## Package ‘SSLR’

March 20, 2020

<table>
<thead>
<tr>
<th>Type</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Semi-Supervised Classification and Regression Methods</td>
</tr>
<tr>
<td>Version</td>
<td>0.9.0</td>
</tr>
<tr>
<td>Maintainer</td>
<td>Francisco Jesús Palomares Alabarce &lt;<a href="mailto:fpalomares@correo.ugr.es">fpalomares@correo.ugr.es</a>&gt;</td>
</tr>
<tr>
<td>Description</td>
<td>Providing a collection of techniques for semi-supervised classification and regression. In semi-supervised problem, both labeled and unlabeled data are used to train a classifier. The package includes a collection of semi-supervised learning techniques: self-training, co-training, democratic, decision tree, random forest, ‘S3VM’ ... etc, with a fairly intuitive interface that is easy to use.</td>
</tr>
<tr>
<td>License</td>
<td>GPL-3</td>
</tr>
<tr>
<td>ByteCompile</td>
<td>true</td>
</tr>
<tr>
<td>Depends</td>
<td>R (&gt;= 2.10)</td>
</tr>
<tr>
<td>Encoding</td>
<td>UTF-8</td>
</tr>
<tr>
<td>LazyData</td>
<td>true</td>
</tr>
<tr>
<td>RoxygenNote</td>
<td>7.1.0</td>
</tr>
<tr>
<td>Imports</td>
<td>stats, parsnip, plyr, dplyr (&gt;= 0.8.0.1), magrittr, purrr, rlang (&gt;= 0.3.1), proxy, methods, generics, utils, RANN, foreach, RSSL</td>
</tr>
<tr>
<td>LinkingTo</td>
<td>Rcpp, RcppArmadillo</td>
</tr>
<tr>
<td>Suggests</td>
<td>caret, tidymodels, e1071, C50, kernlab, testthat, doParallel, tidyverse, survival, xgboost, covr, kknn, randomForest, ranger, MASS, nlm, knitr, rmardown</td>
</tr>
<tr>
<td>VignetteBuilder</td>
<td>knitr</td>
</tr>
<tr>
<td>NeedsCompilation</td>
<td>yes</td>
</tr>
<tr>
<td>Author</td>
<td>Francisco Jesús Palomares Alabarce [aut, cre] (<a href="https://orcid.org/0000-0002-0499-7034">https://orcid.org/0000-0002-0499-7034</a>), José Manuel Benítez [ctb] (<a href="https://orcid.org/0000-0002-2346-0793">https://orcid.org/0000-0002-2346-0793</a>), Isaac Triguero [ctb] (<a href="https://orcid.org/0000-0002-0150-0651">https://orcid.org/0000-0002-0150-0651</a>), Christoph Bergmeir [ctb] (<a href="https://orcid.org/0000-0002-3665-9021">https://orcid.org/0000-0002-3665-9021</a>), Mabel González [ctb] (<a href="https://orcid.org/0000-0003-0152-444X">https://orcid.org/0000-0003-0152-444X</a>)</td>
</tr>
</tbody>
</table>
### R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abalone</td>
<td>4</td>
</tr>
<tr>
<td>best_split</td>
<td>4</td>
</tr>
<tr>
<td>best_split,DecisionTreeClassifier-method</td>
<td>5</td>
</tr>
<tr>
<td>breast</td>
<td>5</td>
</tr>
<tr>
<td>calculate_gini</td>
<td>6</td>
</tr>
<tr>
<td>check_value</td>
<td>6</td>
</tr>
<tr>
<td>check_xy_interface</td>
<td>7</td>
</tr>
<tr>
<td>coBC</td>
<td>7</td>
</tr>
<tr>
<td>coBCCombine</td>
<td>9</td>
</tr>
<tr>
<td>coBCG</td>
<td>10</td>
</tr>
<tr>
<td>coBCReg</td>
<td>12</td>
</tr>
<tr>
<td>coBCRegG</td>
<td>13</td>
</tr>
<tr>
<td>coffee</td>
<td>15</td>
</tr>
<tr>
<td>COREG</td>
<td>15</td>
</tr>
<tr>
<td>DecisionTreeClassifier-class</td>
<td>16</td>
</tr>
<tr>
<td>democratic</td>
<td>16</td>
</tr>
<tr>
<td>democraticCombine</td>
<td>18</td>
</tr>
<tr>
<td>democraticG</td>
<td>19</td>
</tr>
<tr>
<td>EMLLeastSquaresClassifierSSLR</td>
<td>20</td>
</tr>
<tr>
<td>EMEarestMeanClassifierSSLR</td>
<td>22</td>
</tr>
<tr>
<td>EntropyRegularizedLogisticRegressionSSLR</td>
<td>23</td>
</tr>
<tr>
<td>fit.model_sslr</td>
<td>24</td>
</tr>
<tr>
<td>fit_decision_tree</td>
<td>25</td>
</tr>
<tr>
<td>fit_decision_tree,DecisionTreeClassifier-method</td>
<td>25</td>
</tr>
<tr>
<td>fit_random_forest,RandomForestSemisupervised-method</td>
<td>26</td>
</tr>
<tr>
<td>fit_xy.model_sslr</td>
<td>27</td>
</tr>
<tr>
<td>fit_x_u</td>
<td>28</td>
</tr>
<tr>
<td>fit_x_u.model_sslr</td>
<td>28</td>
</tr>
<tr>
<td>get_class_max_prob</td>
<td>29</td>
</tr>
<tr>
<td>get_function</td>
<td>29</td>
</tr>
<tr>
<td>get_function_generic</td>
<td>30</td>
</tr>
<tr>
<td>get_levels_categoric</td>
<td>30</td>
</tr>
<tr>
<td>get_most_frequented</td>
<td>31</td>
</tr>
<tr>
<td>get_value_mean</td>
<td>31</td>
</tr>
<tr>
<td>get_x_y</td>
<td>32</td>
</tr>
<tr>
<td>gini_or_variance</td>
<td>32</td>
</tr>
<tr>
<td>gini_prob</td>
<td>33</td>
</tr>
<tr>
<td>grow_tree</td>
<td>33</td>
</tr>
<tr>
<td>grow_tree,DecisionTreeClassifier-method</td>
<td>34</td>
</tr>
<tr>
<td>knn_regression</td>
<td>34</td>
</tr>
<tr>
<td>LaplacianSVMSSLR</td>
<td>35</td>
</tr>
<tr>
<td>LinearTSVMSSLR</td>
<td>36</td>
</tr>
</tbody>
</table>
R topics documented:

- load_parsnip .......................................................... 37
- load_RANN .......................................................... 38
- load_RSSL ............................................................ 38
- MCNearestMeanClassifierSSLR ........................................... 38
- newDecisionTree .................................................... 40
- Node-class .......................................................... 40
- nullOrNumericOrCharacter-class ..................................... 40
- oneNN .............................................................. 41
- predict,DecisionTreeClassifier-method ............................... 41
- predict,RandomForestSemisupervised-method ......................... 42
- predict.coBC ...................................................... 42
- predict.COLREG .................................................... 43
- predict.democratic .................................................. 44
- predict.EMLeastSquaresClassifierSSLR .............................. 44
- predict.EMNearestMeanClassifierSSLR .................................. 45
- predict.EntropyRegularizedLogisticRegressionSSLR ................ 45
- predict.LaplacianSVMSSLR ........................................... 46
- predict.LinearTSVMSSLR ............................................ 46
- predict.MCNearestMeanClassifierSSLR .................................. 47
- predict.model_sslr_fitted ............................................ 47
- predict.OneNN ..................................................... 48
- predict.RandomForestSemisupervised_fitted .......................... 48
- predict.selfTraining ................................................ 49
- predict.setred .................................................... 50
- predict.snnrce ..................................................... 50
- predict.snnrceG ..................................................... 51
- predict.SSLRDecisionTree_fitted .................................... 52
- predict.triTraining ................................................ 52
- predict.TSVMSSLR .................................................. 53
- predict.USMLeastSquaresClassifierSSLR .............................. 53
- predict.WellSVMSSLR ............................................... 54
- predict_inputs ..................................................... 54
- predict_inputs,DecisionTreeClassifier-method ....................... 55
- print.model_sslr ................................................... 55
- RandomForestSemisupervised-class .................................... 55
- selfTraining ....................................................... 56
- selfTrainingG ...................................................... 58
- setred ........................................................... 61
- setredG .......................................................... 63
- snnrce .......................................................... 66
- SSLRDecisionTree .................................................. 68
- SSLRRandomForest .................................................. 69
- train_generic ..................................................... 71
- triTraining ......................................................... 72
- triTrainingCombine .................................................. 73
- triTrainingG ........................................................ 74
- TSVMSSLR .......................................................... 77
- USMLeastSquaresClassifierSSLR ..................................... 78
### Abalone

**Description**

Abalone

**Usage**

```r
data(abalone)
```

**Format**

Predict the age of abalone from physical measurements

**Source**


---

### best_split

**Description**

An S4 method to best split

**Usage**

```r
best_split(object, ...)
```

**Arguments**

- `object` : DecisionTree object
- `...` : This parameter is included for compatibility reasons.
**Description**

Function to get best split in Decision Tree. Find the best split for node. "Best" means that the mean of impurity is the least possible. To find the best division, let’s iterate through all the features. All threshold / feature pairs will be computed in the numerical features. In the features that are not numerical, we get the best group of possible values will be obtained based on an algorithm with the function `get_levels_categorical`.

**Usage**

```r
## S4 method for signature 'DecisionTreeClassifier'
best_split(object, X, y, parms)
```

**Arguments**

- `object`: DecisionTree object
- `X`: is data
- `y`: is class values
- `parms`: parms in function

**Value**

A list with:
- `best_idx`: name of the feature with the best split or Null if it not be found
- `best_thr`: threshold found in the best split, or Null if it not be found

**Description**

Breast

**Usage**

```r
data(breast)
```

**Format**

: Diagnostic Wisconsin Breast Cancer Database

**Source**

calculate_gini  

**Function calculate gini**

**Description**

Function to calculate gini index. Formula is: \(1 - \frac{\text{num_classes}}{\text{sum probability_class}^2}\)

**Usage**

\[\text{calculate_gini(column_factor)}\]

**Arguments**

- column_factor: class values

check_value  

**Check value in leaf**

**Description**

Function to check value in leaf from numeric until character

**Usage**

\[\text{check_value(value, threshold)}\]

**Arguments**

- value: is the value in leaf node
- threshold: in leaf node

**Value**

TRUE if <= in numeric or %in% in factor
check_xy_interface

Description
Check interface

Usage
check_xy_interface(x, y)

Arguments
x  data without class labels
y  values class

coBC
General Interface for CoBC model

Description
Co-Training by Committee (CoBC) is a semi-supervised learning algorithm with a co-training style. This algorithm trains N classifiers with the learning scheme defined in the learner argument using a reduced set of labeled examples. For each iteration, an unlabeled example is labeled for a classifier if the most confident classifications assigned by the other N−1 classifiers agree on the labeling proposed. The unlabeled examples candidates are selected randomly from a pool of size u. The final prediction is the average of the estimates of the N regressors.

Usage
cobC(learner, N = 3, perc.full = 0.7, u = 100, max.iter = 50)

Arguments
learner  model from parsnip package for training a supervised base classifier using a set of instances. This model need to have probability predictions in classification mode
N  The number of classifiers used as committee members. All these classifiers are trained using the gen.learner function. Default is 3.
perc.full  A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-labeling process is stopped. Default is 0.7.
u  Number of unlabeled instances in the pool. Default is 100.
max.iter  Maximum number of iterations to execute in the self-labeling process. Default is 50.
Details

For regression tasks, labeling data is very expensive computationally. Its so slow. This method trains an ensemble of diverse classifiers. To promote the initial diversity the classifiers are trained from the reduced set of labeled examples by Bagging. The stopping criterion is defined through the fulfillment of one of the following criteria: the algorithm reaches the number of iterations defined in the max.iter parameter or the portion of unlabeled set, defined in the perc.full parameter, is moved to the enlarged labeled set of the classifiers.

Value

(When model fit) A list object of class "coBC" containing:

- **model** The final \( N \) base classifiers trained using the enlarged labeled set.
- **model.index** List of \( N \) vectors of indexes related to the training instances used per each classifier. These indexes are relative to the \( y \) argument.
- **instances.index** The indexes of all training instances used to train the \( N \) models. These indexes include the initial labeled instances and the newly labeled instances. These indexes are relative to the \( y \) argument.
- **model.index.map** List of three vectors with the same information in model.index but the indexes are relative to instances.index vector.
- **classes** The levels of \( y \) factor in classification.
- **pred** The function provided in the pred argument.
- **pred.pars** The list provided in the pred.pars argument.

References


Mohamed Farouk Abdel-Hady, Mohamed Farouk Abdel-Hady and Günther Palm. *Semi-supervised Learning for Regression with Cotraining by Committee* Institute of Neural Information Processing University of Ulm D-89069 Ulm, Germany

Examples

```r
library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)
data(wine)
set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[train.index[,]
test <- wine[-train.index[,]
```
coBCCombine

Combining the hypothesis

Description
This function combines the probabilities predicted by the committee of classifiers.

Usage
coBCCombine(h.prob, classes)

Arguments
- h.prob: A list of probability matrices.
- classes: The classes in the same order that appear in the columns of each matrix in h.prob.

Value
A probability matrix
Description

CoBC is a semi-supervised learning algorithm with a co-training style. This algorithm trains \( N \) classifiers with the learning scheme defined in `gen.learner` using a reduced set of labeled examples. For each iteration, an unlabeled example is labeled for a classifier if the most confident classifications assigned by the other \( N-1 \) classifiers agree on the labeling proposed. The unlabeled examples candidates are selected randomly from a pool of size \( u \).

Usage

```r
coBCG(y, gen.learner, gen.pred, N = 3, perc.full = 0.7, u = 100, max.iter = 50)
```

Arguments

- **y**: A vector with the labels of training instances. In this vector the unlabeled instances are specified with the value `NA`.
- **gen.learner**: A function for training \( N \) supervised base classifiers. This function needs two parameters, indexes and cls, where indexes indicates the instances to use and cls specifies the classes of those instances.
- **gen.pred**: A function for predicting the probabilities per classes. This function must be two parameters, model and indexes, where the model is a classifier trained with `gen.learner` function and indexes indicates the instances to predict.
- **N**: The number of classifiers used as committee members. All these classifiers are trained using the `gen.learner` function. Default is 3.
- **perc.full**: A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-labeling process is stopped. Default is 0.7.
- **u**: Number of unlabeled instances in the pool. Default is 100.
- **max.iter**: Maximum number of iterations to execute in the self-labeling process. Default is 50.

Details

`coBCG` can be helpful in those cases where the method selected as base classifier needs a `learner` and `pred` functions with other specifications. For more information about the general `coBC` method, please see `coBC` function. Essentially, `coBC` function is a wrapper of `coBCG` function.

Value

A list object of class "coBCG" containing:

- **model**: The final \( N \) base classifiers trained using the enlarged labeled set.
- **model.index**: List of \( N \) vectors of indexes related to the training instances used per each classifier. These indexes are relative to the `y` argument.
instances.index  The indexes of all training instances used to train the N models. These indexes include the initial labeled instances and the newly labeled instances. These indexes are relative to the y argument.

model.index.map List of three vectors with the same information in model.index but the indexes are relative to instances.index vector.

classes The levels of y factor.

Examples

library(SSLR)
library(caret)
## Load Wine data set
data(wine)

cls <- which(colnames(wine) == "Wine")
x <- wine[, - cls] # instances without classes
y <- wine[, cls] # the classes
x <- scale(x) # scale the attributes

## Prepare data
set.seed(20)
# Use 50% of instances for training
tra.idx <- sample(x = length(y), size = ceiling(length(y) * 0.5))
xtrain <- x[tra.idx,] # training instances
ytrain <- y[tra.idx] # classes of training instances
# Use 70% of train instances as unlabeled set
tra.na.idx <- sample(x = length(tra.idx), size = ceiling(length(tra.idx) * 0.7))
ytrain[tra.na.idx] <- NA # remove class information of unlabeled instances

# Use the other 50% of instances for inductive testing
tst.idx <- setdiff(1:length(y), tra.idx)
xitest <- x[tst.idx,] # testing instances
yitest <- y[tst.idx] # classes of testing instances

## Example: Training from a set of instances with 1-NN (knn3) as base classifier.
gen.learner1 <- function(indexes, cls)
caret::knn3(x = xtrain[indexes,], y = cls, k = 1)
gen.pred1 <- function(model, indexes)
predict(model, xtrain[indexes,])

set.seed(1)
trControl_coBCG <- list(gen.learner = gen.learner1, gen.pred = gen.pred1)
md1 <- train_generic(ytrain, method = "coBCG", trControl = trControl_coBCG)

# Predict probabilities per instances using each model
h.prob <- lapply(
  X = md1$model,
  FUN = function(m) predict(m, xitest)
)
# Combine the predictions
```r
c1 <- coBCCombine(h.prob, md1$classes)
table(c1, yitest)

classificationMatrix(c1, yitest)$overall[1]

## Example: Training from a distance matrix with 1-NN (oneNN) as base classifier.
dtrain <- as.matrix(proxy::dist(x = xtrain, method = "euclidean", by_rows = TRUE))
gen.learner2 <- function(indexes, cls) {
m <- SSLR::oneNN(y = cls)
attr(m, "tra.idxs") <- indexes
m
}
gen.pred2 <- function(model, indexes) {
tra.idxs <- attr(model, "tra.idxs")
d <- dtrain[indexes, tra.idxs]
prob <- predict(model, d, distance.weighting = "none")
prob
}
set.seed(1)
trControl_coBCG2 <- list(gen.learner = gen.learner2, gen.pred = gen.pred2)
md2 <- train_generic(ytrain, method = "coBCG", trControl = trControl_coBCG2)

# Predict probabilities per instances using each model
ditest <- proxy::dist(x = xitest, y = xtrain[md2$instances.index,],
  method = "euclidean", by_rows = TRUE)

h.prob <- list()
ninstances <- nrow(dtrain)
for (i in 1:length(md2$model)) {
m <- md2$model[[i]]
D <- ditest[, md2$model.index.map[[i]]]
  h.prob[[i]] <- predict(m, D)
}

# Combine the predictions
c11 <- coBCCombine(h.prob, md2$classes)
table(c11, yitest)

classificationMatrix(c11, yitest)$overall[1]
```

**General Interface coBCReg model**
Description

cobCReg is based on an ensemble of N diverse regressors. At each iteration and for each regressor, the companion committee labels the unlabeled examples then the regressor selects the most informative newly-labeled examples for itself, where the selection confidence is based on estimating the validation error. The final prediction is the average of the estimates of the N regressors.

Usage

cobCReg(learner, N = 3, perc.full = 0.7, u = 100, max.iter = 50)

Arguments

learner model from parsnip package for training a supervised base classifier using a set of instances. This model needs to have probability predictions

N The number of classifiers used as committee members. All these classifiers are trained using the gen.learner function. Default is 3.

perc.full A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-labeling process is stopped. Default is 0.7.

u Number of unlabeled instances in the pool. Default is 100.

max.iter Maximum number of iterations to execute in the self-labeling process. Default is 50.

Details

For regression tasks, labeling data is very expensive computationally. It’s so slow.

References

Mohamed Farouk Abdel-Hady, Mohamed Farouk Abdel-Hady and Günther Palm.
Semi-supervised Learning for Regression with Cotraining by Committee
Institute of Neural Information Processing University of Ulm D-89069 Ulm, Germany
Usage

```r
coBCRegG(
  y,
  gen.learner,
  gen.pred,
  N = 3,
  perc.full = 0.7,
  u = 100,
  max.iter = 50,
  gr = 1
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y</code></td>
<td>A vector with the labels of training instances. In this vector the unlabeled instances are specified with the value <code>NA</code>.</td>
</tr>
<tr>
<td><code>gen.learner</code></td>
<td>A function for training N supervised base classifiers. This function needs two parameters, indexes and cls, where indexes indicates the instances to use and cls specifies the classes of those instances.</td>
</tr>
<tr>
<td><code>gen.pred</code></td>
<td>A function for predicting the probabilities per classes. This function must be two parameters, model and indexes, where the model is a classifier trained with <code>gen.learner</code> function and indexes indicates the instances to predict.</td>
</tr>
<tr>
<td><code>N</code></td>
<td>The number of classifiers used as committee members. All these classifiers are trained using the <code>gen.learner</code> function. Default is 3.</td>
</tr>
<tr>
<td><code>perc.full</code></td>
<td>A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-labeling process is stopped. Default is 0.7.</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Number of unlabeled instances in the pool. Default is 100.</td>
</tr>
<tr>
<td><code>max.iter</code></td>
<td>Maximum number of iterations to execute in the self-labeling process. Default is 50.</td>
</tr>
<tr>
<td><code>gr</code></td>
<td>growing rate</td>
</tr>
</tbody>
</table>

Details

For regression tasks, labeling data is very expensive computationally. Its so slow.

References

Mohamed Farouk Abdel-Hady, Mohamed Farouk Abdel-Hady and Günther Palm.
*Semi-supervised Learning for Regression with Cotraining by Committee*
Institute of Neural Information Processing University of Ulm D-89069 Ulm, Germany
coffee

Time series data set

Description
A dataset containing 56 times series z-normalized. Time series length is 286.

Usage
data(coffee)

Format
A data frame with 56 rows and 287 variables including the class.

Source

COREG

General Interface for COREG model

Description
COREG is a semi-supervised learning for regression with a co-training style. This technique uses two kNN regressors with different distance metrics. For each iteration, each regressor labels the unlabeled example which can be most confidently labeled for the other learner, where the labeling confidence is estimated through considering the consistency of the regressor with the labeled example set. The final prediction is made by averaging the predictions of both the refined kNN regressors

Usage
COREG(max.iter = 50, k1 = 3, k2 = 5, p1 = 3, p2 = 5, u = 100)

Arguments
max.iter maximum number of iterations to execute the self-labeling process. Default is 50.
k1 parameter in first KNN
k2 parameter in second KNN
p1 distance order 1. Default is 3
p2 distance order 1. Default is 5
u Number of unlabeled instances in the pool. Default is 100.
Details

labeling data is very expensive computationally. It's so slow. For executing this model, we need RANN installed.

References

Zhi-Hua Zhou and Ming Li. 
Semi-Supervised Regression with Co-Training. 
National Laboratory for Novel Software Technology Nanjing University, Nanjing 210093, China

Examples

library(SSLR)

m <- COREG(max.iter = 1)

--------------

DecisionTreeClassifier-class

Class DecisionTreeClassifier

Description

Class DecisionTreeClassifier Slots: max_depth, n_classes, n_features, tree, classes, min_samples_split, min_samples_leaf

--------------

democratic

General Interface for Democratic model

Description

Democratic Co-Learning is a semi-supervised learning algorithm with a co-training style. This algorithm trains N classifiers with different learning schemes defined in list gen.learners. During the iterative process, the multiple classifiers with different inductive biases label data for each other.

Usage

democratic(learners, schemes = NULL)

Arguments

learners List of models from parsnip package for training a supervised base classifier using a set of instances. This model need to have probability predictions

schemes List of schemes (col x names in each learner). Default is null, it means that learner uses all x columns
Details

This method trains an ensemble of diverse classifiers. To promote the initial diversity the classifiers must represent different learning schemes. When \(x\text{.inst}\) is FALSE all learners defined must be able to learn a classifier from the precomputed matrix in \(x\). The iteration process of the algorithm ends when no changes occurs in any model during a complete iteration. The generation of the final hypothesis is produced via a weighted majority voting.

Value

(When model fit) A list object of class "democratic" containing:

- \(W\) A vector with the confidence-weighted vote assigned to each classifier.
- \(\text{model}\) A list with the final N base classifiers trained using the enlarged labeled set.
- \(\text{model.index}\) List of N vectors of indexes related to the training instances used per each classifier. These indexes are relative to the \(y\) argument.
- \(\text{instances.index}\) The indexes of all training instances used to train the N models. These indexes include the initial labeled instances and the newly labeled instances. These indexes are relative to the \(y\) argument.
- \(\text{model.index.map}\) List of three vectors with the same information in \(\text{model.index}\) but the indexes are relative to \(\text{instances.index}\) vector.
- \(\text{classes}\) The levels of \(y\) factor.
- \(\text{preds}\) The functions provided in the \(\text{preds}\) argument.
- \(\text{preds.pars}\) The set of lists provided in the \(\text{preds.pars}\) argument.
- \(x\text{.inst}\) The value provided in the \(x\text{.inst}\) argument.

Examples

library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)

data(wine)

set.seed(1)
train.index <- createDataPartition(wine$Wine, p = 0.7, list = FALSE)
train <- wine[train.index,]
test <- wine[-train.index,]

cls <- which(colnames(wine) == "Wine")

#% LABELED
labeled.index <- createDataPartition(wine$Wine, p = 0.2, list = FALSE)
train[-labeled.index,cls] <- NA

#We need a model with probability predictions from parsnip
#https://tidymodels.github.io/parsnip/articles/articles/Models.html
#It should be with mode = classification
democraticCombine

Combining the hypothesis of the classifiers

Description
This function combines the probabilities predicted by the set of classifiers.

Usage

democraticCombine(pred, W, classes)

Arguments

pred
A list with the prediction for each classifier.
W
A vector with the confidence-weighted vote assigned to each classifier during the training process.
classes
the classes.
democraticG

Value

The classification proposed.

democraticG  Democratic generic method

Description

Democratic is a semi-supervised learning algorithm with a co-training style. This algorithm trains N classifiers with different learning schemes defined in list gen.learners. During the iterative process, the multiple classifiers with different inductive biases label data for each other.

Usage

democraticG(y, gen.learners, gen.preds)

Arguments

y  A vector with the labels of training instances. In this vector the unlabeled instances are specified with the value NA.

gen.learners  A list of functions for training N different supervised base classifiers. Each function needs two parameters, indexes and cls, where indexes indicates the instances to use and cls specifies the classes of those instances.

gen.preds  A list of functions for predicting the probabilities per classes. Each function must be two parameters, model and indexes, where the model is a classifier trained with gen.learner function and indexes indicates the instances to predict.

Details

democraticG can be helpful in those cases where the method selected as base classifier needs a learner and pred functions with other specifications. For more information about the general democratic method, please see democratic function. Essentially, democratic function is a wrapper of democraticG function.

Value

A list object of class "democraticG" containing:

W  A vector with the confidence-weighted vote assigned to each classifier.

model  A list with the final N base classifiers trained using the enlarged labeled set.

model.index  List of N vectors of indexes related to the training instances used per each classifier. These indexes are relative to the y argument.

instances.index  The indexes of all training instances used to train the N models. These indexes include the initial labeled instances and the newly labeled instances. These indexes are relative to the y argument.
model.index.map  List of three vectors with the same information in model.index but the indexes are relative to instances.index vector.

classes  The levels of y factor.

References

Yan Zhou and Sally Goldman.
Democratic co-learning.
In IEEE 16th International Conference on Tools with Artificial Intelligence (ICTAI), pages 594-602.

---

EMLeastSquaresClassifierSSLR

*General Interface for EMLeastSquaresClassifier model*

---

**Description**

model from RSSL package

An Expectation Maximization like approach to Semi-Supervised Least Squares Classification

As studied in Krijthe & Loog (2016), minimizes the total loss of the labeled and unlabeled objects by finding the weight vector and labels that minimize the total loss. The algorithm proceeds similar to EM, by subsequently applying a weight update and a soft labeling of the unlabeled objects. This is repeated until convergence.

By default (method="block") the weights of the classifier are updated, after which the unknown labels are updated. method="simple" uses LBFGS to do this update simultaneously. Objective="responsibility" corresponds to the responsibility based, instead of the label based, objective function in Krijthe & Loog (2016), which is equivalent to hard-label self-learning.

**Usage**

```r
EMLeastSquaresClassifierSSLR(
  x_center = FALSE,
  scale = FALSE,
  verbose = FALSE,
  intercept = TRUE,
  lambda = 0,
  eps = 1e-09,
  y_scale = FALSE,
  alpha = 1,
  beta = 1,
  init = "supervised",
  method = "block",
  objective = "label",
  save_all = FALSE,
  max_iter = 1000
)
```
EMLeastSquaresClassifierSSLR

Arguments

- **x_center**: logical; Should the features be centered?
- **scale**: Should the features be normalized? (default: FALSE)
- **verbose**: logical; Controls the verbosity of the output
- **intercept**: logical; Whether an intercept should be included
- **lambda**: numeric; L2 regularization parameter
- **eps**: Stopping criterion for the minimization
- **y_scale**: logical; whether the target vector should be centered
- **alpha**: numeric; the mixture of the new responsibilities and the old in each iteration of the algorithm (default: 1)
- **beta**: numeric; value between 0 and 1 that determines how much to move to the new solution from the old solution at each step of the block gradient descent
- **init**: objective character; "random" for random initialization of labels, "supervised" to use supervised solution as initialization or a numeric vector with a coefficient vector to use to calculate the initialization
- **method**: character; one of "block", for block gradient descent or "simple" for LBFGS optimization (default="block")
- **objective**: character; "responsibility" for hard label self-learning or "label" for soft-label self-learning
- **save_all**: logical; saves all classifiers trained during block gradient descent
- **max_iter**: integer; maximum number of iterations

References


Examples

```r
library(tidyverse)
#' \donttest{
library(tidymodels)
library(caret)
library(SSLR)

data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,]
test <- breast[-train.index,]

cls <- which(colnames(breast) == "Class")

#' LABLED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
```
train[-labeled.index,cls] <- NA

m <- EMLeastSquaresClassifierSSLR() %>% fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Class", estimate = .pred_class)
#
}

EMNearestMeanClassifierSSLR

*General Interface for EMNearestMeanClassifier model*

**Description**

- model from RSSL package Semi-Supervised Nearest Mean Classifier using Expectation Maximization
- Expectation Maximization applied to the nearest mean classifier assuming Gaussian classes with a spherical covariance matrix.
- Starting from the supervised solution, uses the Expectation Maximization algorithm (see Dempster et al. (1977)) to iteratively update the means and shared covariance of the classes (Maximization step) and updates the responsibilities for the unlabeled objects (Expectation step).

**Usage**

```r
EMNearestMeanClassifierSSLR(method = "EM", scale = FALSE, eps = 1e-04)
```

**Arguments**

- **method**: character; Currently only "EM"
- **scale**: Should the features be normalized? (default: FALSE)
- **eps**: Stopping criterion for the maximinimization

**References**


**Examples**

```r
library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)
```
```r
data(breast)
set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,]
test <- breast[-train.index,]

cls <- which(colnames(breast) == "Class")

#% LABELED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- EMNearestMeanClassifierSSLR() %>% fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Class", estimate = .pred_class)
```

---

**EntropyRegularizedLogisticRegressionSSLR**

*General Interface for EntropyRegularizedLogisticRegression model*

**Description**

Model from RSSL package R Implementation of entropy regularized logistic regression implementation as proposed by Grandvalet & Bengio (2005). An extra term is added to the objective function of logistic regression that penalizes the entropy of the posterior measured on the unlabeled examples.

**Usage**

```r
EntropyRegularizedLogisticRegressionSSLR(
  lambda = 0,
  lambda_entropy = 1,
  intercept = TRUE,
  init = NA,
  scale = FALSE,
  x_center = FALSE
)
```

**Arguments**

- `lambda`: L2 Regularization
- `lambda_entropy`: Weight of the labeled observations compared to the unlabeled observations
- `intercept`: logical; Whether an intercept should be included
init        Initial parameters for the gradient descent
scale       logical; Should the features be normalized? (default: FALSE)
x_center    logical; Should the features be centered?

References


Examples

library(tidyverse)
library(caret)
library(tidymodels)
library(SSLR)

data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,]
test <- breast[-train.index,]

cls <- which(colnames(breast) == "Class")

#% LABELED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- EntropyRegularizedLogisticRegressionSSLR() %>% fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Class", estimate = .pred_class)

fit.model_sslr     Fit with formula and data

Description

Funtion to fit through the formula

Usage

## S3 method for class 'model_sslr'
fit(object, formula = NULL, data = NULL, ...)


Arguments

object is the model
formula is the formula
data is the total data train
... unused in this case

Description

An S4 method to fit decision tree.

Usage

fit_decision_tree(object, ...)

Arguments

object DecisionTree object
... This parameter is included for compatibility reasons.

Description

method in class DecisionTreeClassifier used to build a Decision Tree

Usage

# S4 method for signature 'DecisionTreeClassifier'
fit_decision_tree(
    object,
    X,
    y,
    min_samples_split = 20,
    min_samples_leaf = ceiling(min_samples_split/3),
    w = 0.5
)
Arguments

object: A RandomForestSemisupervised object
X: A object that can be coerced as data.frame. Training instances
y: A vector with the labels of the training instances. In this vector the unlabeled instances are specified with the value NA.
min_samples_split: the minimum number of observations to do split
min_samples_leaf: the minimum number of any terminal leaf node
w: weight parameter ranging from 0 to 1

fit_random_forest, RandomForestSemisupervised-method

Fit Random Forest

Description

method in class RandomForestSemisupervised used to build a Decision Tree

Usage

## S4 method for signature 'RandomForestSemisupervised'
fit_random_forest(
  object,
  X,
  y,
  mtry = 2,
  trees = 500,
  min_n = 2,
  w = 0.5,
  replace = TRUE,
  tree_max_depth = Inf,
  sampsize = if (replace) nrow(X) else ceiling(0.632 * nrow(X)),
  min_samples_leaf = if (!is.null(y) && !is.factor(y)) 5 else 1,
  allowParallel = TRUE
)

Arguments

object: A RandomForestSemisupervised object
X: A object that can be coerced as data.frame. Training instances
y: A vector with the labels of the training instances. In this vector the unlabeled instances are specified with the value NA.
mtry: number of features in each decision tree
**Description**

Function to fit with x and y

**Usage**

```r
## S3 method for class 'model_sslr'
fit_xy(object, x = NULL, y = NULL, ...)
```

**Arguments**

- `object` is the model
- `x` is a data frame or matrix with train dataset without objective feature. X have labeled and unlabeled data
- `y` is objective feature with labeled values and NA values in unlabeled data
- `...` unused in this case
**Description**

`fit_x_u`

**Usage**

```r
fit_x_u(object, ...) 
```

**Arguments**

- `object`: object
- `...`: other parameters to be passed

---

**Description**

`fit_x_u.model_sslr`  
*Fit with $x$, $y$ (labeled data) and unlabeled data ($x_U$)*

**Usage**

```r
## S3 method for class 'model_sslr'
fit_x_u.model_sslr(object, x = NULL, y = NULL, x_U = NULL, ...) 
```

**Arguments**

- `object`: is the model
- `x`: is a data frame or matrix with train dataset without objective feature. X only have labeled data
- `y`: is objective feature with labeled values
- `x_U`: train unlabeled data without objective feature
- `...`: This parameter is included for compatibility reasons.
**get_class_max_prob**

Get most frequented

**Description**

Get value most frequented in vector. Used in predictions. It calls a predict with type = "prob" in Decision Tree

**Usage**

```r
get_class_max_prob(trees, input)
```

**Arguments**

- `trees` - trees list
- `input` - is input to be predicted

**get_function**

FUNCTION TO GET FUNCTION METHOD

**Description**

FUNCTION TO GET FUNCTION METHOD SPECIFIC

**Usage**

```r
get_function(met)
```

**Arguments**

- `met` - character

**Value**

`method_train` (function)
### get_function_generic

**FUNCTION TO GET FUNCTION METHOD GENERIC**

**Description**

FUNCTION TO GET FUNCTION METHOD GENERIC

**Usage**

get_function_generic(met)

**Arguments**

- `met` character

**Value**

method_train (function)

---

### get_levels_categoric

**Function to get group from gini index**

**Description**

Function to get group from gini index. Used in categorical variable From: https://freakonometrics.hypotheses.org/20736

**Usage**

get_levels_categoric(column, Y)

**Arguments**

- `column` is the column
- `Y` values
get_most_frequented

Description
Get value most frequented in vector Used in predictions

Usage
get_most_frequented(elements)

Arguments
- elements vector with values

get_value_mean

Description
Get value most frequented in vector Used in predictions. It calls a predict with type = "numeric" in Decision Tree

Usage
get_value_mean(trees, input)

Arguments
- trees trees list
- input is input to be predicted
get_x_y  

FUNCTION TO GET REAL X AND Y WITH FORMULA AND DATA

Description

FUNCTION TO GET REAL X AND Y WITH FORMULA AND DATA

Usage

get_x_y(form, data)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>form</td>
<td>formula</td>
</tr>
<tr>
<td>data</td>
<td>data values, matrix, dataframe..</td>
</tr>
</tbody>
</table>

Value

x (matrix, dataframe...) and y(factor)

---

gini_or_variance  

Gini or Variance by column

Description

function used to calculate the gini coefficient or variance according to the type of the column. This function is called for the creation of the decision tree

Usage

gini_or_variance(X)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>column to calculate variance or gini</td>
</tr>
</tbody>
</table>
**gini_prob**

Function to compute Gini index

From: https://freakonometrics.hypotheses.org/20736

**Usage**

```r
gini_prob(y, classe)
```

**Arguments**

- `y`: values
- `classe`: classes

---

**grow_tree**

An S4 method to grow tree.

**Description**

An S4 method to grow tree.

**Usage**

```r
grow_tree(object, ...)```

**Arguments**

- `object`: DecisionTree object
- `...`: This parameter is included for compatibility reasons.
Description

Function to grow tree in Decision Tree

Usage

## S4 method for signature 'DecisionTreeClassifier'

\texttt{grow\_tree(object, X, y, parms, depth = 0)}

Arguments

- \texttt{object} \hspace{1cm} \text{DecisionTree instance}
- \texttt{X} \hspace{1cm} \text{data values}
- \texttt{y} \hspace{1cm} \text{classes}
- \texttt{parms} \hspace{1cm} \text{parameters for grow tree}
- \texttt{depth} \hspace{1cm} \text{depth in tree}

Description

create model knn

Usage

\texttt{knn\_regression(k, x, y, p)}

Arguments

- \texttt{k} \hspace{1cm} \text{parameter in KNN model}
- \texttt{x} \hspace{1cm} \text{data}
- \texttt{y} \hspace{1cm} \text{vector labeled data}
- \texttt{p} \hspace{1cm} \text{distance order}
LaplacianSVMSSLR

General Interface for LaplacianSVM model

Description

model from RSSL package Manifold regularization applied to the support vector machine as proposed in Belkin et al. (2006). As an adjacency matrix, we use the k nearest neighbour graph based on a chosen distance (default: euclidean).

Usage

LaplacianSVMSSLR(
  lambda = 1,
  gamma = 1,
  scale = TRUE,
  kernel = kernlab::vanilladot(),
  adjacency_distance = "euclidean",
  adjacency_k = 6,
  normalized_laplacian = FALSE,
  eps = 1e-09
)

Arguments

lambda numeric; L2 regularization parameter

gamma numeric; Weight of the unlabeled data

scale logical; Should the features be normalized? (default: FALSE)

kernel kernlab::kernel to use

adjacency_distance character; distance metric used to construct adjacency graph from the dist function. Default: "euclidean"

adjacency_k integer; Number of of neighbours used to construct adjacency graph.

normalized_laplacian logical; If TRUE use the normalized Laplacian, otherwise, the Laplacian is used

eps numeric; Small value to ensure positive definiteness of the matrix in the QP formulation

References

**Examples**

```r
library(tidyverse)
library(caret)
library(tidymodels)
library(SSLR)

data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[ train.index,]
test <- breast[-train.index,]

cls <- which(colnames(breast) == "Class")

#% LABELED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

library(kernlab)
m <- LaplacianSVMSSLR(kernel=kernlab::vanilladot()) %>%
  fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Class", estimate = .pred_class)
```

---

**LinearTSVMSSLR**

*General Interface for LinearTSVM model*

**Description**

model from RSSL package Implementation of the Linear Support Vector Classifier. Can be solved in the Dual formulation, which is equivalent to SVM or the Primal formulation.

**Usage**

```r
LinearTSVMSSLR(
  C = 1,
  Cstar = 0.1,
  s = 0,
  x_center = FALSE,
  scale = FALSE,
  eps = 1e-06,
  verbose = FALSE,
  init = NULL
)
```
Arguments

- **C**: Cost variable
- **Cstar**: numeric; Cost parameter of the unlabeled objects
- **s**: numeric; Parameter controlling the loss function of the unlabeled objects
- **x_center**: logical; Should the features be centered?
- **scale**: Whether a z-transform should be applied (default: TRUE)
- **eps**: Small value to ensure positive definiteness of the matrix in QP formulation
- **verbose**: logical; Controls the verbosity of the output
- **init**: numeric; Initial classifier parameters to start the convex concave procedure

Examples

```r
library(tidyverse)
library(caret)
library(tidymodels)
library(SSLR)

data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,]

m <- LinearTSVMSSLR() %>% fit(Class ~ ., data = train)

# Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Class", estimate = .pred_class)
```

Description

function to load parsnip package
Usage

load_parsnip()

----

load_RANN

Load parsnip

----

Description

function to load parsnip package

Usage

load_RANN()

---

load_RSSL

Load RSSL

---

Description

function to load RSSL package

Usage

load_RSSL()

---

**MCNearestMeanClassifierSSLR**

*General Interface for MCNearestMeanClassifier (Moment Constrained Semi-supervised Nearest Mean Classifier) model*

Description

model from RSSL package Update the means based on the moment constraints as defined in Loog (2010). The means estimated using the labeled data are updated by making sure their weighted mean corresponds to the overall mean on all (labeled and unlabeled) data. Optionally, the estimated variance of the classes can be re-estimated after this update is applied by setting update_sigma to TRUE. To get the true nearest mean classifier, rather than estimate the class priors, set them to equal priors using, for instance prior=matrix(0.5,2).
Usage

MCNearestMeanClassifierSSLR(
    update_sigma = FALSE,
    prior = NULL,
    x_center = FALSE,
    scale = FALSE
)

Arguments

update_sigma logical; Whether the estimate of the variance should be updated after the means have been updated using the unlabeled data
prior matrix; Class priors for the classes
x_center logical; Should the features be centered?
scale logical; Should the features be normalized? (default: FALSE)

References


Examples

library(tidyverse)
library(caret)
library(tidymodels)
library(SSLR)
data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,]
test <- breast[-train.index,]
cls <- which(colnames(breast) == "Class")

#% LABELED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- MCNearestMeanClassifierSSLR() %>% fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
microbenchmark::microbenchmark(metrics(truth = "Class", estimate = .pred_class))
newDecisionTree  
*Function to create DecisionTree*

**Description**

Function to create DecisionTree

**Usage**

newDecisionTree(max_depth)

**Arguments**

max_depth  
max depth in tree

**Node-class**  
*Class Node for Decision Tree*

**Description**

Class Node for Decision Tree Slots: gini, num_samples, num_samples_per_class, predicted_class_value, feature_index threshold, left, right, probabilities

**nullOrNumericOrCharacter-class**  
*An S4 class to represent a class with more types values: null, numeric or character*

**Description**

An S4 class to represent a class with more types values: null, numeric or character
Description

Build a model using the given data to be able to predict the label or the probabilities of other instances, according to 1-NN algorithm.

Usage

oneNN(x = NULL, y)

Arguments

x
This argument is not used, the reason why he gets is to fulfill an agreement

y
a vector with the labels of training instances

Value

A model with the data needed to use 1-NN

predict,DecisionTreeClassifier-method

Function to predict inputs in Decision Tree

Description

Function to predict inputs in Decision Tree

Usage

## S4 method for signature 'DecisionTreeClassifier'
predict(object, inputs, type = "class")

Arguments

object
The Decision Tree object

inputs
data to be predicted

type
Is param to define the type of predict. It can be "class", to get class labels Or "prob" to get probabilites for class in each input. Default is "class"
**predict.coBC**

**Predictions of the coBC method**

**Description**

Predicts the label of instances according to the coBC model.

**Usage**

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

**predict, RandomForestSemisupervised-method**

*Function to predict inputs in Decision Tree*

**Description**

Function to predict inputs in Decision Tree

**Usage**

```r
## S4 method for signature 'RandomForestSemisupervised'
predict(
  object,
  inputs,
  type = "class",
  confident = "max_prob",
  allowParallel = TRUE
)
```

**Arguments**

- `object`: The Decision Tree object
- `inputs`: data to be predicted
- `type`: class raw
- `confident`: Is param to define the type of predict. It can be "max_prob", to get class with sum of probability is the maximum Or "vote" to get the most frequented class in all trees. Default is "max_prob"
- `allowParallel`: Execute Random Forest in parallel if doParallel is loaded.
predict.COREG

Arguments

object  coBC model built with the coBC function.
x      An object that can be coerced to a matrix. Depending on how the model was built, x is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.
...
      This parameter is included for compatibility reasons.

Details

For additional help see coBC examples.

Value

Vector with the labels assigned.

Description

Predicts the label of instances according to the COREG model.

Usage

## S3 method for class 'COREG'
predict(object, x, type = "numeric", ...)

Arguments

object      Self-training model built with the COREG function.
x          A object that is data
type       of predict in principal model (numeric)
...
      This parameter is included for compatibility reasons.

Details

For additional help see COREG examples.

Value

Vector with the labels assigned (numeric).
### Predictions of the Democratic method

**Description**

Predicts the label of instances according to the democratic model.

**Usage**

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

**Arguments**

- `object` Democratic model built with the `democratic` function.
- `x` A object that can be coerced as matrix. Depending on how was the model built, `x` is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.
- `...` This parameter is included for compatibility reasons.

**Details**

For additional help see `democratic` examples.

**Value**

Vector with the labels assigned.

---

### Predict EMLeastSquaresClassifierSSLR

**Description**

Predict EMLeastSquaresClassifierSSLR

**Usage**

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

**Arguments**

- `object` is the object
- `x` is the dataset
- `...` This parameter is included for compatibility reasons.
predict.EMNearestMeanClassifierSSLR

\textit{Predict EMNearestMeanClassifierSSLR}

\section*{Description}

Predict EMNearestMeanClassifierSSLR

\section*{Usage}

\begin{verbatim}
## S3 method for class 'EMNearestMeanClassifierSSLR'
predict(object, x, ...)
\end{verbatim}

\section*{Arguments}

\begin{itemize}
  \item \textit{object} is the object
  \item \textit{x} is the dataset
  \item \textit{...} This parameter is included for compatibility reasons.
\end{itemize}

\bigskip

\section*{predict.EntropyRegularizedLogisticRegressionSSLR}

\textit{Predict EntropyRegularizedLogisticRegressionSSLR}

\section*{Description}

Predict EntropyRegularizedLogisticRegressionSSLR

\section*{Usage}

\begin{verbatim}
## S3 method for class 'EntropyRegularizedLogisticRegressionSSLR'
predict(object, x, ...)
\end{verbatim}

\section*{Arguments}

\begin{itemize}
  \item \textit{object} is the object
  \item \textit{x} is the dataset
  \item \textit{...} This parameter is included for compatibility reasons.
\end{itemize}
predict.LaplacianSVMSSLR

*Predict LaplacianSVMSSLR*

Description

Predict LaplacianSVMSSLR

Usage

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

Arguments

- `object` is the object
- `x` is the dataset
- `...` This parameter is included for compatibility reasons.

predict.LinearTSVMSSLR

*Predict LinearTSVMSSLR*

Description

Predict LinearTSVMSSLR

Usage

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

Arguments

- `object` is the object
- `x` is the dataset
- `...` This parameter is included for compatibility reasons.
**predict.MCNearestMeanClassifierSSLR**

*Predict MCNearestMeanClassifierSSLR*

**Description**

Predict MCNearestMeanClassifierSSLR

**Usage**

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

**Arguments**

- `object` is the object
- `x` is the dataset
- `...` This parameter is included for compatibility reasons.

**predict.model_sslr_fitted**

*Predictions of model_sslr_fitted class*

**Description**

Predicts from model. There are different types: class, prob, raw class returns tibble with one column
prob returns tibble with probabilities class columns raw returns factor or numeric values

**Usage**

```r
## S3 method for class 'model_sslr_fitted'
predict(object, x, type = NULL, ...)  
```

**Arguments**

- `object` model_sslr_fitted model built.
- `x` A object that can be coerced as matrix. Depending on how was the model built, 
  `x` is interpreted as a matrix with the distances between the unseen instances and 
  the selected training instances, or a matrix of instances.
- `type` of predict in principal model: class, raw, prob, vote, max_prob, numeric
- `...` This parameter is included for compatibility reasons.

**Value**

tibble or vector.
predict.RandomForestSemisupervised_fitted

Predictions of the SSLRDecisionTree_fitted method

Description

Predicts the label of instances according to the RandomForestSemisupervised_fitted model.

Usage

## S3 method for class 'RandomForestSemisupervised_fitted'
predict(object, x, type = "class", confident = "max_prob", ...)

--

predict.OneNN

Model Predictions

Description

This function predicts the class label of instances or its probability of pertaining to each class based on the distance matrix.

Usage

## S3 method for class 'OneNN'
predict(object, dists, type = "prob", ...)

Arguments

object A model of class OneNN built with oneNN
dists A matrix of distances between the instances to classify (by rows) and the instances used to train the model (by column)
type A string that can take two values: "class" for computing the class of the instances or "prob" for computing the probabilities of belonging to each class.
... Currently not used.

Value

If type is equal to "class" a vector of length equal to the rows number of matrix dists, containing the predicted labels. If type is equal to "prob" it returns a matrix which has nrow(dists) rows and a column for every class, where each cell represents the probability that the instance belongs to the class, according to 1NN.
**predict.selfTraining**  

*Arguments*

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>RandomForestSemisupervised_fitted.</td>
</tr>
<tr>
<td>x</td>
<td>A object that can be coerced as matrix. Depending on how was the model built, x is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.</td>
</tr>
<tr>
<td>type</td>
<td>of predict in principal model</td>
</tr>
<tr>
<td>confident</td>
<td>Is param to define the type of predict. It can be &quot;max_prob&quot;, to get class with sum of probability is the maximum Or &quot;vote&quot; to get the most frequented class in all trees. Default is &quot;max_prob&quot;</td>
</tr>
<tr>
<td>...</td>
<td>This parameter is included for compatibility reasons.</td>
</tr>
</tbody>
</table>

*Value*

Vector with the labels assigned.

---

**predict.selfTraining  Predictions of the Self-training method**

**Description**

Predicts the label of instances according to the selfTraining model.

**Usage**

```r
## S3 method for class 'selfTraining'
predict(object, x, type = "class", ...)
```

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Self-training model built with the selfTraining function.</td>
</tr>
<tr>
<td>x</td>
<td>A object that can be coerced as matrix. Depending on how was the model built, x is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.</td>
</tr>
<tr>
<td>type</td>
<td>of predict in principal model</td>
</tr>
<tr>
<td>...</td>
<td>This parameter is included for compatibility reasons.</td>
</tr>
</tbody>
</table>

**Details**

For additional help see selfTraining examples.

**Value**

Vector with the labels assigned.
### predict.setred

**Predictions of the SETRED method**

**Description**

Predicts the label of instances according to the setred model.

**Usage**

```r
## S3 method for class 'setred'
predict(object, x, col_name = ".pred_class", ...)
```

**Arguments**

- `object`: SETRED model built with the `setred` function.
- `x`: A object that can be coerced as matrix. Depending on how was the model built, `x` is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.
- `col_name`: is the colname from returned tibble in class type. The same from parsnip and tidymodels Default is `.pred_class`
- `...`: This parameter is included for compatibility reasons.

**Details**

For additional help see `setred` examples.

**Value**

Vector with the labels assigned.

### predict.snnrce

**Predictions of the SNNRCE method**

**Description**

Predicts the label of instances according to the snnrce model.

**Usage**

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


predict.snnrceG

Arguments

object SNNRCE model built with the snnrce function.

x A object that can be coerced as matrix. Depending on how was the model built, 
ex is interpreted as a matrix with the distances between the unseen instances and 
the selected training instances, or a matrix of instances.

... This parameter is included for compatibility reasons.

Details

For additional help see snnrce examples.

Value

Vector with the labels assigned.

Description

Predicts the label of instances according to the snnrceG model.

Usage

## S3 method for class 'snnrceG'
predict(object, D, ...)

Arguments

object model instance

D distance matrix

... This parameter is included for compatibility reasons.
predict.SSLRDecisionTree_fitted

Predictions of the SSLRDecisionTree_fitted method

Description

Predicts the label of instances SSLRDecisionTree_fitted model.

Usage

```r
## S3 method for class 'SSLRDecisionTree_fitted'
predict(object, x, type = "class", ...)
```

Arguments

- `object`: model SSLRDecisionTree_fitted.
- `x`: A object that can be coerced as matrix. Depending on how was the model built, `x` is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.
- `type`: of predict in principal model
- `...`: This parameter is included for compatibility reasons.

Value

Vector with the labels assigned.

predict.triTraining

Predictions of the Tri-training method

Description

Predicts the label of instances according to the triTraining model.

Usage

```r
## S3 method for class 'triTraining'
predict(object, x, ...)n```

Arguments

- `object`: Tri-training model built with the triTraining function.
- `x`: A object that can be coerced as matrix. Depending on how was the model built, `x` is interpreted as a matrix with the distances between the unseen instances and the selected training instances, or a matrix of instances.
- `...`: This parameter is included for compatibility reasons.
predict.TSVMSSLR

Details
For additional help see triTraining examples.

Value
Vector with the labels assigned.

---

**predict.TSVMSSLR**

*Predict TSVMS SSLR*

**Description**

Predict TSVMS SSLR

**Usage**

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

**Arguments**

- **object** is the object
- **x** is the dataset
- **...** This parameter is included for compatibility reasons.

---

**predict.USMLeastSquaresClassifierSSLR**

*Predict USMLeastSquaresClassifierSSLR*

**Description**

Predict USMLeastSquaresClassifierSSLR

**Usage**

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

**Arguments**

- **object** is the object
- **x** is the dataset
- **...** This parameter is included for compatibility reasons.
### predict.WellSVMSSLR

*Predict WellSVMSSLR*

#### Description

Predict WellSVMSSLR

#### Usage

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

#### Arguments

- `object`: is the object
- `x`: is the dataset
- `...`: This parameter is included for compatibility reasons.

---

### predict_inputs

*An S4 method to predict inputs.*

#### Description

An S4 method to predict inputs.

#### Usage

```r
predict_inputs(object, ...)```

#### Arguments

- `object`: DecisionTree object
- `...`: This parameter is included for compatibility reasons.
predict_inputs, DecisionTreeClassifier-method

Predict inputs Decision Tree

Description
Function to predict one input in Decision Tree

Usage
```r
## S4 method for signature 'DecisionTreeClassifier'
predict_inputs(object, inputs, type = "class")
```

Arguments
- `object`: DecisionTree object
- `inputs`: inputs to be predicted
- `type`: type prediction, class or prob

print.model_sslr

Print model SSLR

Description
Print model SSLR

Usage
```r
## S3 method for class 'model_sslr'
print(object)
```

Arguments
- `object`: model_sslr object to print

RandomForestSemisupervised-class

Class Random Forest

Description
Class Random Forest Slots: mtry, trees, min_n, w, classes, mode
Description

Self-training is a simple and effective semi-supervised learning classification method. The self-training classifier is initially trained with a reduced set of labeled examples. Then it is iteratively retrained with its own most confident predictions over the unlabeled examples. Self-training follows a wrapper methodology using a base supervised classifier to establish the possible class of unlabeled instances.

Usage

selfTraining(learner, max.iter = 50, perc.full = 0.7, thr.conf = 0.5)

Arguments

learner model from parsnip package for training a supervised base classifier using a set of instances. This model need to have probability predictions (or optionally a distance matrix) and it's corresponding classes.

max.iter maximum number of iterations to execute the self-labeling process. Default is 50.

perc.full A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-training process is stopped. Default is 0.7.

thr.conf A number between 0 and 1 that indicates the confidence threshold. At each iteration, only the newly labelled examples with a confidence greater than this value (thr.conf) are added to the training set.

Details

For predicting the most accurate instances per iteration, selfTraining uses the predictions obtained with the learner specified. To train a model using the learner function, it is required a set of instances (or a precomputed matrix between the instances if x.inst parameter is FALSE) in conjunction with the corresponding classes. Additional parameters are provided to the learner function via the learner.pars argument. The model obtained is a supervised classifier ready to predict new instances through the pred function. Using a similar idea, the additional parameters to the pred function are provided using the pred.pars argument. The pred function returns the probabilities per class for each new instance. The value of the thr.conf argument controls the confidence of instances selected to enlarge the labeled set for the next iteration.

The stopping criterion is defined through the fulfillment of one of the following criteria: the algorithm reaches the number of iterations defined in the max.iter parameter or the portion of the unlabeled set, defined in the perc.full parameter, is moved to the labeled set. In some cases, the process stops and no instances are added to the original labeled set. In this case, the user must assign a more flexible value to the thr.conf parameter.
Value

(When model fit) A list object of class "selfTraining" containing:

- **model** The final base classifier trained using the enlarged labeled set.
- **instances.index** The indexes of the training instances used to train the **model**. These indexes include the initial labeled instances and the newly labeled instances. Those indexes are relative to **x** argument.
- **classes** The levels of **y** factor.
- **pred** The function provided in the **pred** argument.
- **pred.pars** The list provided in the **pred.pars** argument.

References

David Yarowsky.  
*Unsupervised word sense disambiguation rivaling supervised methods.*  

Examples

```r
library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)

data(wine)

set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[train.index,]
test <- wine[-train.index,]

cls <- which(colnames(wine) == "Wine")

#% LABELED
labeled.index <- createDataPartition(train$Wine, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

#We need a model with probability predictions from parsnip  
#https://tidymodels.github.io/parsnip/articles/articles/Models.html  
#It should be with mode = classification

#For example, with Random Forest  
rf <- rand_forest(trees = 100, mode = "classification") %>%
  set_engine("randomForest")

m <- selfTraining(learner = rf,
  perc.full = 0.7,
  thr.conf = 0.5, max.iter = 10) %>% fit(Wine ~ ., data = train)
```
# Accuracy
predict(m, test) %>%
  bind_cols(test) %>%
  metrics(truth = "Wine", estimate = .pred_class)

---

**selfTrainingG**

**Self-training generic method**

**Description**

Self-training is a simple and effective semi-supervised learning classification method. The self-training classifier is initially trained with a reduced set of labeled examples. Then it is iteratively retrained with its own most confident predictions over the unlabeled examples. Self-training follows a wrapper methodology using one base supervised classifier to establish the possible class of unlabeled instances.

**Usage**

```r
selfTrainingG(
  y,
  gen.learner,
  gen.pred,
  max.iter = 50,
  perc.full = 0.7,
  thr.conf = 0.5
)
```

**Arguments**

- `y` A vector with the labels of training instances. In this vector the unlabeled instances are specified with the value `NA`.
- `gen.learner` A function for training a supervised base classifier. This function needs two parameters, indexes and cls, where indexes indicates the instances to use and cls specifies the classes of those instances.
- `gen.pred` A function for predicting the probabilities per classes. This function must be two parameters, model and indexes, where the model is a classifier trained with `gen.learner` function and indexes indicates the instances to predict.
- `max.iter` Maximum number of iterations to execute the self-labeling process. Default is 50.
- `perc.full` A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-training process is stopped. Default is 0.7.
- `thr.conf` A number between 0 and 1 that indicates the confidence threshold. At each iteration, only the newly labelled examples with a confidence greater than this value (`thr.conf`) are added to the training set.
Details

SelfTrainingG can be helpful in those cases where the method selected as base classifier needs learner and pred functions with other specifications. For more information about the general self-training method, please see the selfTraining function. Essentially, the selfTraining function is a wrapper of the selfTrainingG function.

Value

A list object of class "selfTrainingG" containing:

- **model** The final base classifier trained using the enlarged labeled set.
- **instances.index** The indexes of the training instances used to train the model. These indexes include the initial labeled instances and the newly labeled instances. Those indexes are relative to the y argument.

Examples

```r
library(SSLR)

## Load Wine data set
data(wine)
cls <- which(colnames(wine) == "Wine")
x <- wine[, - cls] # instances without classes
y <- wine[, cls] # the classes
x <- scale(x)

set.seed(20)
# Use 50% of instances for training
tra.idx <- sample(x = length(y), size = ceiling(length(y) * 0.5))
xtrain <- x[tra.idx,]
ytrain <- y[tra.idx]

# Use 70% of train instances as unlabeled set
tra.na.idx <- sample(x = length(tra.idx), size = ceiling(length(tra.idx) * 0.7))
ytrain[tra.na.idx] <- NA

# Use the other 50% of instances for inductive testing
tst.idx <- setdiff(1:length(y), tra.idx)
xtest <- x[tst.idx,] # testing instances
ytest <- y[tst.idx] # classes of instances in xtest

# Use the unlabeled examples for transductive testing
xttest <- x[tra.idx[tra.na.idx],] # transductive testing instances
ytest <- y[tra.idx[tra.na.idx]] # classes of instances in xttest

library(caret)

#PREPARE DATA
data <- cbind(xtrain, Class = ytrain)
```
selfTrainingG

```r
dtrain <- as.matrix(proxy::dist(x = xtrain, method = "euclidean", by_rows = TRUE))
ditest <- as.matrix(proxy::dist(x = xitest, y = xtrain, method = "euclidean", by_rows = TRUE))

ddata <- cbind(dtrain, Class = ytrain)
ddata <- as.data.frame(ddata)

ktrain <- as.matrix(exp(-0.048 * dtrain ^ 2))
kdata <- cbind(ktrain, Class = ytrain)
kdata <- as.data.frame(kdata)

kitest <- as.matrix(exp(-0.048 * ditest ^ 2))

## Example: Training from a set of instances with 1-NN (knn3) as base classifier.
gen.learner <- function(indexes, cls)
caret::knn3(x = xtrain[indexes,], y = cls, k = 1)
gen.pred <- function(model, indexes)
predict(model, xtrain[indexes,])

trControl_selfTrainingG1 <- list(gen.learner = gen.learner, gen.pred = gen.pred)
md1 <- train_generic(ytrain, method = "selfTrainingG", trControl = trControl_selfTrainingG1)
p1 <- predict(md1$model, xitest, type = "class")
table(p1, yitest)

confusionMatrix(p1, yitest)$overall[1]

## Example: Training from a distance matrix with 1-NN (oneNN) as base classifier.
dtrain <- as.matrix(proxy::dist(x = xtrain, method = "euclidean", by_rows = TRUE))
gen.learner <- function(indexes, cls) {
m <- SSLR::oneNN(y = cls)
attr(m, "tra.idxs") <- indexes
m
}
gen.pred <- function(model, indexes) {
tra.idxs <- attr(model, "tra.idxs")
d <- dtrain[indexes, tra.idxs]
prob <- predict(model, d, distance.weighting = "none")
prob
}

trControl_selfTrainingG2 <- list(gen.learner = gen.learner, gen.pred = gen.pred)
md2 <- train_generic(ytrain, method = "selfTrainingG", trControl = trControl_selfTrainingG2)
ditest <- proxy::dist(x = xitest, y = xtrain[md2$instances.index,],)
```

setred

```r
method = "euclidean", by_rows = TRUE)
p2 <- predict(md2$model, ditest, type = "class")
table(p2, yitest)

confusionMatrix(p2, yitest)$overall[1]
```

---

**General Interface for SETRED model**

**Description**

SETRED (SElf-TRaining with EDiting) is a variant of the self-training classification method (as implemented in the function `selfTraining`) with a different addition mechanism. The SETRED classifier is initially trained with a reduced set of labeled examples. Then, it is iteratively retrained with its own most confident predictions over the unlabeled examples. SETRED uses an amending scheme to avoid the introduction of noisy examples into the enlarged labeled set. For each iteration, the mislabeled examples are identified using the local information provided by the neighborhood graph.

**Usage**

```r
setred(
  dist = "Euclidean",
  learner,
  theta = 0.1,
  max.iter = 50,
  perc.full = 0.7,
  D = NULL
)
```

**Arguments**

- `dist` A distance function or the name of a distance available in the `proxy` package to compute. Default is "Euclidean" the distance matrix in the case that `D` is `NULL`.
- `learner` model from `parsnip` package for training a supervised base classifier using a set of instances. This model need to have probability predictions (or optionally a distance matrix) and it's corresponding classes.
- `theta` Rejection threshold to test the critical region. Default is 0.1.
- `max.iter` maximum number of iterations to execute the self-labeling process. Default is 50.
- `perc.full` A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-training process is stopped. Default is 0.7.
- `D` A distance matrix between all the training instances. This matrix is used to construct the neighborhood graph. Default is `NULL`, this means the method create a matrix with dist param.
Details

SETRED initiates the self-labeling process by training a model from the original labeled set. In each iteration, the learner function detects unlabeled examples for which it makes the most confident prediction and labels those examples according to the pred function. The identification of mislabeled examples is performed using a neighborhood graph created from the distance matrix. Most examples possess the same label in a neighborhood. So if an example locates in a neighborhood with too many neighbors from different classes, this example should be considered problematic. The value of the theta argument controls the confidence of the candidates selected to enlarge the labeled set. The lower this value is, the more restrictive is the selection of the examples that are considered good. For more information about the self-labeled process and the rest of the parameters, please see selfTraining.

Value

(When model fit) A list object of class "setred" containing:

model  The final base classifier trained using the enlarged labeled set.
instances.index  The indexes of the training instances used to train the model. These indexes include the initial labeled instances and the newly labeled instances. Those indexes are relative to x argument.
classes  The levels of y factor.
pred  The function provided in the pred argument.
pred.pars  The list provided in the pred.pars argument.

References

Ming Li and ZhiHua Zhou.
Setred: Self-training with editing.

Examples

library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)
data(wine)
set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[train.index,]
test <- wine[-train.index,]
cls <- which(colnames(wine) == "Wine")

#% LABELED
labeled.index <- createDataPartition(wine$Wine, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

# We need a model with probability predictions from parsnip
# https://tidymodels.github.io/parsnip/articles/articles/Models.html
# It should be with mode = classification

# For example, with Random Forest
rf <- rand_forest(trees = 100, mode = "classification") %>%
  set_engine("randomForest")

m <- setred(learner = rf,
            theta = 0.1,
            max.iter = 2,
            perc.full = 0.7) %>% fit(Wine ~ ., data = train)

# Accuracy
predict(m, test) %>%
  bind_cols(test) %>%
  metrics(truth = "Wine", estimate = .pred_class)

# Another example, with dist matrix

distance <- as.matrix(proxy::dist(train[,-cls], method = "Euclidean",
                                   by_rows = TRUE, diag = TRUE, upper = TRUE))

m <- setred(learner = rf,
            theta = 0.1,
            max.iter = 2,
            perc.full = 0.7,
            D = distance) %>% fit(Wine ~ ., data = train)

# Accuracy
predict(m, test) %>%
  bind_cols(test) %>%
  metrics(truth = "Wine", estimate = .pred_class)

---

setredG

**SETRED generic method**

**Description**

SETRED is a variant of the self-training classification method (`selfTraining`) with a different addition mechanism. The SETRED classifier is initially trained with a reduced set of labeled examples. Then it is iteratively retrained with its own most confident predictions over the unlabeled examples. SETRED uses an amending scheme to avoid the introduction of noisy examples into
the enlarged labeled set. For each iteration, the mislabeled examples are identified using the local information provided by the neighborhood graph.

Usage

```
setredG(
  y, D,
  gen.learner, gen.pred,
  theta = 0.1, max.iter = 50, perc.full = 0.7
)
```

Arguments

- **y**: A vector with the labels of training instances. In this vector the unlabeled instances are specified with the value `NA`.
- **D**: A distance matrix between all the training instances. This matrix is used to construct the neighborhood graph.
- **gen.learner**: A function for training a supervised base classifier. This function needs two parameters, indexes and cls, where indexes indicates the instances to use and cls specifies the classes of those instances.
- **gen.pred**: A function for predicting the probabilities per classes. This function must be two parameters, model and indexes, where the model is a classifier trained with `gen.learner` function and indexes indicates the instances to predict.
- **theta**: Rejection threshold to test the critical region. Default is 0.1.
- **max.iter**: Maximum number of iterations to execute the self-labeling process. Default is 50.
- **perc.full**: A number between 0 and 1. If the percentage of new labeled examples reaches this value the self-training process is stopped. Default is 0.7.

Details

SetredG can be helpful in those cases where the method selected as base classifier needs a learner and pred functions with other specifications. For more information about the general setred method, please see `setred` function. Essentially, `setred` function is a wrapper of `setredG` function.

Value

A list object of class "setredG" containing:

- **model**: The final base classifier trained using the enlarged labeled set.
- **instances.index**: The indexes of the training instances used to train the model. These indexes include the initial labeled instances and the newly labeled instances. Those indexes are relative to the `y` argument.
Examples

library(SSLR)
library(caret)

## Load Wine data set
data(wine)

cls <- which(colnames(wine) == "Wine")
x <- wine[, -cls] # instances without classes
y <- wine[, cls] # the classes
x <- scale(x) # scale the attributes

## Prepare data
set.seed(20)
# Use 50% of instances for training
tra.idx <- sample(x = length(y), size = ceiling(length(y) * 0.5))
xtrain <- x[tra.idx,] # training instances
ytrain <- y[tra.idx] # classes of training instances
# Use 70% of train instances as unlabeled set
tra.na.idx <- sample(x = length(tra.idx), size = ceiling(length(tra.idx) * 0.7))
ytrain[tra.na.idx] <- NA # remove class information of unlabeled instances
# Use the other 50% of instances for inductive testing
tst.idx <- setdiff(1:length(y), tra.idx)
xtest <- x[tst.idx,] # testing instances
ytest <- y[tst.idx] # classes of testing instances

# Compute distances between training instances
D <- as.matrix(proxy::dist(x = xtrain, method = "euclidean", by_rows = TRUE))

## Example: Training from a set of instances with 1-NN (knn3) as base classifier.
# Compute distances between training instances
D <- as.matrix(proxy::dist(x = xtrain, method = "euclidean", by_rows = TRUE))

## Example: Training from a set of instances with 1-NN (knn3) as base classifier.
gen.learner <- function(indexes, cls)
  caret::knn3(x = xtrain[indexes,], y = cls, k = 1)
gen.pred <- function(model, indexes)
  predict(model, xtrain[indexes,])
trControl_SETRED1 <- list(D = D, gen.learner = gen.learner,
                           gen.pred = gen.pred)
md1 <- train_generic(ytrain, method = "setredG", trControl = trControl_SETRED1)

'md1 <- setredG(y = ytrain, D, gen.learner, gen.pred)'

cls1 <- predict(md1$model, xitest, type = "class")
table(cls1, yitest)
confusionMatrix(cls1, yitest)$overall[1]
## Example: Training from a distance matrix with 1-NN (oneNN) as base classifier

```r
gen.learner <- function(indexes, cls) {
  m <- SSLR::oneNN(y = cls)
  attr(m, "tra.idxs") <- indexes
  m
}

gen.pred <- function(model, indexes) {
  tra.idxs <- attr(model, "tra.idxs")
  d <- D[indexes, tra.idxs]
  prob <- predict(model, d, distance.weighting = "none")
  prob
}

trControl_SETRED2 <- list(D = D, gen.learner = gen.learner,
                          gen.pred = gen.pred)
md2 <- train_generic(ytrain, method = "setredG", trControl = trControl_SETRED2)

ditest <- proxy::dist(x = xitest, y = xtrain[md2$instances.index,],
                      method = "euclidean", by_rows = TRUE)
cls2 <- predict(md2$model, ditest, type = "class")
table(cls2, yitest)
confusionMatrix(cls2, yitest)$overall[1]
```

---

**snnrce**

### General Interface for SNNRCE model

#### Description

SNNRCE (Self-training Nearest Neighbor Rule using Cut Edges) is a variant of the self-training classification method (selfTraining) with a different addition mechanism and a fixed learning scheme (1-NN). SNNRCE uses an amending scheme to avoid the introduction of noisy examples into the enlarged labeled set. The mislabeled examples are identified using the local information provided by the neighborhood graph. A statistical test using cut edge weight is used to modify the labels of the misclassified examples.

#### Usage

```r
snnrce(x.inst = TRUE, dist = "Euclidean", alpha = 0.1)
```
Arguments

x.inst  A boolean value that indicates if x is or not an instance matrix. Default is TRUE.
dist  A distance function available in the proxy package to compute the distance matrix in the case that xINST is TRUE.
alpha  Rejection threshold to test the critical region. Default is 0.1.

Details

SNNRCE initiates the self-labeling process by training a 1-NN from the original labeled set. This method attempts to reduce the noise in examples by labeling those instances with no cut edges in the initial stages of self-labeling learning. These highly confident examples are added into the training set. The remaining examples follow the standard self-training process until a minimum number of examples will be labeled for each class. A statistical test using cut edge weight is used to modify the labels of the misclassified examples. The value of the alpha argument defines the critical region where the candidates examples are tested. The higher this value is, the more relaxed it is the selection of the examples that are considered mislabeled.

Value

(When model fit) A list object of class "snnrce" containing:

model  The final base classifier trained using the enlarged labeled set.
instances.index  The indexes of the training instances used to train the model. These indexes include the initial labeled instances and the newly labeled instances. Those indexes are relative to x argument.
classes  The levels of Y factor.
x.inst  The value provided in the xINST argument.
dist  The value provided in the dist argument when xINST is TRUE.
xtrain  A matrix with the subset of training instances referenced by the indexes instances.index when xINST is TRUE.

References


Examples

library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)
data(wine)
set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[, train.index,]
```r
# Labeled
labeled.index <- createDataPartition(wine$Wine, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- snnrce(x.inst = TRUE,
             dist = "Euclidean",
             alpha = 0.1) %>% fit(Wine ~ ., data = train)

predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Wine", estimate = .pred_class)
```

---

**SSLRDecisionTree**  
*General Interface Decision Tree model*

**Description**

Decision Tree is a simple and effective semi-supervised learning method. Based on the article "Semi-supervised classification trees". It also offers many parameters to modify the behavior of this method. It is the same as the traditional Decision Tree algorithm, but the difference is how the gini coefficient is calculated (classification). In regression we use SSE metric (different from the original investigation) It can be used in classification or regression. If Y is numeric is for regression, classification in another case

**Usage**

```r
SSLRDecisionTree(
  max_depth = 30,
  w = 0.5,
  min_samples_split = 20,
  min_samples_leaf = ceiling(min_samples_split/3)
)
```

**Arguments**

- `max_depth`  
  A number from 1 to Inf. Is the maximum number of depth in Decision Tree  
  Default is 30

- `w`  
  weight parameter ranging from 0 to 1. Default is 0.5

- `min_samples_split`  
  the minimum number of observations to do split. Default is 20

- `min_samples_leaf`  
  the minimum number of any terminal leaf node. Default is ceiling(min_samples_split/3)
Details

In this model we can make predictions with prob type.

References

Jurica Levati, Michelangelo Ceci, Dragi Kocev, Saso Dzeroski.
Semi-supervised classification trees.
Published online: 25 March 2017 © Springer Science Business Media New York 2017

Examples

```r
library(tidyverse)
library(caret)
library(SSLR)
library(tidymodels)

data(wine)

set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[train.index,]
test <- wine[-train.index,]

cls <- which(colnames(wine) == "Wine")

#% LABELED
labeled.index <- createDataPartition(wine$Wine, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- SSLRDecisionTree(min_samples_split = round(length(labeled.index) * 0.25),
  w = 0.3,
  ) %>% fit(Wine ~ ., data = train)

#Accuracy
predict(m,test) %>%
  bind_cols(test) %>%
  metrics(truth = "Wine", estimate = .pred_class)

#For probabilities
predict(m,test, type = "prob")
```
SSLRRandomForest

Description

Random Forest is a simple and effective semi-supervised learning method. It is the same as the traditional Random Forest algorithm, but the difference is that it uses Semi-supervised Decision Trees. It can be used in classification or regression. If Y is numeric, it is for regression; classification in another case.

Usage

SSLRRandomForest(
mtry = NULL,
trees = 500,
min_n = NULL,
w = 0.5,
replace = TRUE,
tree_max_depth = Inf,
sampsize = NULL,
min_samples_leaf = NULL,
allowParallel = TRUE
)

Arguments

mtry number of features in each decision tree. Default is null. This means that mtry = log(n_features) + 1

trees number of trees. Default is 500

min_n number of minimum samples in each tree. Default is null. This means that it uses all training data.

w weight parameter ranging from 0 to 1. Default is 0.5

replace replacing type in sampling. Default is true

tree_max_depth maximum tree depth. Default is Inf

sampsize Size of sample. Default if (replace) nrow(x) else ceiling(.632*nrow(x))

min_samples_leaf the minimum number of any terminal leaf node. Default is 1

allowParallel Execute Random Forest in parallel if doParallel is loaded. Default is TRUE

Details

We can use parallel processing with doParallel package and allowParallel = TRUE.

References

Examples

```r
library(tidyverse)
library(caret)
library(SSLR)
library(tidymodels)

data(wine)

set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[train.index,]

# LABELED
labeled.index <- createDataPartition(wine$Wine, p = .2, list = FALSE)
train[!labeled.index,cls] <- NA

m <- SSLRRandomForest(trees = 5, w = 0.3) %>% fit(Wine ~ ., data = train)

#Accuracy
predict(m,test) %>%
bind_cols(test) %>%
metrics(truth = "Wine", estimate = .pred_class)
```

```
train_generic  FUNCTION TO TRAIN GENERIC MODEL

Description

FUNCTION TO TRAIN GENERIC MODEL

Usage

train_generic(y, ...)

Arguments

y         (optional) factor (classes)

...       list parms trControl (method...)

Value

model trained
```
triTraining  General Interface for Tri-training model

Description

Tri-training is a semi-supervised learning algorithm with a co-training style. This algorithm trains three classifiers with the same learning scheme from a reduced set of labeled examples. For each iteration, an unlabeled example is labeled for a classifier if the other two classifiers agree on the labeling proposed.

Usage

triTraining(learner)

Arguments

learner  model from parsnip package for training a supervised base classifier using a set of instances. This model need to have probability predictions (or optionally a distance matrix) and it’s corresponding classes.

Details

Tri-training initiates the self-labeling process by training three models from the original labeled set, using the learner function specified. In each iteration, the algorithm detects unlabeled examples on which two classifiers agree with the classification and includes these instances in the enlarged set of the third classifier under certain conditions. The generation of the final hypothesis is produced via the majority voting. The iteration process ends when no changes occur in any model during a complete iteration.

Value

A list object of class "triTraining" containing:

- **model**  The final three base classifiers trained using the enlarged labeled set.
- **model.index**  List of three vectors of indexes related to the training instances used per each classifier. These indexes are relative to the y argument.
- **instances.index**  The indexes of all training instances used to train the three models. These indexes include the initial labeled instances and the newly labeled instances. These indexes are relative to the y argument.
- **model.index.map**  List of three vectors with the same information in model.index but the indexes are relative to instances.index vector.
- **classes**  The levels of y factor.
- **pred**  The function provided in the pred argument.
- **pred.pars**  The list provided in the pred.pars argument.
- **x.inst**  The value provided in the x.inst argument.
tritTrainingCombine

References

ZhiHua Zhou and Ming Li.

Tri-training: exploiting unlabeled data using three classifiers.


Examples

library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)
data(wine)
set.seed(1)
train.index <- createDataPartition(wine$Wine, p = .7, list = FALSE)
train <- wine[train.index,]
test <- wine[-train.index,]

cls <- which(colnames(wine) == "Wine")

# Labeled
labeled.index <- createDataPartition(wine$Wine, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

# We need a model with probability predictions from parsnip
# https://tidymodels.github.io/parsnip/articles/articles/Models.html
# It should be with mode = classification

# For example, with Random Forest
rf <- rand_forest(trees = 100, mode = "classification") %>%
  set_engine("randomForest")

m <- tritTraining(learner = rf) %>% fit(Wine ~ ., data = train)

# Accuracy
predict(m,test) %>
  bind_cols(test) %>
  metrics(truth = "Wine", estimate = .pred_class)

triTrainingCombine

Combining the hypothesis

Description

This function combines the predictions obtained by the set of classifiers.
triTrainingG

Usage

triTrainingCombine(pred)

Arguments

pred A list with the predictions of each classifiers

Value

A vector of classes

triTrainingG Tri-training generic method

Description

Tri-training is a semi-supervised learning algorithm with a co-training style. This algorithm trains three classifiers with the same learning scheme from a reduced set of labeled examples. For each iteration, an unlabeled example is labeled for a classifier if the other two classifiers agree on the labeling proposed.

Usage

triTrainingG(y, gen.learner, gen.pred)

Arguments

y A vector with the labels of training instances. In this vector the unlabeled instances are specified with the value NA.

gen.learner A function for training three supervised base classifiers. This function needs two parameters, indexes and cls, where indexes indicates the instances to use and cls specifies the classes of those instances.

gen.pred A function for predicting the probabilities per classes. This function must be two parameters, model and indexes, where the model is a classifier trained with gen.learner function and indexes indicates the instances to predict.

Details

TriTrainingG can be helpful in those cases where the method selected as base classifier needs a learner and pred functions with other specifications. For more information about the general triTraining method, please see the triTraining function. Essentially, the triTraining function is a wrapper of the triTrainingG function.
Value

A list object of class "triTrainingG" containing:

- **model**: The final three base classifiers trained using the enlarged labeled set.
- **model.index**: List of three vectors of indexes related to the training instances used per each classifier. These indexes are relative to the `y` argument.
- **instances.index**: The indexes of all training instances used to train the three models. These indexes include the initial labeled instances and the newly labeled instances. These indexes are relative to the `y` argument.
- **model.index.map**: List of three vectors with the same information in `model.index` but the indexes are relative to `instances.index` vector.

Examples

```r
library(SSLR)
library(caret)

## Load Wine data set
data(wine)

cls <- which(colnames(wine) == "Wine")
x <- wine[, - cls] # instances without classes
y <- wine[, cls] # the classes
x <- scale(x) # scale the attributes

## Prepare data
set.seed(20)
# Use 50% of instances for training
tra.idx <- sample(x = length(y), size = ceiling(length(y) * 0.5))
xtrain <- x[tra.idx,] # training instances
ytrain <- y[tra.idx] # classes of training instances
# Use 70% of train instances as unlabeled set
tra.na.idx <- sample(x = length(tra.idx), size = ceiling(length(tra.idx) * 0.7))
ytrain[tra.na.idx] <- NA # remove class information of unlabeled instances
# Use the other 50% of instances for inductive testing
tst.idx <- setdiff(1:length(y), tra.idx)
xittest <- x[tst.idx,] # testing instances
yittest <- y[tst.idx] # classes of testing instances

## Example: Training from a set of instances with 1-NN (knn3) as base classifier.
gen.learner <- function(indexes, cls)
caret::knn3(x = xtrain[indexes,], y = cls, k = 1)
gen.pred <- function(model, indexes)
predict(model, xtrain[indexes,])

# Train
set.seed(1)

trControl_triTraining1 <- list(gen.learner = gen.learner,
gen.pred = gen.pred)
```
md1 <- train_generic(ytrain, method = "triTrainingG", trControl = trControl_triTraining1)

# Predict testing instances using the three classifiers
pred <- lapply(
  X = md1$model,
  FUN = function(m) predict(m, xitest, type = "class")
)

# Combine the predictions
cls1 <- triTrainingCombine(pred)
table(cls1, yitest)

confusionMatrix(cls1, yitest)$overall[1]

## Example: Training from a distance matrix with 1-NN (oneNN) as base classifier.

dtrain <- as.matrix(proxy::dist(x = xtrain, method = "euclidean", by_rows = TRUE))
gen.learner <- function(indexes, cls) {
  m <- SSLR::oneNN(y = cls)
  attr(m, "tra.idxs") <- indexes
  m
}

gen.pred <- function(model, indexes) {
  tra.idxs <- attr(model, "tra.idxs")
  d <- dtrain[indexes, tra.idxs]
  prob <- predict(model, d, distance.weighting = "none")
  prob
}

# Train
set.seed(1)
trControl_triTraining2 <- list(gen.learner = gen.learner,
                            gen.pred = gen.pred)
md2 <- train_generic(ytrain, method = "triTrainingG", trControl = trControl_triTraining2)

# Predict
ditest <- proxy::dist(x = xitest, y = xtrain[md2$instances.index,],
                      method = "euclidean", by_rows = TRUE)

# Predict testing instances using the three classifiers
pred <- mapply(
  FUN = function(m, indexes) {
    D <- ditest[, indexes]
    predict(m, D, type = "class")
  },
  m = md2$model,
  indexes = md2$model.index.map,
  SIMPLIFY = FALSE
)

# Combine the predictions
cls2 <- triTrainingCombine(pred)  
table(cls2, yitest)

confusionMatrix(cls2, yitest)$overall[1]

---

**TSVMSSLR**

*General Interface for TSVM (Transductive SVM classifier using the convex concave procedure) model*

**Description**

model from RSSL package Transductive SVM using the CCCP algorithm as proposed by Collobert et al. (2006) implemented in R using the quadprog package. The implementation does not handle large datasets very well, but can be useful for smaller datasets and visualization purposes. C is the cost associated with labeled objects, while Cstar is the cost for the unlabeled objects. s control the loss function used for the unlabeled objects: it controls the size of the plateau for the symmetric ramp loss function. The balancing constraint makes sure the label assignments of the unlabeled objects are similar to the prior on the classes that was observed on the labeled data.

**Usage**

```r
TSVMSSLR(
  C = 1,
  Cstar = 0.1,
  kernel = kernlab::vanilladot(),
  balancing_constraint = TRUE,
  s = 0,
  x_center = TRUE,
  scale = FALSE,
  eps = 1e-09,
  max_iter = 20,
  verbose = FALSE
)
```

**Arguments**

- **C** numeric; Cost parameter of the SVM
- **Cstar** numeric; Cost parameter of the unlabeled objects
- **kernel** kernlab::kernel to use
- **balancing_constraint** logical; Whether a balancing constraint should be enforced that causes the fraction of objects assigned to each label in the unlabeled data to be similar to the label fraction in the labeled data.
- **s** numeric; parameter controlling the loss function of the unlabeled objects (generally values between -1 and 0)
- **x_center** logical; Should the features be centered?
scale  If TRUE, apply a z-transform to all observations in X and X_u before running the regression

eps  numeric; Stopping criterion for the maximinimization

max_iter  integer; Maximum number of iterations

verbose  logical; print debugging messages, only works for vanilladot() kernel (default: FALSE)

References


Examples

library(tidyverse)
library(caret)
library(tidymodels)
library(SSLR)

data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,]
test <- breast[-train.index,]

cls <- which(colnames(breast) == "Class")

#% LABELED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

library(kernlab)
m <- TSVMSSLR(kernel = kernlab::vanilladot()) %>% fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%	bind_cols(test) %>%
metrics(truth = "Class", estimate = .pred_class)
USMLeastSquaresClassifierSSLR

Description

This method is a supervised least squares model from RSSL package. It uses the closed form solution of the supervised least squares problem, except that the second moment matrix (X'X) is exchanged with a second moment matrix that is estimated based on all data. See for instance Shaffer1991, where in this implementation we use all data to estimate E(X'X), instead of just the labeled data. This method seems to work best when the data is first centered x_center=TRUE and the outputs are scaled using y_scale=TRUE.

Usage

USMLeastSquaresClassifierSSLR(
  lambda = 0,
  intercept = TRUE,
  x_center = FALSE,
  scale = FALSE,
  y_scale = FALSE,
  ...
)

Arguments

lambda numeric; L2 regularization parameter
intercept logical; Whether an intercept should be included
x_center logical; Should the features be centered?
scale logical; Should the features be normalized? (default: FALSE)
y_scale logical; whether the target vector should be centered
... Not used
use_Xu_for_scaling logical; whether the unlabeled objects should be used to determine the mean and scaling for the normalization

References


Examples

library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)

data(breast)

set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[ train.index,]
test <- breast[-train.index,]
cls <- which(colnames(breast) == "Class")

# Labeled
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- USMLeastSquaresClassifierSSLR() %>% fit(Class ~ ., data = train)

# Accuracy
predict(m,test) %>%
bind_cols(test) %>%
metrics(truth = "Class", estimate = .pred_class)

---

**WellSVMSSLR**

*General Interface for WellSVM model*

**Description**

Model from RSSL package WellSVM is a minimax relaxation of the mixed integer programming problem of finding the optimal labels for the unlabeled data in the SVM objective function. This implementation is a translation of the Matlab implementation of Li (2013) into R.

**Usage**

WellSVMSSLR(
  C1 = 1,
  C2 = 0.1,
  gamma = 1,
  x_center = TRUE,
  scale = FALSE,
  use_Xu_for_scaling = FALSE,
  max_iter = 20
)

**Arguments**

- **C1**  
  double; A regularization parameter for labeled data, default 1;

- **C2**  
  double; A regularization parameter for unlabeled data, default 0.1;

- **gamma**  
  double; Gaussian kernel parameter, i.e., \(k(x,y) = \exp(-\gamma^2||x-y||^2/\text{avg})\) where \(\text{avg}\) is the average distance among instances; when \(\gamma = 0\), linear kernel is used. default gamma = 1;

- **x_center**  
  logical; Should the features be centered?

- **scale**  
  logical; Should the features be normalized? (default: FALSE)
use_Xu_for_scaling
logical; whether the unlabeled objects should be used to determine the mean and scaling for the normalization

max_iter
integer; Maximum number of iterations

References

Examples
library(tidyverse)
library(tidymodels)
library(caret)
library(SSLR)
data(breast)
set.seed(1)
train.index <- createDataPartition(breast$Class, p = .7, list = FALSE)
train <- breast[train.index,

test <- breast[-train.index,

cls <- which(colnames(breast) == "Class")

# LABELED
labeled.index <- createDataPartition(breast$Class, p = .2, list = FALSE)
train[-labeled.index,cls] <- NA

m <- WellSVMSSLR() %>% fit(Class ~ ., data = train)

#Accuracy
predict(m,test) %>%
bind_cols(test) %>%
metrics(truth = "Class", estimate = .pred_class)

wine                  Wine recognition data

Description
This dataset is the result of a chemical analysis of wine grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.
Usage

data(wine)

Format

A data frame with 178 rows and 14 variables including the class.

Details

The dataset is taken from the UCI data repository, to which it was donated by Riccardo Leardi, University of Genova. The attributes are as follows:

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline
- Wine (class)

Source

https://archive.ics.uci.edu/ml/datasets/Wine
Index

*Topic datasets
  abalone, 4
  breast, 5
  coffee, 15
  wine, 81

abalone, 4
best_split, 4
best_split, DecisionTreeClassifier-method, 5
breast, 5
calculate_gini, 6
check_value, 6
check_xy_interface, 7
coBC, 7, 10, 43
coBCCombine, 9
cobCG, 10
cobCReg, 12
cobCRegG, 13
coffee, 15
COREG, 15, 43

DecisionTreeClassifier-class, 16
democratic, 16, 19, 44
democraticCombine, 18
democraticG, 19
EMLeastSquaresClassifierSSLR, 20
EMNearestMeanClassifierSSLR, 22
EntropyRegularizedLogisticRegressionSSLR, 23
fit_model_sslr, 24
fit_decision_tree, 25
fit_decision_tree, DecisionTreeClassifier-method, 25
fit_random_forest, RandomForestSemisupervised-method, 26
fit_x_u, 28
fit_x_u.model_sslr, 28
fit_xy.model_sslr, 27
get_class_max_prob, 29
get_function, 29
get_function_generic, 30
get_levels_categorical, 30
get_most_frequented, 31
get_value_mean, 31
get_x_y, 32
gini_or_variance, 32
gini_prob, 33
grow_tree, 33
grow_tree, DecisionTreeClassifier-method, 34
knn_regression, 34
LaplacianSVMSSLR, 35
LinearTSVMSSLR, 36
load_parsnip, 37
load_RANN, 38
load_RSSSL, 38
MCNearestMeanClassifierSSLR, 38
newDecisionTree, 40
Node-class, 40
nullOrNumericOrCharacter-class, 40
oneNN, 41, 48
predict, DecisionTreeClassifier-method, 41
predict, RandomForestSemisupervised-method, 42
predict.coBC, 42
predict.COREG, 43
predict.democratic, 44
predict.EMLeastSquaresClassifierSSLR, 44

83
predict.EMNearestMeanClassifierSSLR, 45
predict.EntropyRegularizedLogisticRegressionSSLR, 45
predict.LaplacianSVMSSLR, 46
predict.LinearTSVMSSLR, 46
predict.MCNearestMeanClassifierSSLR, 47
predict.model_sslr_fitted, 47
predict.OneNN, 48
predict.RandomForestSemisupervised_fitted, 48
predict.selfTraining, 49
predict.setred, 50
predict.snnrce, 50
predict.snnrceG, 51
predict.SSLRDecisionTree_fitted, 52
predict.triTraining, 52
predict.TSVMSSLR, 53
predict.USMLeastSquaresClassifierSSLR, 53
predict.WellSVMSSLR, 54
predict_inputs, 54
predict_inputs, DecisionTreeClassifier-method, 55
print.model_sslr, 55
RandomForestSemisupervised-class, 55
selfTraining, 49, 56, 59, 61–63, 66
selfTrainingG, 58
setred, 50, 61, 64
setredG, 63
snnrce, 51, 66
SSLRDecisionTree, 68
SSLRRandomForest, 69
SVM, 36

train_generic, 71
triTraining, 52, 53, 72, 74
triTrainingCombine, 73
triTrainingG, 74
TSVMSSLR, 77
USMLeastSquaresClassifierSSLR, 78

WellSVMSSLR, 80
wine, 81