Package 'multiclassPairs'

July 23, 2025

Type Package

Title Build MultiClass Pair-Based Classifiers using TSPs or RF

Version 0.4.3

Author Nour-al-dain Marzouka

Maintainer Nour-al-dain Marzouka <Nour-al-dain.Marzouka@med.lu.se>

Description

A toolbox to train a single sample classifier that uses in-sample feature relationships. The relationships are represented as feature1 < feature2 (e.g. gene1 < gene2). We provide two options to go with. First is based on 'switchBox' package which uses Top-score pairs algorithm. Second is a novel implementation based on random forest algorithm. For simple problems we recommend to use one-vs-rest using TSP option due to its simplicity and for being easy to interpret. For complex problems RF performs better. Both lines filter the features first then combine the filtered features to make the list of all the possible rules (i.e. rule1: feature1 < feature2, rule2: feature1 < feature3, etc...). Then the list of rules will be filtered and the most important and informative rules will be kept. The informative rules will be assembled in an one-vs-rest model or in an RF model. We provide a detailed description with each function in this package to explain the filtration and training methodology in each line. Reference: Marzouka & Eriksson (2021) <doi:10.1093/bioinformatics/btab088>.

URL https://github.com/NourMarzouka/multiclassPairs

License GPL (>= 2)

Encoding UTF-8

biocViews Classification

Depends R (>= 4.0.0)

Imports methods, utils, stats, graphics, grDevices, ranger, Boruta, dunn.test, caret, e1071, rdist

Suggests BiocManager, Biobase, switchBox, knitr, rmarkdown, BiocStyle, leukemiasEset, qpdf

RoxygenNote 7.1.1

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2021-05-16 22:30:02 UTC

Contents

do_dunn_test . . . 2
filter_genes_TSP . . . 2
group_TSP . . . 5
optimize_RF . . . 6
plot_binary_RF . . . 10
plot_binary_TSP . . . 15
predict_one_vs_rest_TSP . . . 18
predict_RF . . . 20
print-methods . . . 24
proximity_matrix_RF . . . 25
ReadData . . . 29
sort_genes_RF . . . 30
sort_rules_RF . . . 35
summary_genes_RF . . . 39
train_one_vs_rest_TSP . . . 43
train_RF . . . 45

Index **50**

`do_dunn_test` *internal function*

Description

It loops dunn.test from dunn.test package to perform pairwise comparison between groups for each gene in the data

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

`filter_genes_TSP` *Filter genes/features for multiclass one-vs-rest classifier downstream training*

Description

`filter_genes_TSP` filters genes/features prior of downstream training steps that will involve top scores pairs using switchBox package.

Usage

```
filter_genes_TSP(data_object,
                 filter = c("one_vs_one", "one_vs_rest"),
                 platform_wise = FALSE,
                 featureNo = 1000,
                 UpDown = TRUE,
                 verbose = TRUE)
```

Arguments

data_object	data object generated by ReadData function. Object contains the data and labels.
filter	a character indicating the method of comparison to be used in the filtering. Should be "one_vs_one" or "one_vs_rest".
platform_wise	a logical indicating if the comparisons will be done in each platform alone then the features should be ranked high in all platforms to be selected. If TRUE then the data object should contain platform vector to be used in splitting samples based on the platform before the comparisons
featureNo	an integer indicating the number of features to be returned after the filtering per class
UpDown	a logical value indicating whether an equal number of up and down genes should be considered. If FALSE then the number of features will be collected regardless of the portion of the up genes and down genes
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.

Details

Input data will be ranked (rank the features inside each sample) before appling the comparison methods. Sufficient number of returned features is recommended if large number of rules is used in the downstream training steps.

For one vs rest comparisons, Wilcoxon test will be performed through SWAP.Filter.Wilcoxon function from switchBox package. For one vs one comparisons, dunn test will be performed through dunn.test function from dunn.test package, and this method is recommended in case of class imbalance or if the classes are so close to each other in their properties.

P-values from dunn test are ranked in each one vs one comparison then the ranks are combined, the selected genes should be ranked at the top in all comparisons.

If platform-wise is TRUE, then the gene that is ranked high in all comparisons and in all platforms will be selected.

For example, if we have five classes (i.e. C1-C5), and dunn.test was performed, then C1 will have four comparison agains C2-C5 (so four lists of p-values), pvalues will be ranked (smaller number means smaller p.value) in each list, and the gene that is ranked (5,5,5,5) will be prioritized over the gene that is ranked (1,1,1,6), and in case we have two platforms and we truned the platform-wise to TRUE then we will have 8 lists of p-values, and the top genes will be selected in the same way. And this is apply also on the platform-wise one-vs-rest comparison. So in brief, the lowest rank in the different list will determine the position of the gene in the output.

Other p-value combining methods could be added in the future.

Value

OnevsrestScheme_genes_TSP object that contains the names of the top filtered features for each class

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# random data
Data <- matrix(runif(10000), nrow=100, ncol=100,
               dimnames = list(paste0("G",1:100), paste0("S",1:100)))

# labels
L <- sample(x = c("A","B","C"), size = 100, replace = TRUE)

# study/platform
P <- sample(c("P1","P2"), size = 100, replace = TRUE)

object <- ReadData(Data = Data,
             Labels = L,
                  Platform = P)

# not to run
# switchBox package from Bioconductor is needed
# Visit their website or install switchBox package using:
# if(!requireNamespace("switchBox", quietly = TRUE)){
#       if (!requireNamespace('BiocManager', quietly = TRUE)) {
#       install.packages('BiocManager')
#      }
#      BiocManager::install('switchBox')", call. = FALSE)
#  }

# filtered_genes <- filter_genes_TSP(data_object = object,
#                                    filter = "one_vs_rest",
#                                    platform_wise = FALSE,
#                                    featureNo = 10,
#                                    UpDown = TRUE,
#                                    verbose = FALSE)

# training
# classifier <- train_one_vs_rest_TSP(data_object = object,
#                                filtered_genes = filtered_genes,
#                                k_range = 10:50,
#                                include_pivot = FALSE,
#                                one_vs_one_scores = FALSE,
#                                platform_wise_scores = FALSE,
#                                seed = 1234,
#                                verbose = FALSE)

# results <- predict_one_vs_rest_TSP(classifier = classifier,
```

```
#                                   Data = object,
#                                   tolerate_missed_genes = TRUE,
#                                   weighted_votes = TRUE,
#                                   verbose = FALSE)

# Confusion Matrix and Statistics on training data
#  caret::confusionMatrix(data = factor(results$max_score, levels = unique(L)),
#                         reference = factor(L, levels = unique(L)),
#                         mode="everything")

# plot_binary_TSP(Data = object, classes=c("A","B","C"),
#                 classifier = classifier,
#                 prediction = results,
#                 title = "Test")
```

---

group_TSP *Internal function: for grouping labels for one-vs-rest usage*

Description

Used to convert labels to factor to be used by switchBox package.

Usage

```
group_TSP(label, my_group)
```

Arguments

label	a vector indicating multi classes
my_group	character indicate the wanted class

Value

a factor contains two levels one is the wanted class and the other is "rest" that represent any other class other than the wanted class

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
L <- sample(x = c("A","B","C","D"), size = 1000, replace = TRUE)
```

optimize_RF *Optimize parameters to be used in training the final RF model*

Description

optimize_RF takes a different sets of parameters to be used as training parameters. optimize_RF passes each set of the parameters to train_RF function, then optimize_RF returns the accuracies and related measurements (i.e. number of used genes and rules) for each trained RF model based on each set of parameters. Accuracies can be calculated based on the training data or by applying the trained RF model on another testing data.

Usage

```
optimize_RF(data_object,
            sorted_rules_RF,
            parameters,
            overall = c("Accuracy", "Kappa", "AccuracyLower",
                        "AccuracyUpper", "AccuracyNull", "AccuracyPValue",
                        "McnemarPValue")[1:2],
            byclass = c("Sensitivity", "Specificity",
                        "Pos Pred Value", "Neg Pred Value",
                        "Precision", "Recall", "F1", "Prevalence",
                        "Detection Rate", "Detection Prevalence",
                        "Balanced Accuracy" )[c(11)],
            seed = 123456,
            test_object = NULL,
            impute = TRUE,
            impute_reject = 0.67,
            verbose = FALSE)
```

Arguments

data_object	Data object with labels generated by ReadData function
sorted_rules_RF	sorted rules object generated by sort_rules_RF function
parameters	a dataframe with the variables that the RF model will be trained based on. Column names should match arguments used in train_RF function. Each row represents one trial (model), e.g. a dataframe with 10 rows means you want to check the performance of 10 different RF models based on 10 different set of parameters.
overall	a vector with the names of the overall performance measurements to be reported in the summary table in results. It can be one or more of these measurements: "Accuracy", "Kappa", "AccuracyLower", "AccuracyUpper", "AccuracyNull", "AccuracyPValue", "McnemarPValue". Default is c("Accuracy", "Kappa"). These masurements based on confusionMatrix function output in caret package.

byclass	a vector with the names of the performance measurements for individual classes to be reported in the summary table in results. It can be one or more of these measurements: "Sensitivity", "Specificity", "Pos Pred Value", "Neg Pred Value", "Precision", "Recall", "F1", "Prevalence", "Detection Rate", "Detection Prevalence", "Balanced Accuracy". Default is "Balanced Accuracy". These masurements based on confusionMatrix function output in caret package.
seed	seed to be used in the training process for reproducibility.
test_object	data object with labels generated by ReadData to be used as testing data. If this object is provided then the accuracies and performance results will be based on this object not the training data.
impute	logical to be passed to predict_RF when test_object is used. To impute missed genes and NA values in test_object. Default is TRUE.
impute_reject	a number between 0 and 1 to be passed to predict_RF when test_object is used. It indicate the threshold of the missed rules in the sample. Based on this threshold the sample will be rejected (i.e. skipped) and the missed rules will not be imputed in this sample. Default is 0.67.
verbose	a logical value indicating whether processing messages will be printed or not. Default is FALSE.

Details

optimize_RF helps the user to optimize parameters to be used in train_RF function for a given training dataset.

Value

return optimize_RF_output object which is a list caintains:

summary	dataframe contains the input parameters, number of genes and rules in the model, and the selected overall and by class performance measurements. Each trials (i.e. set of parameters) as on row.
confusionMatrix	list of confusionMatrix objects generated by caret package, which contains the fulloverall and by class performance for each trial
errors	list of errors generated by trials
calls	the call which used to generate this object.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
```

```
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                   genes_altogether = c(10,20,50,100,150,200),
#                   genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
```

```
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
```

```
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
#                       cluster_cols = TRUE)

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

`plot_binary_RF` *Plot binary rule-based heatmaps*

Description

`plot_binary_RF` Plot binary heatmaps for datasets based on rule-based random forest classifier

Usage

```
plot_binary_RF(Data,
               classifier,
               ref              = NULL,
               prediction       = NULL,
               as_training      = FALSE,
               platform         = NULL,
               classes          = NULL,
               platforms_ord    = NULL,
               top_anno         = c("ref", "prediction", "platform")[1],
               title            = "",
               binary_col       = c("white", "black", "gray"),
               ref_col          = NULL,
               pred_col         = NULL,
               platform_col     = NULL,
               show_ref         = TRUE,
               show_predictions = TRUE,
               show_platform    = TRUE,
               show_scores      = TRUE,
```

```
  show_rule_name   = TRUE,
  legend           = TRUE,
  cluster_cols     = TRUE,
  cluster_rows     = TRUE,
  anno_height      = 0.03,
  score_height     = 0.03,
  margin           = c(0, 5, 0, 5))
```

Arguments

Data	a matrix or a dataframe, samples as columns and row as features/genes. Can also take ExpressionSet, or data_object generated by ReadData function.
classifier	Classifier as a rule_based_RandomForest object, generated by train_RF function
ref	Optional vector with the reference labels. Ref labels in data_object will be used if not ref input provided. For ExpressionSet, the name of the ref variable in the pheno data can be used.
prediction	Optional. "ranger.prediction" object for the class scores generated by predict_RF function.
as_training	Logical indicates if the plot is for the training data. It means the predictions will be extracted from the classifier itself and any prediction object will be ignored. If TRUE, then the training data/object should be used for Data argument.
platform	Optional vector with the platform/study labels or any additional annotation. Platform labels in data_object will be used if no platform input is provided. For ExpressionSet, the name of the variable in the pheno data can be used.
classes	Optional vector with class names. This will determine which classes will be plotted and in which order. It is not recommended to use both "classes" and "platforms_ord" arguments together to avoid sample order conflict and may result in an improper plotting for samples.
platforms_ord	Optional vector with the platform/study names. This will determine which platform/study will be plotted and in which order. This will be used when top_anno="platform". It is not recommended to use both "classes" and "platforms_ord" arguments together.
top_anno	Determine the top annotation level. Samples will be grouped based on the top_anno. Input can be one of three options: "ref", "prediction", "platform". Default is "ref".
title	Character input as a title for the whole heatmap. Default is "".
binary_col	Vector determines the colors of the binary heatmap. Default is c("white", "black", "gray"). First color for the color when the rule is false in the sample. Second color for the color when the rule is true. Third color is for NAs.
ref_col	Optional named vector determines the colors of classes for the reference labels. Default is NULL. Vector names should match with the ref labels.
pred_col	Optional named vector determines the colors of classes for the prediction labels. Default is NULL. Vector names should match with the prediction labels in the prediction labels.

platform_col Optional named vector determines the colors of platforms/study labels. Default is NULL. Vector names should match with the platforms/study labels.

show_ref Logical. Determines if the ref labels will be plotted or not. If the top_anno argument is "ref" then show_ref will be ignored and ref labels will be plotted.

show_predictions Logical. Determines if the prediction labels will be plotted or not. If the top_anno argument is "prediction" then show_predictions will be ignored and predictions will be plotted.

show_platform Logical. Determines if the platform/study labels will be plotted or not. If the top_anno argument is "platform" then show_platform will be ignored and platforms will be plotted.

show_scores Logical. Determines if the prediction scores will be plotted or not. To visualize scores, the classifier should be trained with probability=TRUE otherwise show_scores will be turned FALSE automatically.

show_rule_name Logical. Determines if the rule names will be plotted on the left side of the heatmapp or not.

legend Logical. Determines if a legend will be plotted under the heatmap.

cluster_cols Logical. Clustering the samples in each class (i.e. not all samples in the cohort) based on the binary rules for that class. If top_anno is "platform" then the rules from all classes are used to cluster the samples in each platform.

cluster_rows Logical. Clustering the rules in each class.

anno_height Determines the height of the annotations. It is recommended not to go out of this range 0.01<height<0.1. Default is 0.03.

score_height Determines the height of the score bars. It is recommended not to go out of this range 0.01<height<0.1. Default is 0.03.

margin Determines the margins of the heatmap. Default is c(0, 5, 0, 5).

Value

returns a heatmap plot for the binary rule

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)
```

```
# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                   genes_altogether = c(10,20,50,100,150,200),
#                   genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
```

```
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
```

```
#                       cluster_cols = TRUE)

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

---

plot_binary_TSP *Plot binary rule-based heatmaps*

Description

plot_binary_TSP Plot binary heatmaps for datasets based on one-vs-rest multiclass top score pairs classifier

Usage

```
plot_binary_TSP(Data,
                classifier,
                ref              = NULL,
                prediction       = NULL,
                platform         = NULL,
                classes          = NULL,
                platforms_ord    = NULL,
                top_anno         = c("ref", "prediction", "platform")[1],
                title            = "",
                binary_col       = c("white", "black", "gray"),
                ref_col          = NULL,
                pred_col         = NULL,
                platform_col     = NULL,
                show_ref         = TRUE,
                show_predictions = TRUE,
                show_platform    = TRUE,
                show_scores      = TRUE,
                show_rule_name   = TRUE,
                legend           = TRUE,
                cluster_cols     = TRUE,
                cluster_rows     = TRUE,
```

```
anno_height      = 0.03,
score_height     = 0.03,
margin           = c(0, 5, 0, 5))
```

Arguments

Data	a matrix, dataframe, where samples as columns and row as features/genes. Can also take ExpressionSet, or data_object generated by ReadData function.
classifier	classifier as a OnevsrestScheme_TSP object, generated by train_one_vs_rest_TSP function
ref	Optional vector with the reference labels. Ref labels in data_object will be used if not ref input provided. For ExpressionSet, the name of the variable in the pheno data.
prediction	Optional dataframe with class "OneVsRestTSP prediction" generated by predict_one_vs_rest_TSP function with the scores and the predicted labels.
platform	Optional vector with the platform/study labels or any additional annotation. Platform labels in data_object will be used if no platform input provided. For ExpressionSet, the name of the variable in the pheno data.
classes	Optional vector with the class names. This will determine which classes will be plotted and in which order. It is not recommended to use both "classes" and "platforms_ord" arguments together.
platforms_ord	Optional vector with the platform/study names. This will determine which platform/study will be plotted and in which order. This will be used when top_anno="platform". It is not recommended to use both "classes" and "platforms_ord" arguments together.
top_anno	Determine the top annotation level. Samples will be grouped based on the top_anno. Input can be one of three options: "ref", "prediction", "platform". Default is "ref".
title	Carachter input as a title for the whole heatmap. Default is "".
binary_col	vector determines the colors of the binary heatmap. Default is c("white", "black", "gray"). First color for the color when the rule is false in the sample. Second color for the color when the rule is true. Third color is for NAs.
ref_col	optional named vector determines the colors of classes for the reference labels. Default is NULL. Vector names should match with the ref labels.
pred_col	optional named vector determines the colors of classes for the prediction labels. Default is NULL. Vector names should match with the prediction labels in the prediction labels.
platform_col	optional named vector determines the colors of platforms/study labels. Default is NULL. Vector names should match with the platforms/study labels.
show_ref	logical. Determines if the ref labels will be plotted or not. If the top_anno argument is "ref" then show_ref will be ignored.
show_predictions	logical. Determines if the prediction labels will be plotted or not. If the top_anno argument is "prediction" then show_predictions will be ignored.

show_platform	logical. Determines if the platform/study labels will be plotted or not. If the top_anno argument is "platform" then show_platform will be ignored.
show_scores	logical. Determines if the prediction scores will be plotted or not.
show_rule_name	logical. Determines if the rule names will be plotted on the left side of the heatmapp or not.
legend	logical. Determines if a legend will be plotted under the heatmap.
cluster_cols	logical. Clustering the samples in each class (i.e. not all samples in the cohort) based on the binary rules for that class. If top_anno is "platform" then the rules from all classes are used to cluster the samples in each platform.
cluster_rows	logical. Clustering the rules in each class.
anno_height	Determines the height of the annotations. It is recommended not to go out of this range 0.01<height<0.1. Default is 0.03.
score_height	Determines the height of the score bars. It is recommended not to go out of this range 0.01<height<0.1. Default is 0.03.
margin	Determines the margins of the heatmap. Default is c(0, 5, 0, 5).

Value

returns a heatmap plot for the binary rule

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# random data
Data <- matrix(runif(10000), nrow=100, ncol=100,
               dimnames = list(paste0("G",1:100), paste0("S",1:100)))

# labels
L <- sample(x = c("A","B","C"), size = 100, replace = TRUE)

# study/platform
P <- sample(c("P1","P2"), size = 100, replace = TRUE)

object <- ReadData(Data = Data,
             Labels = L,
                   Platform = P)

# not to run
# switchBox package from Bioconductor is needed
# Visit their website or install switchBox package using:
# if(!requireNamespace("switchBox", quietly = TRUE)){
#       if (!requireNamespace('BiocManager', quietly = TRUE)) {
#       install.packages('BiocManager')
#      }
#      BiocManager::install('switchBox')", call. = FALSE)
#  }
```

```
#filtered_genes <- filter_genes_TSP(data_object = object,
#                                   filter = "one_vs_rest",
#                                  platform_wise = FALSE,
#                                  featureNo = 10,
#                                  UpDown = TRUE,
#                                  verbose = FALSE)

# training
# classifier <- train_one_vs_rest_TSP(data_object = object,
#                                filtered_genes = filtered_genes,
#                                k_range = 2:50,
#                                include_pivot = FALSE,
#                                one_vs_one_scores = FALSE,
#                                platform_wise_scores = FALSE,
#                                seed = 1234,
#                                verbose = FALSE)

# results <- predict_one_vs_rest_TSP(classifier = classifier,
#                                    Data = object,
#                                    tolerate_missed_genes = TRUE,
#                                    weighted_votes = TRUE,
#                                    verbose = FALSE)

# Confusion Matrix and Statistics on training data
#  caret::confusionMatrix(data = factor(results$max_score, levels = unique(L)),
#                         reference = factor(L, levels = unique(L)),
#                         mode="everything")

# plot_binary_TSP(Data = object, classes=c("A","B","C"),
#                 classifier = classifier,
#                 prediction = results,
#                 title = "Test")
```

`predict_one_vs_rest_TSP` *Predict sample class based on one-vs-rest multiclass top score pairs classifier*

Description

`predict_one_vs_rest_TSP` predicts sample class based on one-vs-rest multiclass top score pairs classifier classifier

Usage

```
predict_one_vs_rest_TSP(classifier,
                        Data,
                        tolerate_missed_genes = TRUE,
                        weighted_votes = TRUE,
```

```
                        classes,
                        verbose = TRUE)
```

Arguments

classifier	classifier as a OnevsrestScheme_TSP object, generated by train_one_vs_rest_TSP function
Data	a matrix, dataframe, ExpressionSet, or data_object generated by ReadData function. Samples as columns and row as features/genes.
tolerate_missed_genes	logical. TRUE means that if a gene in the classifier is missed in the data then this rule will be not considered in the prediction. If tolerate_missed_genes=TRUE then the user should keep an eye on the left rules. In some cases when the missed genes are too many then no enough rules left for a good prediction.
weighted_votes	logical indicates if the rules will be treated equally or be weighted by their scores. weighted_votes=TRUE is useful to break vote ties between classes.
classes	optional vector with the names of the classes. This will be used to order the columns of the ouput dataframe. In case not all classes in the classifier is mentioned in the vector, then these classes will be excluded from the prediction.
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.

Value

returns dataframe with classes votes, ties, and final prediction

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# random data
Data <- matrix(runif(10000), nrow=100, ncol=100,
               dimnames = list(paste0("G",1:100), paste0("S",1:100)))

# labels
L <- sample(x = c("A","B","C"), size = 100, replace = TRUE)

# study/platform
P <- sample(c("P1","P2"), size = 100, replace = TRUE)

object <- ReadData(Data = Data,
             Labels = L,
                   Platform = P)

# not to run
# switchBox package from Bioconductor is needed
# Visit their website or install switchBox package using:
```

```
# if(!requireNamespace("switchBox", quietly = TRUE)){
#       if (!requireNamespace('BiocManager', quietly = TRUE)) {
#       install.packages('BiocManager')
#      }
#      BiocManager::install('switchBox')", call. = FALSE)
#  }

# filtered_genes <- filter_genes_TSP(data_object = object,
#                                    filter = "one_vs_rest",
#                                    platform_wise = FALSE,
#                                    featureNo = 10,
#                                    UpDown = TRUE,
#                                    verbose = FALSE)

# training
# classifier <- train_one_vs_rest_TSP(data_object = object,
#                                filtered_genes = filtered_genes,
#                                k_range = 2:50,
#                                include_pivot = FALSE,
#                                one_vs_one_scores = FALSE,
#                                platform_wise_scores = FALSE,
#                                seed = 1234,
#                                verbose = FALSE)

# results <- predict_one_vs_rest_TSP(classifier = classifier,
#                                    Data = object,
#                                    tolerate_missed_genes = TRUE,
#                                    weighted_votes = TRUE,
#                                    verbose = FALSE)

# Confusion Matrix and Statistics on training data
#  caret::confusionMatrix(data = factor(results$max_score, levels = unique(L)),
#                         reference = factor(L, levels = unique(L)),
#                         mode="everything")

# plot_binary_TSP(Data = object, classes=c("A","B","C"),
#                 classifier = classifier,
#                 prediction = results,
#                 title = "Test")
```

`predict_RF` *Predict sample class based on gene pair-based random forest classifier*

Description

`predict_RF` predicts sample class based on pair-based random forest classifier

Usage

```
predict_RF(classifier,
           Data,
           impute = FALSE,
           impute_reject = 0.67,
           impute_kNN = 5,
           verbose = TRUE)
```

Arguments

classifier	classifier as a rule_based_RandomForest object, generated by train_RF function
Data	a matrix, dataframe, ExpressionSet, or data_object generated by ReadData function. Samples as columns and row as features/genes.
impute	logical. To determine if missed genes and NA values should be imputed or not. The non missed rules will be used to detemine the closest samples in the training binary matrix (i.e. which is stored in the classifier object). For each sample, the mode value for nearest samples in the training data will be assigned to the missed rules. Default is FALSE.
impute_reject	a number between 0 and 1 indicating the threshold of the missed rules in the sample. Based on this threshold the sample will be rejected (i.e. skipped if higher than the impute_reject threshold) and the missed rules will not be imputed in this sample. Default is 0.67. NOTE, The results object will not have any results for this sample.
impute_kNN	interger determines the number of the nearest samples in the training data to be used in the imputation. Default is 5. It is not recommended to use large number (i.e. >10).
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.

Value

returns predictions object as "ranger.prediction" class from ranger package. If the RF classifier was trained with probability=TRUE then the results will contain the scores for the classes, and to help the user to get clearer outputs predict_RF adds a new slot (i.e. results$predictions_classes) contains a vector with the prediction based on the highest scores in results$predictions. If the RF classifier was trained with probability=FALSE then the results will contain the final class but no scores are provided in results$predictions. In case a sample was rejected in the imputation process (passed the reject cutoff) then it will not be included in the prediction results. This should be kept in mind in case the user wants to match the input samples with the results for the confusion matrix for example. To help the user to get clearer outputs predict_RF adds the sample names as names/row names to the factor/matrix in results$predictions.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                  genes_altogether = c(10,20,50,100,150,200),
#                  genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
```

```
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
```

```
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
#                       cluster_cols = TRUE)

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

`print-methods` *Methods for Function* `print` *in Package* **multiclassPairs**

Description

Methods for function `print` in package **multiclassPairs**

Methods

`signature(x = "multiclassPairs_object")`

`signature(x = "OnevsrestScheme_genes_SB")`

`signature(x = "OnevsrestScheme_SB")`

`signature(x = "RandomForest_sorted_genes")`

`signature(x = "RandomForest_sorted_rules")`

`signature(x = "rule_based_RandomForest")`

proximity_matrix_RF *Plot binary rule-based heatmaps*

Description

proximity_matrix_RF Plot clustering heatmaps showing which out-of-bag samples are predicted in the same class and in the same trees during the training process for the rule-based random forest classifier

Usage

```
proximity_matrix_RF(object,
            classifier,
            plot=TRUE,
            return_matrix=TRUE,
            title         = "",
            top_anno      = c("ref","platform")[1],
            classes       = NULL,
            sam_order     = NULL,
            ref_col       = NULL,
            platform_col  = NULL,
            platforms_ord = NULL,
            show_platform = TRUE,
            cluster_cols  = FALSE,
            legend        = TRUE,
            anno_height   = 0.03,
            margin        = c(0, 5, 0, 5))
```

Arguments

object	data_object generated by ReadData function which was used in the training process.
classifier	classifier as a rule_based_RandomForest object, generated by train_RF function
plot	logical. To plot the proximity matrix or not. Default is TRUE.
return_matrix	logical. To return the proximity matrix or not. Default is TRUE.
title	Character input as a title for the whole heatmap. Default is "".
top_anno	Determine the top annotation level. Samples will be grouped based on the top_anno. Input can be one of two options: "ref", "platform". Default is "ref".
classes	Optional vector with the class names. Classes will determine which classes will be plotted and in which order. It is not recommended to use both "classes" and "platforms_ord" arguments together.
sam_order	Optional vector with the samples order in the heatmap.
ref_col	optional named vector determines the colors of classes for the reference labels. Default is NULL. Vector names should match with the ref labels.

platform_col	optional named vector determines the colors of platforms/study labels. Default is NULL. Vector names should match with the platforms/study labels.
platforms_ord	Optional vector with the platform/study names. This will determine which platform/study will be plotted and in which order. This will be used when top_anno="platform". It is not recommended to use both "classes" and "platforms_ord" arguments together.
show_platform	logical. Determines if the platform/study labels will be plotted or not. If the top_anno argument is "platform" then show_platform will be ignored.
cluster_cols	logical. samples will be grouped based on the class then will be Clustered in each class (i.e. not all samples in the cohort). If top_anno is "platform" then the rules from all classes are used to cluster the samples in each platform.
legend	logical. Determines if a legend will be plotted under the heatmap.
anno_height	Determines the height of the annotations. It is recommended not to go out of this range 0.01<height<0.1. Default is 0.03.
margin	Determines the margins of the heatmap. Default is c(0, 5, 0, 5).

Value

returns the proximity matrix and/or a heatmap plot for the proximity matrix.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
```

```
#                  genes_altogether = c(10,20,50,100,150,200),
#                  genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
```

```
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
#                       cluster_cols = TRUE)

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
```

```
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

ReadData *Function for preparing data object*

Description

ReadData takes data such as matrix, labels, and platform information, and produce data object to be used in the down stream analysis, such as filtering genes.

Usage

```
ReadData(Data, Labels, Platform = NULL, verbose = TRUE)
```

Arguments

Data	a dataframe, matrix, or ExpressionSet with values to be used in the down stream analysis. Samples as columns and rows genes/features as rows. Matrix should has column names and row names. It is recommended to avoid "-" symbol for the feature/gene names.
Labels	a vector indicating the classes of the samples. Should be with the same length of the columns number in data. This can be a variable name stored in the ExpressionSet if ExpressionSet is used.
Platform	Optional, vector with the same length of labels indicating. This can be a variable name stored in the ExpressionSet if ExpressionSet is used.
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.

Value

data object multiclassPairs_object

Data	dataframe (gene as rows and samples as columns)
Labels	a vector containing classes information
Platform	a vector containing Platform information, or NULL if no input is used

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# example of loading data from matrix
Data <- matrix(runif(10000), nrow=100, ncol=100,
               dimnames = list(paste0("G",1:100), paste0("S",1:100)))

L <- sample(x = c("A","B","C"), size = 100, replace = TRUE)

P <- sample(x = c("P1","P2"), size = 100, replace = TRUE)


table(P,L)

object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)
object


# Not to run
# example of loading data from ExpressionSet
# library(leukemiasEset, quietly = TRUE)
# data(leukemiasEset)

# split the data to training and testing
# n <- ncol(leukemiasEset)
# set.seed(1234)
# training_samples <- sample(1:n,size = n*0.6)

# train <- leukemiasEset[1:1000,training_samples]
# test  <- leukemiasEset[1:1000,-training_samples]

# create the data object
# when we use Expressionset we can use the name of the phenotypes variable
# ReadData will automatically extract the phenotype variable and use it as class labels
# the same can be used with the Platform/study labels
# in this example we are not using any platform labels, so leave it NULL
# object <- ReadData(Data = train,
#                    Labels = "LeukemiaType",
#                    Platform = NULL,
#                    verbose = FALSE)
# object
```

`sort_genes_RF` *Sort genes/features for pair-based random forest classifier downstream steps*

Description

`sort_genes_RF` uses random forest to sort genes/features prior of downstream steps such as gene pairs/rules selection which will involve random forest models.

Usage

```
sort_genes_RF(data_object,
              featureNo_altogether,
              featureNo_one_vs_rest,
              rank_data = FALSE,
              platform_wise = FALSE,
              num.trees = 500,
              min.node.size = 1,
              importance = "impurity",
              write.forest = FALSE,
              keep.inbag = FALSE,
              verbose = TRUE, ...)
```

Arguments

data_object	data object generated by ReadData function. Object contains the data and labels.
featureNo_altogether	an integer. Optional. Indicating specific number of top sorted genes to be returned from one random forest model contains all the labels together. If 0 then this sorting will be skipped. By default, if no number is specified then all available genes will be sorted and returned because user can specify how many top genes will be used in the downstream analysis.
featureNo_one_vs_rest	an integer. Optional. Indicating specific number of top sorted genes to be returned from 'one vs rest' random forest models. This means each class will have a random forest where the samples from the other classes will be labels as 'rest'. If 0 then this sorting will be skipped. By default, if no number is specified then all available genes will be sorted and returned because user can specify how many top genes will be used in the downstream analysis.
rank_data	logical indicates if the data should be ranked (features will be ranked inside each sample). Default is FALSE.
platform_wise	logical indicates if the gene importance should be calculated in each platform seperatly then combined based on the lowest importance value (i.e. a gene with low importance in any platform will not be prioritized). Default is FALSE. see details for more description.
num.trees	an integer. Number of trees. Default is 500. It is recommended to increase num.trees in case of having large number of features (ranger function argument).
min.node.size	an integer. Minimal node size. Default is 1. (ranger function argument)
importance	Variable importance mode, should be one of 'impurity', 'impurity_corrected', 'permutation'. Defualt is 'impurity' (ranger function argument)
write.forest	Save ranger.forest object, required for prediction. Default is FALSE to reduce memory. (ranger function argument)
keep.inbag	Save how often observations are in-bag in each tree. Default is FALSE. (ranger function argument)
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.

... any additional arguments to be passed to ranger function (i.e. random forest function) in ranger package. For example, seed for reproducibility.

Details

For platform-wise option. When platform_wise=TRUE, for example, if data has three platforms (i.e. P1, P2, and P3), and random forest was performed for class 1 (C1) versus rest in each platform seperatly, then C1 will have 3 importance lists contain the genes sorted based on P1-P3, genes will be sorted and ranked in each list (lower rank number means higher importance), the combined final sorting will be determined by the lowest importance level in the lists, it means a gene with (5,5,5) will be prioritized over a gene with (1,1,6). And this is applied on the altogether sorting and one-vs-rest sorting. Other combining methods could be added in the future.

Value

returns RandomForest_sorted_genes object which contains sorted genes based on the importance in each class (one-vs-rest) sorting and based altogether sorting. Also it contains the random forest objects those used in the sorting.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                   genes_altogether = c(10,20,50,100,150,200),
#                   genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
```

```
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
```

```
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
#                       cluster_cols = TRUE)

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

sort_rules_RF *Create and sort feature/gene pairs for pair-based random forest classifier training step*

Description

sort_rules_RF uses random forest to create and sort genes/features pairs prior of downstream random forest model training step.

Usage

```
sort_rules_RF(data_object,
              sorted_genes_RF,
              genes_altogether = 50,
              genes_one_vs_rest = 50,
              run_altogether = TRUE,
              run_one_vs_rest = TRUE,
              platform_wise = FALSE,
              num.trees = 500,
              min.node.size = 1,
              importance = "impurity",
              write.forest = FALSE,
              keep.inbag = FALSE,
              verbose = TRUE, ...)
```

Arguments

data_object	data object generated by ReadData function. Object contains the data and labels.
sorted_genes_RF	RandomForest_sorted_genes object created by sort_genes_RF function
genes_altogether	integer indicates how many top gene/features should be used altogether genes. These genes will be pooled with the selected gene from genes_one_vs_rest to create all the possible rules. Default is 200.
genes_one_vs_rest	integer indicates how many top gene/features should be used from one-vs-rest genes. These genes will be pooled with the selected gene from genes_altogether to create all the possible rules. Default is 200.
run_altogether	logical indicates if altogether RF model should be performed to sort the rules based in their importance in all classes together. Default is TRUE.
run_one_vs_rest	logical indicates if one_vs_rest RF model for each class should be performed to sort the rules based in their importance in each class. Default is TRUE.
platform_wise	logical indicates if the rules importance should be calculated in each platform seperatly then combined based on the lowest importance value (i.e. a rule with low importance in any platform will not be prioritized). Default is FALSE. see details for more description.

num.trees	an integer. Number of trees. Default is 500. It is recommended to increase num.trees in case of having large number of features (ranger function argument).
min.node.size	an integer. Minimal node size. Default is 1. (ranger function argument)
importance	Variable importance mode, should be one of 'impurity', 'impurity_corrected', 'permutation'. Defualt is 'impurity' (ranger function argument)
write.forest	Save ranger.forest object, required for prediction. Default is FALSE to reduce memory. (ranger function argument)
keep.inbag	Save how often observations are in-bag in each tree. Default is FALSE. (ranger function argument)
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.
...	any additional arguments to be passed to ranger function (i.e. random forest function) in ranger package. For example, seed for reproducibility.

Details

In case of class imbalance rules_one_vs_rest=TRUE is recommended.

For platform-wise option. When platform_wise=TRUE, for example, if data has three platforms (i.e. P1, P2, and P3), and random forest was performed for class 1 (C1) versus rest in each platform seperatly, then C1 will have 3 importance lists contain the rules sorted based on P1-P3, rules will be sorted and ranked in each list (lower rank number means higher importance), the combined final sorting will be determined by the lowest importance level in the lists, it means a rule with (5,5,5) will be prioritized over a rule with (1,1,6). And this is applied on the altogether sorting and one-vs-rest sorting. Other combining methods could be added in the future.

Value

returns RandomForest_sorted_rules object which contains sorted rules based on the importance in each class (one-vs-rest) sorting and based on altogether sorting. Also it contains the random forest objects those used in the sorting.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
```

```
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                   genes_altogether = c(10,20,50,100,150,200),
#                   genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
```

```
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
#                       cluster_cols = TRUE)
```

```
# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

---

summary_genes_RF *Summarize sorted genes to rules*

Description

After sorting genes RF by sort_genes_RF function summary_genes_RF gives an idea of how many genes you need to use to generate specific number of rules in sort_rules_RF function.

Usage

```
summary_genes_RF(sorted_genes_RF,
                 genes_altogether,
                 genes_one_vs_rest)
```

Arguments

sorted_genes_RF
: sorted genes object with class RandomForest_sorted_genes generated by sort_genes_RF function

genes_altogether
: numeric vector indicating how many genes from altogether slot (i.e. 'all') should be used each time. genes_altogether should be a vector with zero or positive numbers and with the same length of genes_one_vs_rest vector. Each element in this vector will be used with the element with the same index in genes_one_vs_rest vector.

genes_one_vs_rest
: numeric vector indicating how many genes from one_vs_rest slots (i.e. per class) should be used each time. genes_one_vs_rest should be a vector with zero or positive numbers and with the same length of genes_altogether vector. Each element in this vector will be used with the element with the same index in genes_altogether vector.

Details

summary_genes_RF function helps the user to know which number of genes should be used to get the needed number of rules in sort_rules_RF function. NOTE: without consideration of gene replication in rules, because the rules are not sorted yet. summary_genes_RF workes as follows: take the first element in genes_altogether and genes_one_vs_rest, then bring this number of top genes from altogether slot and one_vs_rest slots (this number of genes will be taken from each class), respectively, from the sorted_genes_RF object. Then pool the extracted genes. Then take the unique genes. Then calculate the number of the possible combinations. Store the number of unique genes and rules in first row in the output dataframe then pick the second element from the genes_altogether and genes_one_vs_rest and repeat the steps again.

Value

returns a dataframe with the used paramerters and the expected number of unique genes and rules. Number of rows of the dataframe equals the length of genes_altogether and genes_one_vs_rest.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(8000), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                  genes_altogether = c(10,20,50,100,150,200),
#                  genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
```

```
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)
# parameters

# Or you can use expand.grid to generate dataframe with all parameter combinations
# parameters <- expand.grid(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   num.trees=c(100,500,1000),
#   stringsAsFactors = FALSE)
# parameters


# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
```

```
# # if the classifier trained using probability = FALSE
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# Extract and plot the proximity matrix from the classifier for the training data
# it takes long time for large data
# proximity_mat <- proximity_matrix_RF(object = object,
#                       classifier = RF_classifier,
#                       plot=TRUE,
#                       return_matrix=TRUE,
#                       title = "Test",
#                       cluster_cols = TRUE)

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

`train_one_vs_rest_TSP` *Build multiclass rule-based classifier as one-vs-rest scheme*

Description

`train_one_vs_rest_TSP` trains multiclass classifier in a one-vs-rest scheme by combining binary classifiers for each class produced by switchBox package.

Usage

```
train_one_vs_rest_TSP(data_object,
                      filtered_genes,
                      k_range = 10:50,
                      include_pivot = FALSE,
                      one_vs_one_scores = FALSE,
                      platform_wise_scores = FALSE,
                      disjoint = TRUE,
                      seed = NULL,
                      classes,
                      SB_arg = list(),
                      verbose = TRUE)
```

Arguments

`data_object`	data object generated by ReadData function. Object contains the data and labels.
`filtered_genes`	filtered genes object produced by filter_genes_TSP function
`k_range`	an integer or range represent the candidate number of Top Scoring Pairs (TSPs) in the individual (i.e. binary) classifiers. Default range from 10 to 50.
`include_pivot`	a logical indicating if the filtered genes should also be paired with all features available in the data matrix. Default is FALSE. include_pivot=FALSE means filtered genes will be paired with themselves only.
`one_vs_one_scores`	logical indicating if rules scores for each class should be calculated as a mean of one-vs-one scores instead of one-vs-rest manner. Default is FALSE.
`platform_wise_scores`	logical indicating if rules scores for each class should be calculated in each platform-wise then averaged instead of merging all platforms together. Default is FALSE.
`disjoint`	is a logical value indicating whether only disjoint pairs should be considered in the final set of selected pairs; i.e. all features occur only once among the set of TSPs. This is an argument to be passed to the training function `SWAP.Train.KTSP` from **switchBox** package.
`seed`	an integer to set a seed for the training process (for reproducibility).

classes	optional vector contains the names of classes in the wanted order. This means the individual classifiers will be ordered based on this vector. If this vector does not have all class names, then no classifiers will be train for those classes that are not mentioned and the samples from these classes will be removed from the training dataset.
SB_arg	list of any additional arguments to be passed to the training function SWAP.Train.KTSP from **switchBox** package
verbose	a logical value indicating whether processing messages will be printed or not. Default is TRUE.

Details

This function uses SWAP.Train.KTSP function from **switchBox** where the algorithm (Afsari et al (AOAS, 2014)) chooses the optimal number of rules (i.e. pairs) among the input range.

Value

Returns OnevsrestScheme_TSP object which contains one-vs-rest classifiers for the classes. These individual classifiers are top score pairs classifiers.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# random data
Data <- matrix(runif(10000), nrow=100, ncol=100,
               dimnames = list(paste0("G",1:100), paste0("S",1:100)))

# labels
L <- sample(x = c("A","B","C"), size = 100, replace = TRUE)

# study/platform
P <- sample(c("P1","P2"), size = 100, replace = TRUE)

object <- ReadData(Data = Data,
             Labels = L,
                   Platform = P)

# not to run
# switchBox package from Bioconductor is needed
# Visit their website or install switchBox package using:
# if(!requireNamespace("switchBox", quietly = TRUE)){
#       if (!requireNamespace('BiocManager', quietly = TRUE)) {
#       install.packages('BiocManager')
#      }
#      BiocManager::install('switchBox')", call. = FALSE)
#  }

#filtered_genes <- filter_genes_TSP(data_object = object,
```

```
#                                 filter = "one_vs_rest",
#                                 platform_wise = FALSE,
#                                 featureNo = 10,
#                                 UpDown = TRUE,
#                                 verbose = FALSE)

# training
# classifier <- train_one_vs_rest_TSP(data_object = object,
#                               filtered_genes = filtered_genes,
#                               k_range = 10:50,
#                               include_pivot = FALSE,
#                               one_vs_one_scores = FALSE,
#                               platform_wise_scores = FALSE,
#                               seed = 1234,
#                               verbose = FALSE)

# results <- predict_one_vs_rest_TSP(classifier = classifier,
#                                    Data = object,
#                                    tolerate_missed_genes = TRUE,
#                                    weighted_votes = TRUE,
#                                    verbose = FALSE)

# Confusion Matrix and Statistics on training data
#  caret::confusionMatrix(data = factor(results$max_score, levels = unique(L)),
#                         reference = factor(L, levels = unique(L)),
#                         mode="everything")

# plot_binary_TSP(Data = object, classes=c("A","B","C"),
#                 classifier = classifier,
#                 prediction = results,
#                 title = "Test")
```

`train_RF` *Train pair-based random forest model*

Description

`train_RF` trains random forest model based on binary gene rules (such as geneA<geneB). Boruta package is used to remove the unimportant rules and ranger function from ranger package is used for the training.

Usage

```
train_RF(data_object,
          sorted_rules_RF,
          gene_repetition = 1,
          rules_altogether = 50,
          rules_one_vs_rest = 50,
          run_boruta = FALSE,
```

```
plot_boruta = FALSE,
boruta_args = list(doTrace = 1),
num.trees = 500,
min.node.size = 1,
importance = "impurity",
write.forest = TRUE,
keep.inbag = TRUE,
probability = TRUE,
verbose = TRUE, ...)
```

Arguments

data_object	data object generated by ReadData function. Object contains the data and labels.
sorted_rules_RF	RandomForest_sorted_rules object generated by sort_rules_RF function
gene_repetition	interger indicating how many times the gene is allowed to be repeated in the pairs/rules. Default is 1.
rules_altogether	integer indicating how many unique rules to be used from altogether slot in the sorted rules object. Default is 200.
rules_one_vs_rest	integer indicating how many unique rules to be used from each one_vs_rest slot (class vs rest slots) in the sorted rules object. Default is 200.
run_boruta	logical indicates if Boruta algorithm should be run before building the RF model. Boruta will be used to remove the unimportant rules. Default is FALSE.
plot_boruta	logical indicates if Boruta is allowed to plot importance history plots. Default is FALSE.
boruta_args	list of argument to be passed to Boruta algorithm. Default for doTrace argument in Boruta is 1.
num.trees	an integer. Number of trees. Default is 500. It is recommended to increase num.trees in case of having large number of features (ranger function argument).
min.node.size	an integer. Minimal node size. Default is 1. (ranger function argument)
importance	Variable importance mode, should be one of 'impurity', 'impurity_corrected', 'permutation'. Defualt is 'impurity' (ranger function argument)
write.forest	Save ranger.forest object, required for prediction. Default is TRUE. (ranger function argument). Always should be true to return the trained RF model.
keep.inbag	Save how often observations are in-bag in each tree. Default is TRUE. (ranger function argument). Needed for co-clustering heatmaps.
probability	Grow a probability forest as in Malley et al. (2012). Default is TRUE. (ranger function argument). Needed to plot probability scores in the binary rules heatmaps. If TRUE, when the classifier is used to predict a sample class the user will get "ranger.prediction" object with a matrix with scores for each class. If FALSE, the classifier will give a "ranger.prediction" object with the predicted class without scores for each class.

verbose — a logical value indicating whether processing messages will be printed or not. Default is TRUE.

... — any additional arguments to be passed to ranger function (i.e. random forest function) in ranger package. For example, seed for reproducibility. Note, seed argument will be used also for Burota run.

Details

train_RF function extracts the lists of the sorted rules from altogether and classes slots, then it keep the top rules those fit with the gene_repetition number, this step reduces the number of the rule dramatically. From the left rules, rules_altogether and rules_one_vs_rest determine how many rules will be used. In case rules_altogether and rules_one_vs_rest were larger than the left rules then all the rules will be used. After that these rules will be pooled in one list and fid to Boruta function to remove the unimportant rules. Then random forest will be trained on the important rules.

Value

train_RF returns rule_based_RandomForest object which contains the final RF classifier and the used genes and rules in the final model.

Boruta results are also included in the object.

The object also contains TrainingMatrix which is a binary matrix for the rules in the training data, this is used for imputation purposes during the prediction if the sample misses some values.

Author(s)

Nour-al-dain Marzouka <nour-al-dain.marzouka at med.lu.se>

Examples

```
# generate random data
Data <- matrix(runif(800), nrow=100, ncol=80,
               dimnames = list(paste0("G",1:100), paste0("S",1:80)))

# generate random labels
L <- sample(x = c("A","B","C","D"), size = 80, replace = TRUE)

# generate random platform labels
P <- sample(c("P1","P2","P3"), size = 80, replace = TRUE)

# create data object
object <- ReadData(Data = Data,
                   Labels = L,
                   Platform = P,
                   verbose = FALSE)

# sort genes
genes_RF <- sort_genes_RF(data_object = object,
                          seed=123456, verbose = FALSE)

# to get an idea of how many genes we will use
```

```
# and how many rules will be generated
# summary_genes_RF(sorted_genes_RF = genes_RF,
#                  genes_altogether = c(10,20,50,100,150,200),
#                  genes_one_vs_rest = c(10,20,50,100,150,200))

# creat and sort rules
# rules_RF <- sort_rules_RF(data_object = object,
#                           sorted_genes_RF = genes_RF,
#                           genes_altogether = 100,
#                           genes_one_vs_rest = 100,
#                           seed=123456,
#                           verbose = FALSE)

# parameters <- data.frame(
#   gene_repetition=c(3,2,1),
#   rules_one_vs_rest=0,
#   rules_altogether=c(2,3,10),
#   run_boruta=c(FALSE,"produce_error",FALSE),
#   plot_boruta = FALSE,
#   num.trees=c(100,200,300),
#   stringsAsFactors = FALSE)

# parameters

# test <- optimize_RF(data_object = object,
#                     sorted_rules_RF = rules_RF,
#                     test_object = NULL,
#                     overall = c("Accuracy"),
#                     byclass = NULL, verbose = FALSE,
#                     parameters = parameters)
# test
# test$summary[which.max(test$summary$Accuracy),]
#
# # train the final model
# # it is preferred to increase the number of trees and rules in case you have
# # large number of samples and features
# # for quick example, we have small number of trees and rules here
# # based on the optimize_RF results we will select the parameters
# RF_classifier <- train_RF(data_object = object,
#                           gene_repetition = 1,
#                           rules_altogether = 0,
#                           rules_one_vs_rest = 10,
#                           run_boruta = FALSE,
#                           plot_boruta = FALSE,
#                           probability = TRUE,
#                           num.trees = 300,
#                           sorted_rules_RF = rules_RF,
#                           boruta_args = list(),
#                           verbose = TRUE)
#
# # training accuracy
# # get the prediction labels
# # if the classifier trained using probability = FALSE
```

```
# training_pred <- RF_classifier$RF_scheme$RF_classifier$predictions
# if (is.factor(training_pred)) {
#   x <- as.character(training_pred)
# }
#
# # if the classifier trained using probability = TRUE
# if (is.matrix(training_pred)) {
#   x <- colnames(training_pred)[max.col(training_pred)]
# }
#
# # training accuracy
# caret::confusionMatrix(data =factor(x),
#                 reference = factor(object$data$Labels),
#                 mode = "everything")

# not to run
# visualize the binary rules in training dataset
# plot_binary_RF(Data = object,
#                classifier = RF_classifier,
#                prediction = NULL, as_training = TRUE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Training data")

# not to run
# predict
# test_object # any test data
# results <- predict_RF(classifier = RF_classifier, impute = TRUE,
#                       Data = test_object)
#
# # visualize the binary rules in training dataset
# plot_binary_RF(Data = test_object,
#                classifier = RF_classifier,
#                prediction = results, as_training = FALSE,
#                show_scores = TRUE,
#                top_anno = "ref",
#                show_predictions = TRUE,
#                title = "Test data")
```

Index

∗ **methods**
print-methods, 24

do_dunn_test, 2

filter_genes_TSP, 2

group_TSP, 5

optimize_RF, 6

plot_binary_RF, 10
plot_binary_TSP, 15
predict_one_vs_rest_TSP, 18
predict_RF, 20
print,multiclassPairs_object-method
(print-methods), 24
print,OnevsrestScheme_genes_SB-method
(print-methods), 24
print,OnevsrestScheme_SB-method
(print-methods), 24
print,RandomForest_sorted_genes-method
(print-methods), 24
print,RandomForest_sorted_rules-method
(print-methods), 24
print,rule_based_RandomForest-method
(print-methods), 24
print-methods, 24
proximity_matrix_RF, 25

ReadData, 29

sort_genes_RF, 30
sort_rules_RF, 35
summary_genes_RF, 39

train_one_vs_rest_TSP, 43
train_RF, 45