# Package 'superml'

November 29, 2019

**Type** Package

**Title** Build Machine Learning Models Like Using Python's Scikit-Learn Library in R

**Version** 0.5.0

**Maintainer** Manish Saraswat <manish06saraswat@gmail.com>

**Description** The idea is to provide a standard interface to users who use both R and Python for building machine learning models. This package provides a scikit-learn's fit, predict interface to train machine learning models in R.

**License** GPL-3 | file LICENSE

**Encoding** UTF-8

**LazyData** true

**URL** https://github.com/saraswatmks/superml

**BugReports** https://github.com/saraswatmks/superml/issues

**Depends** R(>= 3.5), R6(>= 2.2)

**Imports** data.table (>= 1.10), assertthat (>= 0.2), parallel, doParallel (>= 1.0), Metrics (>= 0.1)

**Suggests** knitr, rlang, testthat, rmarkdown, naivebayes(>= 0.9), ClusterR(>= 1.1), FNN(>= 1.1), ranger(>= 0.10), caret(>= 6.0), xgboost(>= 0.6), glmnet(>= 2.0), kableExtra, tm(>= 0.7), Matrix(>= 1.2), e1071(>= 1.7)

**RoxygenNote** 7.0.1

**VignetteBuilder** knitr

**NeedsCompilation** no

**Author** Manish Saraswat [aut, cre]

**Repository** CRAN

**Date/Publication** 2019-11-29 09:10:03 UTC

# R topics documented:



---

`bm25` *Best Matching(BM25)*

---

## Description

Computer BM25 distance between sentences/documents.

## Details

BM25 stands for Best Matching 25. It is widely using for ranking documents and a preferred method than TF*IDF scores. It is used to find the similar documents from a corpus, given a new document. It is popularly used in information retrieval systems. This implementation uses multiple cores for faster and parallel computation.

## Public fields

`corpus` a list containing sentences

`use_parallel` enables parallel computation, defaults to FALSE

## Methods

### Public methods:

- `bm25$new()`
- `bm25$most_similar()`
- `bm25$clone()`

**Method** `new()`**:**

*Usage:*

```
bm25$new(corpus, use_parallel)
```

*Arguments:*

`corpus` list, a list containing sentences

`use_parallel` logical, enables parallel computation, defaults to FALSE. if TRUE uses n - 1 cores.

*Details:* Create a new 'bm25' object.

*Returns:* A 'bm25' object.

*Examples:*

```
example <- c('white audi 2.5 car','black shoes from office',
             'new mobile iphone 7','audi tyres audi a3',
             'nice audi bmw toyota corolla')
obj <- bm25$new(example, use_parallel=FALSE)
```

**Method** `most_similar()`**:**

*Usage:*

```
bm25$most_similar(document, topn = 1)
```

*Arguments:*

`document` character, for this value we find most similar sentences.

`topn` integer, top n sentences to retrieve

*Details:* Returns a list of the most similar sentence

*Returns:* a vector of most similar documents

*Examples:*

```
example <- c('white audi 2.5 car','black shoes from office',
             'new mobile iphone 7','audi tyres audi a3',
             'nice audi bmw toyota corolla')
get_bm <- bm25$new(example, use_parallel=FALSE)
input_document <- c('white toyota corolla')
get_bm$most_similar(document = input_document, topn = 2)
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
bm25$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `bm25$new`
## ------------------------------------------------
```

```
example <- c('white audi 2.5 car','black shoes from office',
             'new mobile iphone 7','audi tyres audi a3',
             'nice audi bmw toyota corolla')
obj <- bm25$new(example, use_parallel=FALSE)

## ------------------------------------------------
## Method `bm25$most_similar`
## ------------------------------------------------

example <- c('white audi 2.5 car','black shoes from office',
             'new mobile iphone 7','audi tyres audi a3',
             'nice audi bmw toyota corolla')
get_bm <- bm25$new(example, use_parallel=FALSE)
input_document <- c('white toyota corolla')
get_bm$most_similar(document = input_document, topn = 2)
```

---

`cla_train` *cla_train*

---

## Description

Training Dataset used for classification examples. This is classic titanic dataset used to predict if a passenger will survive or not in titanic ship disaster.

## Usage

```
cla_train
```

## Format

An object of class `data.table` (inherits from `data.frame`) with 891 rows and 12 columns.

## Source

https://www.kaggle.com/c/titanic/data

---

`Counter` *Calculate count of values in a list or vector*

---

## Description

Handy function to calculate count of values given in a list or vector

## Usage

```
Counter(data, sort = TRUE, decreasing = FALSE)
```

### Arguments

| | |
|---|---|
| data | should be a vector or list of input values |
| sort | a logical value, to sort the result or not |
| decreasing | a logical value, the order of sorting to be followed |

### Value

count of values in a list

### Examples

```
d <- list(c('i','am','bad'),c('you','are','also','bad'))
counts <- Counter(d, sort=TRUE, decreasing=TRUE)
```

---

## CountVectorizer *Count Vectorizer*

---

### Description

Creates CountVectorizer Model.

### Details

Given a list of text, it generates a bag of words model and returns a sparse matrix consisting of token counts.

### Public fields

`sentences` a list containing sentences

`max_df` When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold, value lies between 0 and 1.

`min_df` When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold, value lies between 0 and 1.

`max_features` use top features sorted by count to be used in bag of words matrix.

`split` splitting criteria for strings, default: " "

`regex` regex expression to use for text cleaning.

`model` internal attribute which stores the count model

`remove_stopwords` a list of stopwords to use, by default it uses its inbuilt list of standard stopwords

## Methods

### Public methods:

- CountVectorizer$new()
- CountVectorizer$fit()
- CountVectorizer$fit_transform()
- CountVectorizer$transform()
- CountVectorizer$clone()

### Method new():

*Usage:*

```
CountVectorizer$new(
  min_df,
  max_df,
  max_features,
  regex,
  remove_stopwords,
  split
)
```

*Arguments:*

`min_df` numeric, When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold, value lies between 0 and 1.

`max_df` nuemric, When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold, value lies between 0 and 1.

`max_features` integer, use top features sorted by count to be used in bag of words matrix.

`regex` character, regex expression to use for text cleaning.

`remove_stopwords` list, a list of stopwords to use, by default it uses its inbuilt list of standard stopwords

`split` character, splitting criteria for strings, default: " "

*Details:* Create a new 'CountVectorizer' object.

*Returns:* A 'CountVectorizer' object.

*Examples:*

```
cv = CountVectorizer$new(min_df=0.1)
```

### Method fit():

*Usage:*

```
CountVectorizer$fit(sentences)
```

*Arguments:*

`sentences` a list of text sentences

*Details:* Fits the countvectorizer model on sentences

*Returns:* NULL

*Examples:*

```
sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
cv = CountVectorizer$new(min_df=0.1)
cv$fit(sents)
```

**Method** `fit_transform()`**:**

*Usage:*

```
CountVectorizer$fit_transform(sentences)
```

*Arguments:*

`sentences` a list of text sentences

*Details:* Fits the countvectorizer model and returns a sparse matrix of count of tokens

*Returns:* a sparse matrix containing count of tokens in each given sentence

*Examples:*

```
sents = c('i am alone in dark.','mother_mary a lot',
         'alone in the dark?', 'many mothers in the lot....')
cv <- CountVectorizer$new(min_df=0.1)
cv_count_matrix <- cv$fit_transform(sents)
```

**Method** `transform()`**:**

*Usage:*

```
CountVectorizer$transform(sentences)
```

*Arguments:*

`sentences` a list of new text sentences

*Details:* Returns a matrix of count of tokens

*Returns:* a sparse matrix containing count of tokens in each given sentence

*Examples:*

```
sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
new_sents <- c("dark at night",'mothers day')
cv = CountVectorizer$new(min_df=0.1)
cv$fit(sents)
cv_count_matrix <- cv$transform(new_sents)
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
CountVectorizer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `CountVectorizer$new`
## ------------------------------------------------

cv = CountVectorizer$new(min_df=0.1)

## ------------------------------------------------
## Method `CountVectorizer$fit`
## ------------------------------------------------

sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
cv = CountVectorizer$new(min_df=0.1)
cv$fit(sents)

## ------------------------------------------------
## Method `CountVectorizer$fit_transform`
## ------------------------------------------------

sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
cv <- CountVectorizer$new(min_df=0.1)
cv_count_matrix <- cv$fit_transform(sents)

## ------------------------------------------------
## Method `CountVectorizer$transform`
## ------------------------------------------------

sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
new_sents <- c("dark at night",'mothers day')
cv = CountVectorizer$new(min_df=0.1)
cv$fit(sents)
cv_count_matrix <- cv$transform(new_sents)
```

---

GridSearchCV *Grid Search CV*

---

## Description

Runs grid search cross validation scheme to find best model training parameters.

## Details

Grid search CV is used to train a machine learning model with multiple combinations of training hyper parameters and finds the best combination of parameters which optimizes the evaluation metric. It creates an exhaustive set of hyperparameter combinations and train model on each combination.

## Public fields

`trainer` superml trainer object, could be either XGBTrainer, RFTrainer, NBTrainer etc.

`parameters` a list of parameters to tune

`n_folds` number of folds to use to split the train data

`scoring` scoring metric used to evaluate the best model, multiple values can be provided. currently supports: auc, accuracy, mse, rmse, logloss, mae, f1, precision, recall

`evaluation_scores` parameter for internal use

## Methods

### Public methods:

- `GridSearchCV$new()`
- `GridSearchCV$fit()`
- `GridSearchCV$best_iteration()`
- `GridSearchCV$clone()`

### Method `new()`:

*Usage:*

```
GridSearchCV$new(trainer = NA, parameters = NA, n_folds = NA, scoring = NA)
```

*Arguments:*

`trainer` superml trainer object, could be either XGBTrainer, RFTrainer, NBTrainer etc.

`parameters` list, a list of parameters to tune

`n_folds` integer, number of folds to use to split the train data

`scoring` character, scoring metric used to evaluate the best model, multiple values can be provided. currently supports: auc, accuracy, mse, rmse, logloss, mae, f1, precision, recall

*Details:* Create a new 'GridSearchCV' object.

*Returns:* A 'GridSearchCV' object.

*Examples:*

```
rf <- RFTrainer$new()
gst <-GridSearchCV$new(trainer = rf,
                       parameters = list(n_estimators = c(100),
                                         max_depth = c(5,2,10)),
                                         n_folds = 3,
                                         scoring = c('accuracy','auc'))
```

### Method `fit()`:

*Usage:*

```
GridSearchCV$fit(X, y)
```

*Arguments:*

`X` data.frame or data.table

`y` character, name of target variable

*Details:* Trains the model using grid search

*Returns:* NULL

*Examples:*

```
rf <- RFTrainer$new()
gst <-GridSearchCV$new(trainer = rf,
                       parameters = list(n_estimators = c(100),
                                         max_depth = c(5,2,10)),
                                         n_folds = 3,
                                         scoring = c('accuracy','auc'))
data("iris")
gst$fit(iris, "Species")
```

**Method** `best_iteration()`**:**

*Usage:*

```
GridSearchCV$best_iteration(metric = NULL)
```

*Arguments:*

`metric` character, which metric to use for evaluation

*Details:* Returns the best parameters

*Returns:* a list of best parameters

*Examples:*

```
rf <- RFTrainer$new()
gst <-GridSearchCV$new(trainer = rf,
                       parameters = list(n_estimators = c(100),
                                         max_depth = c(5,2,10)),
                                         n_folds = 3,
                                         scoring = c('accuracy','auc'))
data("iris")
gst$fit(iris, "Species")
gst$best_iteration()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
GridSearchCV$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `GridSearchCV$new`
## ------------------------------------------------

rf <- RFTrainer$new()
gst <-GridSearchCV$new(trainer = rf,
                       parameters = list(n_estimators = c(100),
                                         max_depth = c(5,2,10)),
```

```
                                        n_folds = 3,
                                        scoring = c('accuracy','auc'))

## ------------------------------------------------
## Method `GridSearchCV$fit`
## ------------------------------------------------

rf <- RFTrainer$new()
gst <-GridSearchCV$new(trainer = rf,
                       parameters = list(n_estimators = c(100),
                                         max_depth = c(5,2,10)),
                                         n_folds = 3,
                                         scoring = c('accuracy','auc'))
data("iris")
gst$fit(iris, "Species")

## ------------------------------------------------
## Method `GridSearchCV$best_iteration`
## ------------------------------------------------

rf <- RFTrainer$new()
gst <-GridSearchCV$new(trainer = rf,
                       parameters = list(n_estimators = c(100),
                                         max_depth = c(5,2,10)),
                                         n_folds = 3,
                                         scoring = c('accuracy','auc'))
data("iris")
gst$fit(iris, "Species")
gst$best_iteration()
```

---

`kFoldMean` *kFoldMean Calculator*

---

## Description

Calculates out-of-fold mean features (also known as target encoding) for train and test data. This strategy is widely used to avoid overfitting or causing leakage while creating features using the target variable. This method is experimental. If the results you get are unexpected, please report them in github issues.

## Usage

```
kFoldMean(train_df, test_df, colname, target, n_fold = 5, seed = 42)
```

## Arguments

| | |
|---|---|
| `train_df` | train dataset |
| `test_df` | test dataset |
| `colname` | name of categorical column |

| | |
|---|---|
| target | the target or dependent variable, should be a string. |
| n_fold | the number of folds to use for doing kfold computation, default=5 |
| seed | the seed value, to ensure reproducibility, it could be any positive value, default=42 |

## Value

a train and test data table with out-of-fold mean value of the target for the given categorical variable

## Examples

```
train <- data.frame(region=c('del','csk','rcb','del','csk','pune','guj','del'),
                    win = c(0,1,1,0,0,0,0,1))
test <- data.frame(region=c('rcb','csk','rcb','del','guj','pune','csk','kol'))
train_result <- kFoldMean(train_df = train,
                          test_df = test,
                          colname = 'region',
                          target = 'win',
                          seed = 1220)$train

test_result <- kFoldMean(train_df = train,
                         test_df = test,
                         colname = 'region',
                         target = 'win',
                         seed = 1220)$test
```

---

# KMeansTrainer *K-Means Trainer*

---

## Description

Trains a k-means machine learning model in R

## Details

Trains a unsupervised K-Means clustering algorithm. It borrows mini-batch k-means function from ClusterR package written in c++, hence it is quite fast.

## Public fields

`clusters` the number of clusters

`batch_size` the size of the mini batches

`num_init` number of times the algorithm will be run with different centroid seeds

`max_iters` the maximum number of clustering iterations

`init_fraction` percentage of data to use for the initialization centroids (applies if initializer is kmeans++ or optimal_init). Should be a float number between 0.0 and 1.0.

`initializer` the method of initialization. One of, optimal_init, quantile_init, kmeans++ and random.

`early_stop_iter` continue that many iterations after calculation of the best within-cluster-sum-ofsquared-error

`verbose` either TRUE or FALSE, indicating whether progress is printed during clustering

`centroids` a matrix of initial cluster centroids. The rows of the CENTROIDS matrix should be equal to the number of clusters and the columns should be equal to the columns of the data

`tol` a float number. If, in case of an iteration (iteration > 1 and iteration < max_iters) "tol" is greater than the squared norm of the centroids, then kmeans has converged

`tol_optimal_init` tolerance value for the 'optimal_init' initializer. The higher this value is, the far appart from each other the centroids are.

`seed` integer value for random number generator (RNG)

`model` use for internal purpose

`max_clusters` either a numeric value, a contiguous or non-continguous numeric vector specifying the cluster search space

## Methods

### Public methods:

- `KMeansTrainer$new()`
- `KMeansTrainer$fit()`
- `KMeansTrainer$predict()`
- `KMeansTrainer$clone()`

### Method `new()`:

*Usage:*

```
KMeansTrainer$new(
  clusters,
  batch_size = 10,
  num_init = 1,
  max_iters = 100,
  init_fraction = 1,
  initializer = "kmeans++",
  early_stop_iter = 10,
  verbose = FALSE,
  centroids = NULL,
  tol = 1e-04,
  tol_optimal_init = 0.3,
  seed = 1,
  max_clusters = NA
)
```

*Arguments:*

`clusters` numeric, When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold, value lies between 0 and 1.

`batch_size` nuemric, When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold, value lies between 0 and 1.

`num_init` integer, use top features sorted by count to be used in bag of words matrix.

`max_iters` character, regex expression to use for text cleaning.

`init_fraction` list, a list of stopwords to use, by default it uses its inbuilt list of standard stopwords

`initializer` character, splitting criteria for strings, default: " "

`early_stop_iter` continue that many iterations after calculation of the best within-cluster-sum-ofsquared-error

`verbose` either TRUE or FALSE, indicating whether progress is printed during clustering

`centroids` a matrix of initial cluster centroids. The rows of the CENTROIDS matrix should be equal to the number of clusters and the columns should be equal to the columns of the data

`tol` a float number. If, in case of an iteration (iteration > 1 and iteration < max_iters) "tol" is greater than the squared norm of the centroids, then kmeans has converged

`tol_optimal_init` tolerance value for the 'optimal_init' initializer. The higher this value is, the far appart from each other the centroids are.

`seed` integer value for random number generator (RNG)

`max_clusters` either a numeric value, a contiguous or non-continguous numeric vector specifying the cluster search space

*Details:* Create a new 'KMeansTrainer' object.

*Returns:* A 'KMeansTrainer' object.

*Examples:*

```
data <- rbind(replicate(20, rnorm(1e4, 2)),
              replicate(20, rnorm(1e4, -1)),
              replicate(20, rnorm(1e4, 5)))
km_model <- KMeansTrainer$new(clusters=2, batch_size=30, max_clusters=6)
```

**Method** `fit()`**:**

*Usage:*

```
KMeansTrainer$fit(X, y = NULL, find_optimal = FALSE)
```

*Arguments:*

`X` data.frame or matrix containing features

`y` NULL only kept here for superml's standard way

`find_optimal` logical, to find the optimal clusters automatically

*Details:* Trains the KMeansTrainer model

*Returns:* NULL

*Examples:*

```
data <- rbind(replicate(20, rnorm(1e4, 2)),
              replicate(20, rnorm(1e4, -1)),
              replicate(20, rnorm(1e4, 5)))
km_model <- KMeansTrainer$new(clusters=2, batch_size=30, max_clusters=6)
km_model$fit(data, find_optimal = FALSE)
```

**Method** `predict()`**:**

*Usage:*

```
KMeansTrainer$predict(X)
```

*Arguments:*

`X` data.frame or matrix

*Details:* Returns the prediction on test data

*Returns:* a vector of predictions

*Examples:*

```
data <- rbind(replicate(20, rnorm(1e4, 2)),
              replicate(20, rnorm(1e4, -1)),
              replicate(20, rnorm(1e4, 5)))
km_model <- KMeansTrainer$new(clusters=2, batch_size=30, max_clusters=6)
km_model$fit(data, find_optimal = FALSE)
predictions <- km_model$predict(data)
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
KMeansTrainer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `KMeansTrainer$new`
## ------------------------------------------------

data <- rbind(replicate(20, rnorm(1e4, 2)),
              replicate(20, rnorm(1e4, -1)),
              replicate(20, rnorm(1e4, 5)))
km_model <- KMeansTrainer$new(clusters=2, batch_size=30, max_clusters=6)

## ------------------------------------------------
## Method `KMeansTrainer$fit`
## ------------------------------------------------

data <- rbind(replicate(20, rnorm(1e4, 2)),
              replicate(20, rnorm(1e4, -1)),
              replicate(20, rnorm(1e4, 5)))
km_model <- KMeansTrainer$new(clusters=2, batch_size=30, max_clusters=6)
km_model$fit(data, find_optimal = FALSE)

## ------------------------------------------------
## Method `KMeansTrainer$predict`
## ------------------------------------------------
```

```
data <- rbind(replicate(20, rnorm(1e4, 2)),
              replicate(20, rnorm(1e4, -1)),
              replicate(20, rnorm(1e4, 5)))
km_model <- KMeansTrainer$new(clusters=2, batch_size=30, max_clusters=6)
km_model$fit(data, find_optimal = FALSE)
predictions <- km_model$predict(data)
```

---

`KNNTrainer` *K Nearest Neighbours Trainer*

---

## Description

Trains a k nearest neighbour model using fast search algorithms. KNN is a supervised learning algorithm which is used for both regression and classification problems.

## Format

`R6Class` object.

## Usage

For usage details see **Methods, Arguments and Examples** sections.

```
bst = KNNTrainer$new(k=1, prob=FALSE, algorithm=NULL, type="class")
bst$fit(X_train, X_test, "target")
bst$predict(type)
```

## Methods

`$new()` Initialise the instance of the trainer

`$fit()` trains the knn model and stores the test prediction

`$predict()` returns predictions

## Arguments

**k** number of neighbours to predict

**prob** if probability should be computed, default=FALSE

**algorithm** algorithm used to train the model, possible values are 'kd_tree','cover_tree','brute'

**type** type of problem to solve i.e. regression or classification, possible values are 'reg' or 'class'

## Public fields

`k` number of neighbours to predict

`prob` if probability should be computed, default=FALSE

`algorithm` algorithm used to train the model, possible values are 'kd_tree','cover_tree','brute'

`type` type of problem to solve i.e. regression or classification, possible values are 'reg' or 'class'

`model` for internal use

## Methods

### Public methods:

- KNNTrainer$new()
- KNNTrainer$fit()
- KNNTrainer$predict()
- KNNTrainer$clone()

### Method new():

*Usage:*

```
KNNTrainer$new(k, prob, algorithm, type)
```

*Arguments:*

k k number of neighbours to predict

prob if probability should be computed, default=FALSE

algorithm algorithm used to train the model, possible values are 'kd_tree','cover_tree','brute'

type type of problem to solve i.e. regression or classification, possible values are 'reg' or 'class'

*Details:* Create a new 'KNNTrainer' object.

*Returns:* A 'KNNTrainer' object.

*Examples:*

```
data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')
pred <- bst$predict(type="raw")
```

### Method fit():

*Usage:*

```
KNNTrainer$fit(train, test, y)
```

*Arguments:*

train data.frame or matrix

test data.frame or matrix

y character, name of target variable

*Details:* Trains the KNNTrainer model

*Returns:* NULL

*Examples:*

```
data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')
```

**Method** `predict()`**:**

*Usage:*

```
KNNTrainer$predict(type = "raw")
```

*Arguments:*

`type` character, 'raw' for labels else 'prob'

*Details:* Predits the nearest neigbours for test data

*Returns:* a list of predicted neighbours

*Examples:*

```
data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')
pred <- bst$predict(type="raw")
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
KNNTrainer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')
```

```
pred <- bst$predict(type="raw")

## ------------------------------------------------
## Method `KNNTrainer$new`
## ------------------------------------------------

data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')
pred <- bst$predict(type="raw")

## ------------------------------------------------
## Method `KNNTrainer$fit`
## ------------------------------------------------

data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')

## ------------------------------------------------
## Method `KNNTrainer$predict`
## ------------------------------------------------

data("iris")

iris$Species <- as.integer(as.factor(iris$Species))

xtrain <- iris[1:100,]
xtest <- iris[101:150,]

bst <- KNNTrainer$new(k=3, prob=TRUE, type="class")
bst$fit(xtrain, xtest, 'Species')
pred <- bst$predict(type="raw")
```

---

LabelEncoder *Label Encoder*

---

## Description

Encodes and decodes categorical variables into integer values and vice versa. This is a commonly performed task in data preparation during model training, because all machine learning models require the data to be encoded into numerical format. It takes a vector of character or factor values and encodes them into numeric.

## Format

`R6Class` object.

## Usage

For usage details see **Methods, Arguments and Examples** sections.

```
lbl = LabelEncoder$new()
lbl$fit(x)
lbl$fit_transform(x)
lbl$transform(x)
```

## Methods

`$new()` Initialise the instance of the encoder

`$fit()` creates a memory of encodings but doesn't return anything

`$transform()` based on encodings learned in `fit` method is applies the transformation

`$fit_transform()` encodes the data and keep a memory of encodings simultaneously

`$inverse_transform()` encodes the data and keep a memory of encodings simultaneously

## Arguments

**data** a vector or list containing the character / factor values

## Public fields

`input_data` internal use

`encodings` internal use

`decodings` internal use

`fit_model` internal use

## Methods

### Public methods:

- `LabelEncoder$fit()`
- `LabelEncoder$fit_transform()`
- `LabelEncoder$transform()`
- `LabelEncoder$inverse_transform()`
- `LabelEncoder$clone()`

**Method** `fit()`**:**

*Usage:*

```
LabelEncoder$fit(data_col)
```

*Arguments:*

`data_col` a vector containing non-null values

*Details:* Fits the labelencoder model on given data

*Returns:* NULL, calculates the encoding and save in memory

*Examples:*

```
data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$fit_transform(data_ex$Name)
decode_names <- lbl$inverse_transform(data_ex$Name)
```

**Method** `fit_transform()`**:**

*Usage:*

```
LabelEncoder$fit_transform(data_col)
```

*Arguments:*

`data_col` a vector containing non-null values

*Details:* Fits and returns the encoding

*Returns:* encoding values for the given input data

*Examples:*

```
data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$fit_transform(data_ex$Name)
```

**Method** `transform()`**:**

*Usage:*

```
LabelEncoder$transform(data_col)
```

*Arguments:*

`data_col` a vector containing non-null values

*Details:* Returns the encodings from the fitted model

*Returns:* encoding values for the given input data

*Examples:*

```
data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$transform(data_ex$Name)
```

**Method** `inverse_transform()`**:**

*Usage:*

```
LabelEncoder$inverse_transform(coded_col)
```

*Arguments:*

`coded_col` a vector containing label encoded values

*Details:* Gives back the original values from a encoded values

*Returns:* original values from the label encoded data

*Examples:*

```
data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$fit_transform(data_ex$Name)
decode_names <- lbl$inverse_transform(data_ex$Name)
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
LabelEncoder$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
data_ex$Name <- lbl$fit_transform(data_ex$Name)
decode_names <- lbl$inverse_transform(data_ex$Name)

## ------------------------------------------------
## Method `LabelEncoder$fit`
## ------------------------------------------------

data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$fit_transform(data_ex$Name)
decode_names <- lbl$inverse_transform(data_ex$Name)

## ------------------------------------------------
## Method `LabelEncoder$fit_transform`
## ------------------------------------------------

data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$fit_transform(data_ex$Name)

## ------------------------------------------------
## Method `LabelEncoder$transform`
## ------------------------------------------------
```

```
data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$transform(data_ex$Name)

## ------------------------------------------------
## Method `LabelEncoder$inverse_transform`
## ------------------------------------------------

data_ex <- data.frame(Score = c(10,20,30,4), Name=c('Ao','Bo','Bo','Co'))
lbl <- LabelEncoder$new()
lbl$fit(data_ex$Name)
data_ex$Name <- lbl$fit_transform(data_ex$Name)
decode_names <- lbl$inverse_transform(data_ex$Name)
```

---

`LMTrainer` *Linear Models Trainer*

---

## Description

Trains regression, lasso, ridge model in R

## Details

Trains linear models such as Logistic, Lasso or Ridge regression model. It is built on glmnet R package. This class provides fit, predict, cross valdidation functions.

## Public fields

`family` type of regression to perform, values can be "gaussian" ,"binomial", "multinomial","mgaussian"

`weights` observation weights. Can be total counts if responses are proportion matrices. Default is 1 for each observation

`alpha` The elasticnet mixing parameter, alpha=1 is the lasso penalty, alpha=0 the ridge penalty, alpha=NULL is simple regression

`lambda` the number of lambda values - default is 100

`standardize` normalise the features in the given data

`standardize.response` normalise the dependent variable between 0 and 1, default = FALSE

`model` internal use

`cvmodel` internal use

`Flag` internal use

`is_lasso` internal use

`iid_names` internal use

## Methods

### Public methods:

- LMTrainer$new()
- LMTrainer$fit()
- LMTrainer$predict()
- LMTrainer$cv_model()
- LMTrainer$cv_predict()
- LMTrainer$get_importance()
- LMTrainer$clone()

### Method new():

*Usage:*

```
LMTrainer$new(family, weights, alpha, lambda, standardize.response)
```

*Arguments:*

`family` character, type of regression to perform, values can be "gaussian" ,"binomial", "multinomial","mgaussian"

`weights` numeric, observation weights. Can be total counts if responses are proportion matrices. Default is 1 for each observation

`alpha` integer, The elasticnet mixing parameter, alpha=1 is the lasso penalty, alpha=0 the ridge penalty, alpha=NULL is simple regression

`lambda` integer, the number of lambda values - default is 100

`standardize.response` logical, normalise the dependent variable between 0 and 1, default = FALSE

*Details:* Create a new 'LMTrainer' object.

*Returns:* A 'LMTrainer' object.

*Examples:*

```
\donttest{
LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
}
```

### Method fit():

*Usage:*

```
LMTrainer$fit(X, y)
```

*Arguments:*

`X` data.frame containing train featuers

`y` character, name of target variable

*Details:* Fits the LMTrainer model on given data

*Returns:* NULL, train the model and saves internally

*Examples:*

```
\donttest{
LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$fit(X = housing, y = 'MEDV')
}
```

**Method** `predict()`**:**

*Usage:*

```
LMTrainer$predict(df, lambda = NULL)
```

*Arguments:*

`df` data.frame containing test features

`lambda` integer, the number of lambda values - default is 100. By default it picks the best value from the model.

*Details:* Returns predictions for test data

*Returns:* vector, a vector containing predictions

*Examples:*

```
\donttest{
LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$fit(X = housing, y = 'MEDV')
predictions <- lf$cv_predict(df = housing)
}
```

**Method** `cv_model()`**:**

*Usage:*

```
LMTrainer$cv_model(X, y, nfolds, parallel, type.measure = "deviance")
```

*Arguments:*

`X` data.frame containing test features

`y` character, name of target variable

`nfolds` integer, number of folds

`parallel` logical, if do parallel computation. Default=FALSE

`type.measure` character, evaluation metric type. Default = deviance

*Details:* Train regression model using cross validation

*Returns:* NULL, trains the model and saves it in memory

*Examples:*

```
\donttest{
LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$cv_model(X = housing, y = 'MEDV', nfolds = 5, parallel = FALSE)
}
```

**Method** `cv_predict()`**:**

*Usage:*

```
LMTrainer$cv_predict(df, lambda = NULL)
```

*Arguments:*

`df` data.frame containing test features

`lambda` integer, the number of lambda values - default is 100. By default it picks the best value from the model.

*Details:* Get predictions from the cross validated regression model

*Returns:* vector a vector containing predicted values

*Examples:*

```
\donttest{
LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$cv_model(X = housing, y = 'MEDV', nfolds = 5, parallel = FALSE)
predictions <- lf$cv_predict(df = housing)
}
```

**Method** `get_importance()`**:**

*Usage:*

```
LMTrainer$get_importance()
```

*Details:* Get feature importance using model coefficients

*Returns:* a matrix containing feature coefficients

*Examples:*

```
\donttest{
LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
```

```
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$cv_model(X = housing, y = 'MEDV', nfolds = 5, parallel = FALSE)
predictions <- lf$cv_predict(df = housing)
coefs <- lf$get_importance()
}
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
LMTrainer$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `LMTrainer$new`
## ------------------------------------------------


LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)


## ------------------------------------------------
## Method `LMTrainer$fit`
## ------------------------------------------------


LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
           "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$fit(X = housing, y = 'MEDV')


## ------------------------------------------------
## Method `LMTrainer$predict`
## ------------------------------------------------


LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
```

```
          "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$fit(X = housing, y = 'MEDV')
predictions <- lf$cv_predict(df = housing)


## ------------------------------------------------
## Method `LMTrainer$cv_model`
## ------------------------------------------------


LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
          "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$cv_model(X = housing, y = 'MEDV', nfolds = 5, parallel = FALSE)


## ------------------------------------------------
## Method `LMTrainer$cv_predict`
## ------------------------------------------------


LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
          "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$cv_model(X = housing, y = 'MEDV', nfolds = 5, parallel = FALSE)
predictions <- lf$cv_predict(df = housing)


## ------------------------------------------------
## Method `LMTrainer$get_importance`
## ------------------------------------------------


LINK <- "http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
housing <- read.table(LINK)
names <- c("CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS",
          "RAD","TAX","PTRATIO","B","LSTAT","MEDV")
names(housing)  <-  names
lf <- LMTrainer$new(family = 'gaussian', alpha=1)
lf$cv_model(X = housing, y = 'MEDV', nfolds = 5, parallel = FALSE)
predictions <- lf$cv_predict(df = housing)
coefs <- lf$get_importance()
```

NBTrainer *Naive Bayes Trainer*

## Description

Trains a probabilistic naive bayes model

## Details

Trains a naive bayes model. It is built on top high performance naivebayes R package.

## Public fields

`prior` numeric vector with prior probabilities. vector with prior probabilities of the classes. If unspecified, the class proportions for the training set are used. If present, the probabilities should be specified in the order of the factor levels.

`laplace` value used for Laplace smoothing. Defaults to 0 (no Laplace smoothing)

`usekernel` if TRUE, density is used to estimate the densities of metric predictors

`model` for internal use

## Methods

### Public methods:

- `NBTrainer$new()`
- `NBTrainer$fit()`
- `NBTrainer$predict()`
- `NBTrainer$clone()`

### Method `new()`:

*Usage:*

```
NBTrainer$new(prior, laplace, usekernel)
```

*Arguments:*

`prior` numeric, prior numeric vector with prior probabilities. vector with prior probabilities of the classes. If unspecified, the class proportions for the training set are used. If present, the probabilities should be specified in the order of the factor levels.

`laplace` nuemric, value used for Laplace smoothing. Defaults to 0 (no Laplace smoothing)

`usekernel` logical, if TRUE, density is used to estimate the densities of metric predictors

*Details:* Create a new 'NBTrainer' object.

*Returns:* A 'NBTrainer' object.

*Examples:*

```
data(iris)
nb <- NBTrainer$new()
```

**Method** `fit()`**:**

*Usage:*

```
NBTrainer$fit(X, y)
```

*Arguments:*

`X` data.frame containing train features

`y` character, name of target variable

*Details:* Fits the naive bayes model

*Returns:* NULL, trains and saves the model in memory

*Examples:*

```
data(iris)
nb <- NBTrainer$new()
nb$fit(iris, 'Species')
```

**Method** `predict()`**:**

*Usage:*

```
NBTrainer$predict(X, type = "class")
```

*Arguments:*

`X` data.frame containing test features

`type` character, if the predictions should be labels or probability

*Details:* Returns predictions from the model

*Returns:* NULL, trains and saves the model in memory

*Examples:*

```
data(iris)
nb <- NBTrainer$new()
nb$fit(iris, 'Species')
y <- nb$predict(iris)
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
NBTrainer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `NBTrainer$new`
## ------------------------------------------------

data(iris)
nb <- NBTrainer$new()

## ------------------------------------------------
```

```
## Method `NBTrainer$fit`
## ------------------------------------------------

data(iris)
nb <- NBTrainer$new()
nb$fit(iris, 'Species')

## ------------------------------------------------
## Method `NBTrainer$predict`
## ------------------------------------------------

data(iris)
nb <- NBTrainer$new()
nb$fit(iris, 'Species')
y <- nb$predict(iris)
```

---

RandomSearchCV *Random Search CV*

---

## Description

Hyperparameter tuning using random search scheme.

## Details

Given a set of hyper parameters, random search trainer provides a faster way of hyper parameter tuning. Here, the number of models to be trained can be defined by the user.

## Super class

superml::GridSearchCV -> RandomSearchTrainer

## Public fields

`n_iter` number of models to be trained

## Methods

### Public methods:

- RandomSearchCV$new()
- RandomSearchCV$fit()
- RandomSearchCV$clone()

### Method `new()`:

*Usage:*

```
RandomSearchCV$new(
  trainer = NA,
  parameters = NA,
  n_folds = NA,
  scoring = NA,
  n_iter = NA
)
```

*Arguments:*

`trainer` superml trainer object, must be either XGBTrainer, LMTrainer, RFTrainer, NBTrainer

`parameters` list, list containing parameters

`n_folds` integer, number of folds to use to split the train data

`scoring` character, scoring metric used to evaluate the best model, multiple values can be provided. currently supports: auc, accuracy, mse, rmse, logloss, mae, f1, precision, recall

`n_iter` integer, number of models to be trained

*Details:* Create a new 'RandomSearchTrainer' object.

*Returns:* A 'RandomSearchTrainer' object.

*Examples:*

```
rf <- RFTrainer$new()
rst <-RandomSearchCV$new(trainer = rf,
                         parameters = list(n_estimators = c(100,500),
                         max_depth = c(5,2,10,14)),
                         n_folds = 3,
                         scoring = c('accuracy','auc'),
                         n_iter = 4)
```

**Method** `fit()`**:**

*Usage:*

```
RandomSearchCV$fit(X, y)
```

*Arguments:*

`X` data.frame containing features

`y` character, name of target variable

*Details:* Train the model on given hyperparameters

*Returns:* NULL, tunes hyperparameters and stores the result in memory

*Examples:*

```
rf <- RFTrainer$new()
rst <-RandomSearchCV$new(trainer = rf,
                         parameters = list(n_estimators = c(100,500),
                         max_depth = c(5,2,10,14)),
                         n_folds = 3,
                         scoring = c('accuracy','auc'),
                         n_iter = 4)
data("iris")
rst$fit(iris, "Species")
rst$best_iteration()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
RandomSearchCV$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `RandomSearchCV$new`
## ------------------------------------------------

rf <- RFTrainer$new()
rst <-RandomSearchCV$new(trainer = rf,
                            parameters = list(n_estimators = c(100,500),
                            max_depth = c(5,2,10,14)),
                            n_folds = 3,
                            scoring = c('accuracy','auc'),
                            n_iter = 4)

## ------------------------------------------------
## Method `RandomSearchCV$fit`
## ------------------------------------------------

rf <- RFTrainer$new()
rst <-RandomSearchCV$new(trainer = rf,
                            parameters = list(n_estimators = c(100,500),
                            max_depth = c(5,2,10,14)),
                            n_folds = 3,
                            scoring = c('accuracy','auc'),
                            n_iter = 4)
data("iris")
rst$fit(iris, "Species")
rst$best_iteration()
```

---

`reg_train` *reg_train*

---

## Description

Training Dataset used for regression examples. In this data set, we have to predict the sale price of the houses.

## Usage

```
reg_train
```

## Format

A data frame with 81 variables

## Source

https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

---

# RFTrainer *Random Forest Trainer*

---

## Description

Trains a random forest model.

## Details

Trains a Random Forest model. A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. This implementation uses ranger R package which provides faster model training.

## Public fields

`n_estimators` the number of trees in the forest, default= 100

`max_features` the number of features to consider when looking for the best split. Possible values are `auto(default)` takes sqrt(num_of_features), `sqrt` same as auto, `log` takes log(num_of_features), `none` takes all features

`max_depth` the maximum depth of each tree

`min_node_size` the minumum number of samples required to split an internal node

`criterion` the function to measure the quality of split. For classification, `gini` is used which is a measure of gini index. For regression, the `variance` of responses is used.

`classification` whether to train for classification (1) or regression (0)

`verbose` show computation status and estimated runtime

`seed` seed value

`class_weights` weights associated with the classes for sampling of training observation

`always_split` vector of feature names to be always used for splitting

`importance` Variable importance mode, one of 'none', 'impurity', 'impurity_corrected', 'permutation'. The 'impurity' measure is the Gini index for classification, the variance of the responses for regression. Defaults to "impurity"

## Methods

### Public methods:

- RFTrainer$new()
- RFTrainer$fit()
- RFTrainer$predict()
- RFTrainer$get_importance()
- RFTrainer$clone()

### Method new():

*Usage:*

```
RFTrainer$new(
  n_estimators,
  max_depth,
  max_features,
  min_node_size,
  classification,
  class_weights,
  always_split,
  verbose,
  save_model,
  seed,
  importance
)
```

*Arguments:*

`n_estimators` integer, the number of trees in the forest, default= 100

`max_depth` integer, the maximum depth of each tree

`max_features` integer, the number of features to consider when looking for the best split. Possible values are `auto(default)` takes sqrt(num_of_features), `sqrt` same as auto, `log` takes log(num_of_features), `none` takes all features

`min_node_size` integer, the minumum number of samples required to split an internal node

`classification` integer, whether to train for classification (1) or regression (0)

`class_weights` weights associated with the classes for sampling of training observation

`always_split` vector of feature names to be always used for splitting

`verbose` logical, show computation status and estimated runtime

`save_model` logical, whether to save model

`seed` integer, seed value

`importance` Variable importance mode, one of 'none', 'impurity', 'impurity_corrected', 'permutation'. The 'impurity' measure is the Gini index for classification, the variance of the responses for regression. Defaults to "impurity"

*Details:* Create a new 'RFTrainer' object.

*Returns:* A 'RFTrainer' object.

*Examples:*

```
data("iris")
bst <- RFTrainer$new(n_estimators=10,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
```

**Method** `fit()`**:**

*Usage:*

```
RFTrainer$fit(X, y)
```

*Arguments:*

`X` data.frame containing train features

`y` character, name of the target variable

*Details:* Trains the random forest model

*Returns:* NULL, trains and saves the model in memory

*Examples:*

```
data("iris")
bst <- RFTrainer$new(n_estimators=10,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
bst$fit(iris, 'Species')
```

**Method** `predict()`**:**

*Usage:*

```
RFTrainer$predict(df)
```

*Arguments:*

`df` data.frame containing test features

*Details:* Return predictions from random forest model

*Returns:* a vector containing predictions

*Examples:*

```
data("iris")
bst <- RFTrainer$new(n_estimators=10,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
bst$fit(iris, 'Species')
predictions <- bst$predict(iris)
```

**Method** `get_importance()`**:**

*Usage:*

```
RFTrainer$get_importance()
```

*Details:* Returns feature importance from the model

*Returns:* a data frame containing feature predictions

*Examples:*

```
data("iris")
bst <- RFTrainer$new(n_estimators=50,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
bst$fit(iris, 'Species')
predictions <- bst$predict(iris)
bst$get_importance()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
RFTrainer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `RFTrainer$new`
## ------------------------------------------------

data("iris")
bst <- RFTrainer$new(n_estimators=10,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)

## ------------------------------------------------
## Method `RFTrainer$fit`
## ------------------------------------------------

data("iris")
bst <- RFTrainer$new(n_estimators=10,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
bst$fit(iris, 'Species')

## ------------------------------------------------
## Method `RFTrainer$predict`
## ------------------------------------------------
```

```
data("iris")
bst <- RFTrainer$new(n_estimators=10,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
bst$fit(iris, 'Species')
predictions <- bst$predict(iris)

## ------------------------------------------------
## Method `RFTrainer$get_importance`
## ------------------------------------------------

data("iris")
bst <- RFTrainer$new(n_estimators=50,
                     max_depth=4,
                     classification=1,
                     seed=42,
                     verbose=TRUE)
bst$fit(iris, 'Species')
predictions <- bst$predict(iris)
bst$get_importance()
```

---

`smoothMean` *smoothMean Calculator*

---

## Description

Calculates target encodings using a smoothing parameter and count of categorical variables. This approach is more robust to possibility of leakage and avoid overfitting.

## Usage

```
smoothMean(
  train_df,
  test_df,
  colname,
  target,
  min_samples_leaf = 1,
  smoothing = 1,
  noise_level = 0
)
```

## Arguments

| | |
|---|---|
| `train_df` | train dataset |
| `test_df` | test dataset |
| `colname` | name of categorical column |

| | |
|---|---|
| target | name of target column |
| min_samples_leaf | minimum samples to take category average into account |
| smoothing | smoothing effect to balance categorical average vs prior |
| noise_level | random noise to add, optional |

## Value

a train and test data table with mean encodings of the target for the given categorical variable

## Examples

```
train <- data.frame(region=c('del','csk','rcb','del','csk','pune','guj','del'),
                    win = c(0,1,1,0,0,1,0,1))
test <- data.frame(region=c('rcb','csk','rcb','del','guj','pune','csk','kol'))

# calculate encodings
all_means <- smoothMean(train_df = train,
                        test_df = test,
                        colname = 'region',
                        target = 'win')
train_mean <- all_means$train
test_mean <- all_means$test
```

---

# TfIdfVectorizer *TfIDF(Term Frequency Inverse Document Frequency) Vectorizer*

---

## Description

Creates a tf-idf matrix

## Details

Given a list of text, it creates a sparse matrix consisting of tf-idf score for tokens from the text.

## Super class

superml::CountVectorizer -> TfIdfVectorizer

## Public fields

sentences a list containing sentences

max_df When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold, value lies between 0 and 1.

min_df When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold, value lies between 0 and 1.

max_features use top features sorted by count to be used in bag of words matrix.

smooth_idf logical, to prevent zero division, adds one to document frequencies, as if an extra document was seen containing every term in the collection exactly once

## Methods

### Public methods:

- TfIdfVectorizer$new()
- TfIdfVectorizer$fit()
- TfIdfVectorizer$fit_transform()
- TfIdfVectorizer$transform()
- TfIdfVectorizer$clone()

### Method new():

*Usage:*

```
TfIdfVectorizer$new(min_df, max_df, max_features, smooth_idf)
```

*Arguments:*

`min_df` numeric, When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold, value lies between 0 and 1.

`max_df` nuemric, When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold, value lies between 0 and 1.

`max_features` integer, use top features sorted by count to be used in bag of words matrix.

`smooth_idf` logical, to prevent zero division, adds one to document frequencies, as if an extra document was seen containing every term in the collection exactly once

*Details:* Create a new 'TfIdfVectorizer' object.

*Returns:* A 'TfIdfVectorizer' object.

*Examples:*

```
TfIdfVectorizer$new(smooth_idf = TRUE, min_df = 0.3)
```

### Method fit():

*Usage:*

```
TfIdfVectorizer$fit(sentences)
```

*Arguments:*

`sentences` a list of text sentences

*Details:* Fits the TfIdfVectorizer model on sentences

*Returns:* NULL

*Examples:*

```
sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
tf = TfIdfVectorizer$new(smooth_idf = TRUE, min_df = 0.3)
tf$fit(sents)
```

### Method fit_transform():

*Usage:*

```
TfIdfVectorizer$fit_transform(sentences)
```

*Arguments:*

`sentences` a list of text sentences

*Details:* Fits the TfIdfVectorizer model and returns a sparse matrix of count of tokens

*Returns:* a sparse matrix containing tf-idf score for tokens in each given sentence

*Examples:*

```
\donttest{
sents <- c('i am alone in dark.','mother_mary a lot',
         'alone in the dark?', 'many mothers in the lot....')
tf <- TfIdfVectorizer$new(smooth_idf = TRUE, min_df = 0.1)
tf_matrix <- tf$fit_transform(sents)
}
```

**Method** `transform()`**:**

*Usage:*

```
TfIdfVectorizer$transform(sentences)
```

*Arguments:*

`sentences` a list of new text sentences

*Details:* Returns a matrix of tf-idf score of tokens

*Returns:* a sparse matrix containing tf-idf score for tokens in each given sentence

*Examples:*

```
\donttest{
sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
new_sents <- c("dark at night",'mothers day')
tf = TfIdfVectorizer$new(min_df=0.1)
tf$fit(sents)
tf_matrix <- tf$transform(new_sents)
}
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
TfIdfVectorizer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `TfIdfVectorizer$new`
## ------------------------------------------------

TfIdfVectorizer$new(smooth_idf = TRUE, min_df = 0.3)

## ------------------------------------------------
## Method `TfIdfVectorizer$fit`
```

```
## ------------------------------------------------

sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
tf = TfIdfVectorizer$new(smooth_idf = TRUE, min_df = 0.3)
tf$fit(sents)

## ------------------------------------------------
## Method `TfIdfVectorizer$fit_transform`
## ------------------------------------------------


sents <- c('i am alone in dark.','mother_mary a lot',
         'alone in the dark?', 'many mothers in the lot....')
tf <- TfIdfVectorizer$new(smooth_idf = TRUE, min_df = 0.1)
tf_matrix <- tf$fit_transform(sents)


## ------------------------------------------------
## Method `TfIdfVectorizer$transform`
## ------------------------------------------------


sents = c('i am alone in dark.','mother_mary a lot',
          'alone in the dark?', 'many mothers in the lot....')
new_sents <- c("dark at night",'mothers day')
tf = TfIdfVectorizer$new(min_df=0.1)
tf$fit(sents)
tf_matrix <- tf$transform(new_sents)
```

---

`XGBTrainer` *Extreme Gradient Boosting Trainer*

---

## Description

Trains a XGBoost model in R

## Details

Trains a Extreme Gradient Boosting Model. XGBoost belongs to a family of boosting algorithms that creates an ensemble of weak learner to learn about data. It is a wrapper for original xgboost R package, you can find the documentation here: http://xgboost.readthedocs.io/en/latest/parameter.html

## Public fields

`booster` the trainer type, the values are `gbtree(default)`, `gblinear`, `dart:gbtree`

`objective` specify the learning task. Check the link above for all possible values.

`nthread` number of parallel threads used to run, default is to run using all threads available

`silent` 0 means printing running messages, 1 means silent mode

`n_estimators` number of trees to grow, default = 100

`learning_rate` Step size shrinkage used in update to prevents overfitting. Lower the learning rate, more time it takes in training, value lies between between 0 and 1. Default = 0.3

`gamma` Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be. Value lies between 0 and infinity, Default = 0

`max_depth` the maximum depth of each tree, default = 6

`min_child_weight` Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger min_child_weight is, the more conservative the algorithm will be. Value lies between 0 and infinity. Default = 1

`subsample` Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration. Value lies between 0 and 1. Default = 1

`colsample_bytree` Subsample ratio of columns when constructing each tree. Subsampling will occur once in every boosting iteration. Value lies between 0 and 1. Default = 1

`lambda` L2 regularization term on weights. Increasing this value will make model more conservative. Default = 1

`alpha` L1 regularization term on weights. Increasing this value will make model more conservative. Default = 0

`eval_metric` Evaluation metrics for validation data, a default metric will be assigned according to objective

`print_every` print training log after n iterations. Default = 50

`feval` custom evaluation function

`early_stopping` Used to prevent overfitting, stops model training after this number of iterations if there is no improvement seen

`maximize` If feval and early_stopping_rounds are set, then this parameter must be set as well. When it is TRUE, it means the larger the evaluation score the better.

`custom_objective` custom objective function

`save_period` when it is non-NULL, model is saved to disk after every save_period rounds, 0 means save at the end.

`save_name` the name or path for periodically saved model file.

`xgb_model` a previously built model to continue the training from. Could be either an object of class xgb.Booster, or its raw data, or the name of a file with a previously saved model.

`callbacks` a list of callback functions to perform various task during boosting. See callbacks. Some of the callbacks are automatically created depending on the parameters' values. User can provide either existing or their own callback methods in order to customize the training process.

`verbose` If 0, xgboost will stay silent. If 1, xgboost will print information of performance. If 2, xgboost will print some additional information. Setting verbose > 0 automatically engages the cb.evaluation.log and cb.print.evaluation callback functions.

`watchlist` what information should be printed when verbose=1 or verbose=2. Watchlist is used to specify validation set monitoring during training. For example user can specify watchlist=list(validation1=mat1, validation2=mat2) to watch the performance of each round's model on mat1 and mat2

`num_class` set number of classes in case of multiclassification problem

`weight` a vector indicating the weight for each row of the input.

`na_missing` by default is set to NA, which means that NA values should be considered as 'missing' by the algorithm. Sometimes, 0 or other extreme value might be used to represent missing values. This parameter is only used when input is a dense matrix.

`feature_names` internal use, stores the feature names for model importance

`cv_model` internal use

## Methods

### Public methods:

- `XGBTrainer$new()`
- `XGBTrainer$cross_val()`
- `XGBTrainer$fit()`
- `XGBTrainer$predict()`
- `XGBTrainer$show_importance()`
- `XGBTrainer$clone()`

### Method `new()`:

*Usage:*

```
XGBTrainer$new(
  booster,
  objective,
  nthread,
  silent,
  n_estimators,
  learning_rate,
  gamma,
  max_depth,
  min_child_weight,
  subsample,
  colsample_bytree,
  lambda,
  alpha,
  eval_metric,
  print_every,
  feval,
  early_stopping,
  maximize,
```

```
  custom_objective,
  save_period,
  save_name,
  xgb_model,
  callbacks,
  verbose,
  num_class,
  weight,
  na_missing
)
```

*Arguments:*

`booster` the trainer type, the values are `gbtree(default)`, `gblinear`, `dart:gbtree`

`objective` specify the learning task. Check the link above for all possible values.

`nthread` number of parallel threads used to run, default is to run using all threads available

`silent` 0 means printing running messages, 1 means silent mode

`n_estimators` number of trees to grow, default = 100

`learning_rate` Step size shrinkage used in update to prevents overfitting. Lower the learning rate, more time it takes in training, value lies between between 0 and 1. Default = 0.3

`gamma` Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be. Value lies between 0 and infinity, Default = 0

`max_depth` the maximum depth of each tree, default = 6

`min_child_weight` Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger min_child_weight is, the more conservative the algorithm will be. Value lies between 0 and infinity. Default = 1

`subsample` Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration. Value lies between 0 and 1. Default = 1

`colsample_bytree` Subsample ratio of columns when constructing each tree. Subsampling will occur once in every boosting iteration. Value lies between 0 and 1. Default = 1

`lambda` L2 regularization term on weights. Increasing this value will make model more conservative. Default = 1

`alpha` L1 regularization term on weights. Increasing this value will make model more conservative. Default = 0

`eval_metric` Evaluation metrics for validation data, a default metric will be assigned according to objective

`print_every` print training log after n iterations. Default = 50

`feval` custom evaluation function

`early_stopping` Used to prevent overfitting, stops model training after this number of iterations if there is no improvement seen

`maximize` If feval and early_stopping_rounds are set, then this parameter must be set as well. When it is TRUE, it means the larger the evaluation score the better.

`custom_objective` custom objective function

`save_period` when it is non-NULL, model is saved to disk after every save_period rounds, 0 means save at the end.

`save_name` the name or path for periodically saved model file.

`xgb_model` a previously built model to continue the training from. Could be either an object of class xgb.Booster, or its raw data, or the name of a file with a previously saved model.

`callbacks` a list of callback functions to perform various task during boosting. See callbacks. Some of the callbacks are automatically created depending on the parameters' values. User can provide either existing or their own callback methods in order to customize the training process.

`verbose` If 0, xgboost will stay silent. If 1, xgboost will print information of performance. If 2, xgboost will print some additional information. Setting verbose > 0 automatically engages the cb.evaluation.log and cb.print.evaluation callback functions.

`num_class` set number of classes in case of multiclassification problem

`weight` a vector indicating the weight for each row of the input.

`na_missing` by default is set to NA, which means that NA values should be considered as 'missing' by the algorithm. Sometimes, 0 or other extreme value might be used to represent missing values. This parameter is only used when input is a dense matrix.

*Details:* Create a new 'XGBTrainer' object.

*Returns:* A 'XGBTrainer' object.

*Examples:*

```
library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
```

**Method** `cross_val()`**:**

*Usage:*

```
XGBTrainer$cross_val(X, y, nfolds = 5, stratified = TRUE, folds = NULL)
```

*Arguments:*

`X` data.frame

`y` character, name of target variable

`nfolds` integer, number of folds

`stratified` logical, whether to use stratified sampling

`folds` the list of CV folds' indices - either those passed through the folds parameter or randomly generated.

*Details:* Trains the xgboost model using cross validation scheme

*Returns:* NULL, trains a model and saves it in memory

*Examples:*

```
library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)

# do cross validation to find optimal value for n_estimators
xgb$cross_val(X = df, y = 'Species',nfolds = 3, stratified = TRUE)
```

**Method** `fit()`**:**

*Usage:*

```
XGBTrainer$fit(X, y, valid = NULL)
```

*Arguments:*

`X` data.frame, training data

`y` character, name of target variable

`valid` data.frame, validation data

*Details:* Fits the xgboost model on given data

*Returns:* NULL, trains a model and keeps it in memory

*Examples:*

```
library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
xgb$fit(df, 'Species')
```

**Method** `predict()`**:**

*Usage:*

```
XGBTrainer$predict(df)
```

*Arguments:*

`df` data.frame, test data set

*Details:* Returns predicted values for a given test data

*Returns:* xgboost predictions

*Examples:*

```
#' library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
xgb$fit(df, 'Species')

# make predictions
preds <- xgb$predict(as.matrix(iris[,1:4]))
```

**Method** `show_importance()`**:**

*Usage:*

```
XGBTrainer$show_importance(type = "plot", topn = 10)
```

*Arguments:*

`type` character, could be 'plot' or 'table'

`topn` integer, top n features to display

*Details:* Shows feature importance plot

*Returns:* a table or a plot of feature importance

*Examples:*

```
library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
xgb$fit(df, 'Species')
xgb$show_importance()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
XGBTrainer$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `XGBTrainer$new`
## ------------------------------------------------

library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)

## ------------------------------------------------
## Method `XGBTrainer$cross_val`
## ------------------------------------------------

library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)

# do cross validation to find optimal value for n_estimators
xgb$cross_val(X = df, y = 'Species',nfolds = 3, stratified = TRUE)

## ------------------------------------------------
## Method `XGBTrainer$fit`
## ------------------------------------------------

library(data.table)
```

```
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
xgb$fit(df, 'Species')

## ------------------------------------------------
## Method `XGBTrainer$predict`
## ------------------------------------------------

#' library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
xgb$fit(df, 'Species')

# make predictions
preds <- xgb$predict(as.matrix(iris[,1:4]))

## ------------------------------------------------
## Method `XGBTrainer$show_importance`
## ------------------------------------------------

library(data.table)
df <- copy(iris)

# convert characters/factors to numeric
df$Species <- as.numeric(as.factor(df$Species))-1

# initialise model
xgb <- XGBTrainer$new(objective = 'multi:softmax',
                      maximize = FALSE,
                      eval_metric = 'merror',
                      num_class=3,
                      n_estimators = 2)
xgb$fit(df, 'Species')
xgb$show_importance()
```

# Index