X, threshold=5)
# }
#
# But if we want to apply this to only the first (0th) dimension of "X",  we
# could instead run
#
# \dontrun{
# Y <- preprocess_binarize(input=X, threshold=5, dimension=0)
# }
```

---

preprocess_describe *Descriptive Statistics*

Description

A utility for printing descriptive statistics about a dataset. This prints a number of details about a dataset in a tabular format.

Usage

```
preprocess_describe(
  input,
  dimension = 0,
  population = FALSE,
  precision = 4,
  row_major = FALSE,
  verbose = getOption("mlpack.verbose", FALSE),
  width = 8
)
```

Arguments

input	Matrix containing data (numeric matrix).
dimension	Dimension of the data. Use this to specify a dimensio. Default value "0" (integer).
population	If specified, the program will calculate statistics assuming the dataset is the population. By default, the program will assume the dataset as a sample. Default value "FALSE" (logical).
precision	Precision of the output statistics. Default value "4" (integer).
row_major	If specified, the program will calculate statistics across rows, not across columns. (Remember that in mlpack, a column represents a point, so this option is generally not necessary.. Default value "FALSE" (logical).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).
width	Width of the output table. Default value "8" (integer).

Details

This utility takes a dataset and prints out the descriptive statistics of the data. Descriptive statistics is the discipline of quantitatively describing the main features of a collection of information, or the quantitative description itself. The program does not modify the original file, but instead prints out the statistics to the console. The printed result will look like a table.

Optionally, width and precision of the output can be adjusted by a user using the "width" and "precision" parameters. A user can also select a specific dimension to analyze if there are too many dimensions. The "population" parameter can be specified when the dataset should be considered as a population. Otherwise, the dataset will be considered as a sample.

Author(s)

mlpack developers

Examples

```
# So, a simple example where we want to print out statistical facts about the
# dataset "X" using the default settings, we could run
```

```
#
# \dontrun{
# preprocess_describe(input=X, verbose=TRUE)
# }
#
# If we want to customize the width to 10 and precision to 5 and consider the
# dataset as a population, we could run
#
# \dontrun{
# preprocess_describe(input=X, width=10, precision=5, verbose=TRUE)
# }
```

`preprocess_one_hot_encoding` *One Hot Encoding*

Description

A utility to do one-hot encoding on features of dataset.

Usage

```
preprocess_one_hot_encoding(
  input,
  dimensions = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Matrix containing data (numeric matrix/data.frame with info).
`dimensions`	Index of dimensions that need to be one-hot encoded (if unspecified, all categorical dimensions are one-hot encoded) (integer vector).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes a dataset and a vector of indices and does one-hot encoding of the respective features at those indices. Indices represent the IDs of the dimensions to be one-hot encoded.

If no dimensions are specified with "dimensions", then all categorical-type dimensions will be one-hot encoded. Otherwise, only the dimensions given in "dimensions" will be one-hot encoded.

The output matrix with encoded features may be saved with the "output" parameters.

Value

A list with several components defining the class attributes:

output	Matrix to save one-hot encoded features data to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# So, a simple example where we want to encode 1st and 3rd feature from
# dataset "X" into "X_output" would be
#
# \dontrun{
# X_output <- preprocess_one_hot_encoding(input=X, dimensions=1,
# dimensions=3)
# }
```

---

preprocess_scale *Scale Data*

Description

A utility to perform feature scaling on datasets using one of six techniques. Both scaling and inverse scaling are supported, and scalers can be saved and then applied to other datasets.

Usage

```
preprocess_scale(
  input,
  epsilon = 1e-06,
  input_model = NA,
  inverse_scaling = FALSE,
  max_value = 1,
  min_value = 0,
  scaler_method = "standard_scaler",
  seed = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Matrix containing data (numeric matrix).
epsilon	regularization Parameter for pcawhitening, or zcawhitening, should be between -1 to 1. Default value "1e-06" (numeric).
input_model	Input Scaling model (ScalingModel).

inverse_scaling	Inverse Scaling to get original datase. Default value "FALSE" (logical).
max_value	Ending value of range for min_max_scaler. Default value "1" (integer).
min_value	Starting value of range for min_max_scaler. Default value "0" (integer).
scaler_method	method to use for scaling, the default is standard_scaler. Default value "standard_scaler" (character).
seed	Random seed (0 for std::time(NULL)). Default value "0" (integer).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes a dataset and performs feature scaling using one of the six scaler methods namely: 'max_abs_scaler', 'mean_normalization', 'min_max_scaler' ,'standard_scaler', 'pca_whitening' and 'zca_whitening'. The function takes a matrix as "input" and a scaling method type which you can specify using "scaler_method" parameter; the default is standard scaler, and outputs a matrix with scaled feature.

The output scaled feature matrix may be saved with the "output" output parameters.

The model to scale features can be saved using "output_model" and later can be loaded back using"input_model".

Value

A list with several components defining the class attributes:

output	Matrix to save scaled data to (numeric matrix).
output_model	Output scaling model (ScalingModel).

Author(s)

mlpack developers

Examples

```
# So, a simple example where we want to scale the dataset "X" into "X_scaled"
# with  standard_scaler as scaler_method, we could run
#
# \dontrun{
# output <- preprocess_scale(input=X, scaler_method="standard_scaler")
# X_scaled <- output$output
# }
#
# A simple example where we want to whiten the dataset "X" into "X_whitened"
# with  PCA as whitening_method and use 0.01 as regularization parameter, we
# could run
#
# \dontrun{
# output <- preprocess_scale(input=X, scaler_method="pca_whitening",
```

```
#   epsilon=0.01)
# X_scaled <- output$output
# }
#
# You can also retransform the scaled dataset back using"inverse_scaling". An
# example to rescale : "X_scaled" into "X"using the saved model "input_model"
# is:
#
# \dontrun{
# output <- preprocess_scale(input=X_scaled, inverse_scaling=TRUE,
#   input_model=saved)
# X <- output$output
# }
#
# Another simple example where we want to scale the dataset "X" into
# "X_scaled" with  min_max_scaler as scaler method, where scaling range is 1
# to 3 instead of default 0 to 1. We could run
#
# \dontrun{
# output <- preprocess_scale(input=X, scaler_method="min_max_scaler",
#   min_value=1, max_value=3)
# X_scaled <- output$output
# }
```

`preprocess_split` *Split Data*

Description

A utility to split data into a training and testing dataset. This can also split labels according to the same split.

Usage

```
preprocess_split(
  input,
  input_labels = NA,
  no_shuffle = FALSE,
  seed = 0,
  stratify_data = FALSE,
  test_ratio = 0.2,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input`	Matrix containing data (numeric matrix).
`input_labels`	Matrix containing labels (integer matrix).

no_shuffle	Avoid shuffling the data before splitting. Default value "FALSE" (logical).
seed	Random seed (0 for std::time(NULL)). Default value "0" (integer).
stratify_data	Stratify the data according to label. Default value "FALSE" (logical).
test_ratio	Ratio of test set; if not set,the ratio defaults to 0.. Default value "0.2" (numeric).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This utility takes a dataset and optionally labels and splits them into a training set and a test set. Before the split, the points in the dataset are randomly reordered. The percentage of the dataset to be used as the test set can be specified with the "test_ratio" parameter; the default is 0.2 (20

The output training and test matrices may be saved with the "training" and "test" output parameters.

Optionally, labels can also be split along with the data by specifying the "input_labels" parameter. Splitting labels works the same way as splitting the data. The output training and test labels may be saved with the "training_labels" and "test_labels" output parameters, respectively.

Value

A list with several components defining the class attributes:

test	Matrix to save test data to (numeric matrix).
test_labels	Matrix to save test labels to (integer matrix).
training	Matrix to save training data to (numeric matrix).
training_labels	Matrix to save train labels to (integer matrix).

Author(s)

mlpack developers

Examples

```
# So, a simple example where we want to split the dataset "X" into "X_train"
# and "X_test" with 60% of the data in the training set and 40% of the
# dataset in the test set, we could run
#
# \dontrun{
# output <- preprocess_split(input=X, test_ratio=0.4)
# X_train <- output$training
# X_test <- output$test
# }
#
# Also by default the dataset is shuffled and split; you can provide the
# "no_shuffle" option to avoid shuffling the data; an example to avoid
# shuffling of data is:
#
```

```
# \dontrun{
# output <- preprocess_split(input=X, test_ratio=0.4, no_shuffle=TRUE)
# X_train <- output$training
# X_test <- output$test
# }
#
# If we had a dataset "X" and associated labels "y", and we wanted to split
# these into "X_train", "y_train", "X_test", and "y_test", with 30% of the
# data in the test set, we could run
#
# \dontrun{
# output <- preprocess_split(input=X, input_labels=y, test_ratio=0.3)
# X_train <- output$training
# y_train <- output$training_labels
# X_test <- output$test
# y_test <- output$test_labels
# }
# To maintain the ratio of each class in the train and test sets,
# the"stratify_data" option can be used.
#
# \dontrun{
# output <- preprocess_split(input=X, test_ratio=0.4, stratify_data=TRUE)
# X_train <- output$training
# X_test <- output$test
# }
```

`radical` *RADICAL*

Description

An implementation of RADICAL, a method for independent component analysis (ICA). Given a dataset, this can decompose the dataset into an unmixing matrix and an independent component matrix; this can be useful for preprocessing.

Usage

```
radical(
  input,
  angles = 150,
  noise_std_dev = 0.175,
  objective = FALSE,
  replicates = 30,
  seed = 0,
  sweeps = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input	Input dataset for ICA (numeric matrix).
angles	Number of angles to consider in brute-force search during Radical2D. Default value "150" (integer).
noise_std_dev	Standard deviation of Gaussian noise. Default value "0.175" (numeric).
objective	If set, an estimate of the final objective function is printed. Default value "FALSE" (logical).
replicates	Number of Gaussian-perturbed replicates to use (per point) in Radical2D. Default value "30" (integer).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
sweeps	Number of sweeps; each sweep calls Radical2D once for each pair of dimensions. Default value "0" (integer).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of RADICAL, a method for independent component analysis (ICA). Assuming that we have an input matrix X, the goal is to find a square unmixing matrix W such that Y = W * X and the dimensions of Y are independent components. If the algorithm is running particularly slowly, try reducing the number of replicates.

The input matrix to perform ICA on should be specified with the "input" parameter. The output matrix Y may be saved with the "output_ic" output parameter, and the output unmixing matrix W may be saved with the "output_unmixing" output parameter.

Value

A list with several components defining the class attributes:

output_ic	Matrix to save independent components to (numeric matrix).
output_unmixing	Matrix to save unmixing matrix to (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to perform ICA on the matrix "X" with 40 replicates, saving
# the independent components to "ic", the following command may be used:
#
# \dontrun{
# output <- radical(input=X, replicates=40)
# ic <- output$output_ic
# }
```

`random_forest_classify` *Random Forests Prediction*

Description

Class predictions from random forest model.

Usage

```
random_forest_classify(
  input_model,
  test,
  test_labels = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Pre-trained random forest to use for classification (RandomForestModel).
`test`	Test dataset to produce predictions for (numeric matrix).
`test_labels`	Test dataset labels, if accuracy calculation is desired (integer row).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

`predictions`	Predicted classes for each point in the test set (integer row).

Author(s)

mlpack developers

Examples

```
# \dontrun{ pred <- predict(model, newdata=X_test) }
```

`random_forest_probabilities` *Random Forests Probabilities*

Description

Class probabilities from random forest model.

Usage

```
random_forest_probabilities(
  input_model,
  test,
  test_labels = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`input_model`	Pre-trained random forest to use for classification (RandomForestModel).
`test`	Test dataset to produce predictions for (numeric matrix).
`test_labels`	Test dataset labels, if accuracy calculation is desired (integer row).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Value

A list with several components defining the class attributes:

`probabilities`	Predicted class probabilities for each point in the test set (numeric matrix).

Author(s)

mlpack developers

Examples

```
# \dontrun{ prob <- predict(model, newdata=X_test, type="probabilities") }
```

`random_forest_train` *Random Forests train*

Description

An implementation of the standard random forest algorithm by Leo Breiman for classification. Given labeled data, a random forest is trained.

Usage

```
random_forest_train(
  labels,
  training,
  maximum_depth = 0,
  minimum_gain_split = 0,
  minimum_leaf_size = 1,
  num_trees = 10,
  print_training_accuracy = FALSE,
  seed = 0,
  subspace_dim = 0,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

`labels`	Labels for training dataset (integer row).
`training`	Training dataset (numeric matrix).
`maximum_depth`	Maximum depth of the tree (0 means no limit). Default value "0" (integer).
`minimum_gain_split`	Minimum gain needed to make a split when building a tree. Default value "0" (numeric).
`minimum_leaf_size`	Minimum number of points in each leaf node. Default value "1" (integer).
`num_trees`	Number of trees in the random forest. Default value "10" (integer).
`print_training_accuracy`	If set, then the accuracy of the model on the training set will be predicted (verbose must also be specified). Default value "FALSE" (logical).
`seed`	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
`subspace_dim`	Dimensionality of random subspace to use for each split. '0' will autoselect the square root of data dimensionality. Default value "0" (integer).
`verbose`	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program is an implementation of the standard random forest classification algorithm by Leo Breiman. A random forest is trained (and returned for later use for subsequent use where predictions or class probabilities for points may be generated.

The training set and associated labels are specified with the "training" and "labels" parameters, respectively. The labels should be in the range ‘[0, num_classes - 1]‘. Optionally, if "labels" is not specified, the labels are assumed to be the last dimension of the training dataset.

The "minimum_leaf_size" parameter specifies the minimum number of training points that must fall into each leaf for it to be split. The "num_trees" controls the number of trees in the random forest. The "minimum_gain_split" parameter controls the minimum required gain for a decision tree node to split. Larger values will force higher-confidence splits. The "maximum_depth" parameter specifies the maximum depth of the tree. The "subspace_dim" parameter is used to control the number of random dimensions chosen for an individual node's split. If "print_training_accuracy" is specified, the calculated accuracy on the training set will be printed.

Value

A list with several components defining the class attributes:

`output_model` Model to save trained random forest to (RandomForestModel).

Author(s)

mlpack developers

Examples

```
#
# #' # \dontrun{
# suppressMessages(library(mlpack)) # in case 'mlpack' is not yet loaded
# X <- as.matrix(read.csv("http://datasets.mlpack.org/iris.csv",
# header=FALSE))
# y <- as.matrix(read.csv("http://datasets.mlpack.org/iris_labels.csv",
# header=FALSE))
# pp <- preprocess_split(input=X, input_label=as.matrix(1:nrow(X)),
# test_ratio=0.2)
# X_train <- pp[["training"]]
# X_test <- pp[["test"]]
# # labels are indices to operate on both factors or numeric data
# y_train <- y[as.integer(pp[["training_labels"]]), 1]
# y_test <- y[as.integer(pp[["test_labels"]]), 1]
#
# model <- random_forest_train(training=X_train, labels=y_train,
#   minimum_leaf_size=20, num_trees=10, print_training_accuracy=TRUE)
#   }
```

Serialize *Serialize/Unserialize an mlpack model.*

Description

Serialize/Unserialize an mlpack model.

Usage

```
Serialize(model, filename)

Unserialize(filename)
```

Arguments

model	Input model pointer.
filename	Input filename.

Value

For Unserialize, Output model_ptr.

softmax_regression *Softmax Regression*

Description

An implementation of softmax regression for classification, which is a multiclass generalization of logistic regression. Given labeled data, a softmax regression model can be trained and saved for future use, or, a pre-trained softmax regression model can be used for classification of new points.

Usage

```
softmax_regression(
  input_model = NA,
  labels = NA,
  lambda = 1e-04,
  max_iterations = 400,
  no_intercept = FALSE,
  number_of_classes = 0,
  test = NA,
  test_labels = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

input_model	File containing existing model (parameters) (SoftmaxRegression).
labels	A matrix containing labels (0 or 1) for the points in the training set (y). The labels must order as a row (integer row).
lambda	L2-regularization constan. Default value "0.0001" (numeric).
max_iterations	Maximum number of iterations before termination. Default value "400" (integer).
no_intercept	Do not add the intercept term to the model. Default value "FALSE" (logical).
number_of_classes	Number of classes for classification; if unspecified (or 0), the number of classes found in the labels will be used. Default value "0" (integer).
test	Matrix containing test dataset (numeric matrix).
test_labels	Matrix containing test labels (integer row).
training	A matrix containing the training set (the matrix of predictors, X) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

This program performs softmax regression, a generalization of logistic regression to the multiclass case, and has support for L2 regularization. The program is able to train a model, load an existing model, and give predictions (and optionally their accuracy) for test data.

Training a softmax regression model is done by giving a file of training points with the "training" parameter and their corresponding labels with the "labels" parameter. The number of classes can be manually specified with the "number_of_classes" parameter, and the maximum number of iterations of the L-BFGS optimizer can be specified with the "max_iterations" parameter. The L2 regularization constant can be specified with the "lambda" parameter and if an intercept term is not desired in the model, the "no_intercept" parameter can be specified.

The trained model can be saved with the "output_model" output parameter. If training is not desired, but only testing is, a model can be loaded with the "input_model" parameter. At the current time, a loaded model cannot be trained further, so specifying both "input_model" and "training" is not allowed.

The program is also able to evaluate a model on test data. A test dataset can be specified with the "test" parameter. Class predictions can be saved with the "predictions" output parameter. If labels are specified for the test data with the "test_labels" parameter, then the program will print the accuracy of the predictions on the given test set and its corresponding labels.

Value

A list with several components defining the class attributes:

output_model	File to save trained softmax regression model to (SoftmaxRegression).
predictions	Matrix to save predictions for test dataset into (integer row).
probabilities	Matrix to save class probabilities for test dataset into (numeric matrix).

Author(s)

mlpack developers

Examples

```
# For example, to train a softmax regression model on the data "dataset" with
# labels "labels" with a maximum of 1000 iterations for training, saving the
# trained model to "sr_model", the following command can be used:
#
# \dontrun{
# output <- softmax_regression(training=dataset, labels=labels)
# sr_model <- output$output_model
# }
#
# Then, to use "sr_model" to classify the test points in "test_points",
# saving the output predictions to "predictions", the following command can
# be used:
#
# \dontrun{
# output <- softmax_regression(input_model=sr_model, test=test_points)
# predictions <- output$predictions
# }
```

---

sparse_coding *Sparse Coding*

Description

An implementation of Sparse Coding with Dictionary Learning. Given a dataset, this will decompose the dataset into a sparse combination of a few dictionary elements, where the dictionary is learned during computation; a dictionary can be reused for future sparse coding of new points.

Usage

```
sparse_coding(
  atoms = 15,
  initial_dictionary = NA,
  input_model = NA,
  lambda1 = 0,
  lambda2 = 0,
  max_iterations = 0,
  newton_tolerance = 1e-06,
  normalize = FALSE,
  objective_tolerance = 0.01,
  seed = 0,
  test = NA,
  training = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

atoms	Number of atoms in the dictionary. Default value "15" (integer).
initial_dictionary	Optional initial dictionary matrix (numeric matrix).
input_model	File containing input sparse coding model (SparseCoding).
lambda1	Sparse coding l1-norm regularization parameter. Default value "0" (numeric).
lambda2	Sparse coding l2-norm regularization parameter. Default value "0" (numeric).
max_iterations	Maximum number of iterations for sparse coding (0 indicates no limit). Default value "0" (integer).
newton_tolerance	Tolerance for convergence of Newton method. Default value "1e-06" (numeric).
normalize	If set, the input data matrix will be normalized before coding. Default value "FALSE" (logical).
objective_tolerance	Tolerance for convergence of the objective function. Default value "0.01" (numeric).
seed	Random seed. If 0, 'std::time(NULL)' is used. Default value "0" (integer).
test	Optional matrix to be encoded by trained model (numeric matrix).
training	Matrix of training data (X) (numeric matrix).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

An implementation of Sparse Coding with Dictionary Learning, which achieves sparsity via an l1-norm regularizer on the codes (LASSO) or an (l1+l2)-norm regularizer on the codes (the Elastic Net). Given a dense data matrix X with d dimensions and n points, sparse coding seeks to find a dense dictionary matrix D with k atoms in d dimensions, and a sparse coding matrix Z with n points in k dimensions.

The original data matrix X can then be reconstructed as Z * D. Therefore, this program finds a representation of each point in X as a sparse linear combination of atoms in the dictionary D.

The sparse coding is found with an algorithm which alternates between a dictionary step, which updates the dictionary D, and a sparse coding step, which updates the sparse coding matrix.

Once a dictionary D is found, the sparse coding model may be used to encode other matrices, and saved for future usage.

To run this program, either an input matrix or an already-saved sparse coding model must be specified. An input matrix may be specified with the "training" option, along with the number of atoms in the dictionary (specified with the "atoms" parameter). It is also possible to specify an initial dictionary for the optimization, with the "initial_dictionary" parameter. An input model may be specified with the "input_model" parameter.

Value

A list with several components defining the class attributes:

codes	Matrix to save the output sparse codes of the test matrix (–test_file) to (numeric matrix).
dictionary	Matrix to save the output dictionary to (numeric matrix).
output_model	File to save trained sparse coding model to (SparseCoding).

Author(s)

mlpack developers

Examples

```
# As an example, to build a sparse coding model on the dataset "data" using
# 200 atoms and an l1-regularization parameter of 0.1, saving the model into
# "model", use
#
# \dontrun{
# output <- sparse_coding(training=data, atoms=200, lambda1=0.1)
# model <- output$output_model
# }
#
# Then, this model could be used to encode a new matrix, "otherdata", and
# save the output codes to "codes":
#
# \dontrun{
# output <- sparse_coding(input_model=model, test=otherdata)
# codes <- output$codes
# }
```

---

test_r_binding *R binding test*

Description

A simple program to test R binding functionality.

Usage

```
test_r_binding(
  double_in,
  int_in,
  string_in,
  build_model = FALSE,
  col_in = NA,
  flag1 = FALSE,
  flag2 = FALSE,
```

```
  matrix_and_info_in = NA,
  matrix_in = NA,
  model_in = NA,
  row_in = NA,
  str_vector_in = NA,
  tmatrix_in = NA,
  ucol_in = NA,
  umatrix_in = NA,
  urow_in = NA,
  vector_in = NA,
  verbose = getOption("mlpack.verbose", FALSE)
)
```

Arguments

double_in	Input double, must be 4.0 (numeric).
int_in	Input int, must be 12 (integer).
string_in	Input string, must be 'hello' (character).
build_model	If true, a model will be returned. Default value "FALSE" (logical).
col_in	Input column (numeric column).
flag1	Input flag, must be specified. Default value "FALSE" (logical).
flag2	Input flag, must not be specified. Default value "FALSE" (logical).
matrix_and_info_in	Input matrix and info (numeric matrix/data.frame with info).
matrix_in	Input matrix (numeric matrix).
model_in	Input model (GaussianKernel).
row_in	Input row (numeric row).
str_vector_in	Input vector of strings (character vector).
tmatrix_in	Input (transposed) matrix (numeric matrix).
ucol_in	Input unsigned column (integer column).
umatrix_in	Input unsigned matrix (integer matrix).
urow_in	Input unsigned row (integer row).
vector_in	Input vector of numbers (integer vector).
verbose	Display informational messages and the full list of parameters and timers at the end of execution. Default value "getOption("mlpack.verbose", FALSE)" (logical).

Details

A simple program to test R binding functionality. You can build mlpack with the BUILD_TESTS option set to off, and this binding will no longer be built.

Value

A list with several components defining the class attributes:

`col_out`	Output column. 2x input colum (numeric column).
`double_out`	Output double, will be 5.0. Default value "0" (numeric).
`int_out`	Output int, will be 13. Default value "0" (integer).
`matrix_and_info_out`	Output matrix and info; all numeric elements multiplied by 3 (numeric matrix).
`matrix_out`	Output matrix (numeric matrix).
`model_bw_out`	The bandwidth of the model. Default value "0" (numeric).
`model_out`	Output model, with twice the bandwidth (GaussianKernel).
`row_out`	Output row. 2x input row (numeric row).
`str_vector_out`	Output string vector (character vector).
`string_out`	Output string, will be 'hello2'. Default value "" (character).
`ucol_out`	Output unsigned column. 2x input column (integer column).
`umatrix_out`	Output unsigned matrix (integer matrix).
`urow_out`	Output unsigned row. 2x input row (integer row).
`vector_out`	Output vector (integer vector).

Author(s)

mlpack developers

Index

adaboost, 5
adaboost_classify, 7
adaboost_probabilities, 8
adaboost_train, 9
approx_kfn, 10

bayesian_linear_regression, 12
bayesian_linear_regression_predict, 15
bayesian_linear_regression_train, 16

cf, 17

dbscan, 20
decision_tree, 22
decision_tree_classify, 24
decision_tree_probabilities, 25
decision_tree_train, 26
det, 27

emst, 29

fastmks, 31

gmm_generate, 33
gmm_probability, 34
gmm_train, 35

hmm_generate, 37
hmm_loglik, 39
hmm_train, 40
hmm_viterbi, 41
hoeffding_tree, 42

image_converter, 45

kde, 46
kernel_pca, 49
kfn, 51
kmeans, 53
knn, 56
krann, 58

lars, 60
lars_predict *(*predict.mlpack_lars*)*, 98
lars_train, 62
linear_regression, 64
linear_regression_predict
*(*predict.mlpack_linear_regression*)*, 99
linear_regression_train, 66
linear_svm, 67
lmnn, 70
local_coordinate_coding, 73
logistic_regression, 75
logistic_regression_classify
*(*predict.mlpack_logistic_regression*)*, 100
logistic_regression_probabilities, 78
logistic_regression_train, 79
lsh, 81

mean_shift, 83
mlpack, 84
mlpack-package *(*mlpack*)*, 84

nbc, 89
nca, 91
nmf, 93

pca, 95
perceptron, 96
predict.mlpack_adaboost
*(*adaboost_classify*)*, 7
predict.mlpack_bayesian_linear_regression
*(*bayesian_linear_regression*)*, 12
predict.mlpack_lars, 98
predict.mlpack_linear_regression, 99
predict.mlpack_logistic_regression, 100
predict.mlpack_random_forest, 101
preprocess_binarize, 104

preprocess_describe, 105
preprocess_one_hot_encoding, 107
preprocess_scale, 108
preprocess_split, 110

radical, 112
random_forest
        (predict.mlpack_random_forest), 101
random_forest_classify, 114
random_forest_probabilities, 115
random_forest_train, 116

Serialize, 118
softmax_regression, 118
sparse_coding, 120

test_r_binding, 122

Unserialize (Serialize), 118