pyRforest: A comprehensive tool for genomic data analysis featuring scikit-learn Random Forests in R

Contents

1 Introduction to pyRforest

Welcome to pyRforest: A comprehensive tool for genomic data analysis featuring scikit-learn Random Forests in R. Tailored for expression data, such as RNA-seq or Microarray, pyRforest is built for bioinformaticians and researchers looking to explore the relationship between biological features and matched binary or categorical outcome variables using Random Forest models. Please read on for instructions that will guide you through pyRforest's seamless integration of scikit-learn's Random Forest methodologies (imported to R via reticulate) for model development, evaluation, SHAPley additive explanations, and our custom feature reduction approach by way of rank-based permutation. You will also be directed you through our integration with clusterProfiler and g:Profiler for Gene Ontology and Enrichment Analysis.

This vignette will guide you through:

- pyRforest's integration of scikit-learn's Random Forest methodologies (via `reticulate`).
- Data preparation for use in pyRforest (and scikit-learn).
- Model development and evaluation.
- Our custom feature reduction approach using rank-based permutation.
- Integration with SHAP for feature directionality interpretation.
- Integration with clusterProfiler and g:Profiler for gene set enrichment & gene ontology analysis.

1.1 About pyRforest

Consider pyRforest if you:

- Have a genomic dataset (e.g. RNA-seq or Microarray) with a binary or categorical (including multiclass) outcome variables.
- Aim to develop and evaluate Random Forest models from Python's scikit-learn through our custom R wrapper.
- Seek permutation and rank-based feature reduction, yielding a list of significant features.
- Require a memory efficient way to build Random Forest models on large genomic datasets in R.

pyRforest offers:

- Python scikit-learn's superior Random Forest class balancing, BayesSearchCV and GRIDSearchCV hyperparameter tuning with cross-validation, efficient memory management, sparse-data handling, and parallelization.
- A final list of reduced features with corresponding p-values for indicating statistical significance.
- Publication-ready plots to help visualize the feature reduction approach.
- SHAP values for understanding feature influence on the outcome.
- Integration with clusterProfiler and g:Profiler for gene set enrichment & GO analysis.

For more information and to contribute to the pyRforest package, please visit the GitHub repository.

1.2 License

2 R Package Installation and Setup

For the best experience, we recommend the installation of R version 4.3.1, and RStudio by Posit version 023.09.1+494 or later, for installing and running the pyRforest package.

```
# Installation: 1. Install devtools if not already installed:
install.packages("devtools")
library(devtools)

# 2. Use devtools to install pyRforest
devtools::install_github("tkolisnik/pyRforest")

# 3. Load pyRforest
library(pyRforest)
```

This package has been developed on Apple mac (M1 arm64). It does not work properly on windows machines due to incompatibilties with the back-end paralellization of model building. It has been tested and works on Linux and posit Cloud.

This package leverages scikit-learn's random forest model using the R package reticulate to allow us to execute Python code within R. Therefore, in order to use this package you must first install python 3.9.18 and reticulate and optionally conda, and setup either a reticulate virtualenv (linux) or a conda environment (mac) with python 3.9.18, scikit-learn 3.1.2, numpy 1.26.0, shap 0.43.0 and their dependencies.

This package requires the R package reticulate (version 1.34.0 recommended) reticulate is installed and loaded by running the commands:

```
# First ensure reticulate is installed:
if (!requireNamespace("reticulate")) {
    install.packages("reticulate")
}

# If using reticulate's virtualenv (recommended for Linux):

# Initial setup:
library(reticulate)
reticulate::install_python(version = "3.9.18")
Sys.setenv(WORKON_HOME = "virtualenvs")
reticulate::virtualenv_create(envname = "pyRforest-venv", python_version = "3.9.18",
    packages = c("scikit-learn==1.3.2", "numpy==1.26.0", "shap==0.43.0", "scikit-optimize==0.10.2",
        "memory_profiler==0.61.0"))
reticulate::use_virtualenv("pyRforest-venv")

# Subsequent usage after initial setup:
library(reticulate)
Sys.setenv(WORKON_HOME = "virtualenvs")
use_virtualenv("pyRforest-venv")

# If using conda (recommended for M-series Arm64 Macs):
library(reticulate)
# Install python 3.9.18 if not already installed by downloading it from:
# https://www.python.org/downloads/macos/ Ensure it is installed to the folder:
# /usr/local/bin/python3.
reticulate::use_python("/usr/local/bin/python3")
reticulate::conda_install(envname = "pyRforest-conda", python_version = "3.9.18",
    packages = c("scikit-learn==1.3.2", "numpy==1.26.0", "shap==0.43.0", "scikit-optimize==0.10.2",
```

```
        "memory_profiler==0.61.0"))
reticulate::use_condaenv("pyRforest-conda", required = TRUE)

# Subsequent usage after initial setup:
library(reticulate)
use_condaenv("pyRforest-conda", required = TRUE)

# You can check if environment setup has worked by running:
reticulate::py_config()

# If environment installation has worked, the previous command will result in
# an output message similar to:

# python: /opt/homebrew/Caskroom/miniconda/base/envs/pyRforest-conda/bin/python
# libpython:
# /opt/homebrew/Caskroom/miniconda/base/envs/pyRforest-conda/lib/libpython3.9.dylib
# pythonhome:
# /opt/homebrew/Caskroom/miniconda/base/envs/pyRforest-conda:/.../pyRforest-conda
# version: 3.9.18 numpy:
# /opt/homebrew/Caskroom/miniconda/base/envs/pyRforest-conda/lib/python3.9/sit...
# numpy_version: 1.26.0 sklearn:
# /opt/homebrew/Caskroom/miniconda/base/envs/pyRforest-conda/lib/python3.9/sit...
# NOTE: Python version was forced by use_python() function

# You may now proceed to using the package
```

2.1 Installation Notes:

You can also build your conda environment outside of Rstudio in the terminal.

For more information on dependency installation see:

https://www.python.org/downloads/

https://brew.sh/

https://conda.io/projects/conda/en/latest/user-guide/install/index.html

https://docs.conda.io/projects/miniconda/en/latest/

2.2 Troubleshooting

If when trying to load your environment you get the following error:

ERROR: The requested version of Python ('path/env/python') cannot be used, as another version of Python ('/usr/local/bin/python3') has already been initialized. Please restart the R session if you need to attach reticulate to a different version of Python. Error in use_python(python, required = required) : failed to initialize requested version of Python

Then press session -> restart R -> then reload your environment

3 Data Preparation

Load and prepare your datasets. This R package comes pre-loaded with 3 demo RNA-seq datasets which can be loaded using:

```r
data("demo_rnaseq_data", package = "pyRforest") # Recommended default dataset with binary targets
data("demo_rnaseq_data_categorical", package = "pyRforest") # Demo dataset with 2 categorical targets
data("demo_rnaseq_data_multiclass", package = "pyRforest") # Demo dataset with 3 categorical targets
```

Structuring Your Data for Machine Learning

To facilitate the use of our machine-learning tools, we recommend structuring your expression and target data similarly to our demo dataset. This section will guide you through this process and provide illustrative examples.

Your data should be saved as a list containing four tibbles. Each tibble functions similarly to a data frame but offers additional benefits in handling and manipulating large datasets. Below is an overview of the structure and purpose of each tibble:

Training, Validation, and Testing Tibbles

Identifier: A unique identifier for each row (e.g., "2041-04-12-1"), representing a sample or patient. This is crucial for tracking and referencing data points.

Target: A variable indicating the outcome/target to be predicted. May be binary, categorical (including more than 2 classes), but must not be continuous.

Gene Expression Data: The remaining columns, such as ENSG00000253497 and ENSG00000238493, contain gene expression data, here represented as RNA-seq counts. Each column corresponds to a specific gene.

Target Categories Tibble

Includes the full list of identifiers (identifier) from the training, validation, and testing datasets.

The target_categorical Optional. This column provides the outcome variable in a categorical format. The target column provides the outcome variable in binary format. This dataframe is primarily used to assist with mapping for SHAP and to assist the user keep track of their data.

Essential for mapping and identification purposes across all datasets.

Best Practices for Data Splitting

In machine learning, datasets are typically split into three parts:

Training Dataset: Used to train the machine learning model. It learns to make predictions or classify data based on this set.

Validation Dataset: Used to tune the parameters of the model and provide an unbiased evaluation of a model fit during the training phase. It helps in providing a check against overfitting.

Optional Testing Dataset: Used to provide an unbiased evaluation of the final model fit. It's only used after the model training and validation phases are complete.

These splits help in effectively training the model and evaluating its performance in a way that is generalizable to new, unseen data.

Standard Recommendations for Dataset Splitting: It's a common practice to use around 60-80% of the data for training, and the remaining 20-40% split between validation and testing. However, these ratios can vary based on the size and specifics of your dataset.

Example data structures expected by pyRforest: *These are tibbles which should be saved into one list variable*

Table 1: training_data

identifier	target	ENSG00000253497	ENSG00000238493	ENSG00000248122	ENSG00000175544
2041-04-12-1	1	0.234937	0	0.19781	0.395993
2041-08-12-1	1	0.000000	0	0.00000	1.162733

identifier	target	ENSG00000253497	ENSG00000238493	ENSG00000248122	ENSG00000175544
2041-13-12-1	1	2.252000	0	0.00000	0.373436
2041-18-12-1	1	0.000000	0	0.00000	0.026357
2041-21-12-1	1	0.000000	0	0.00000	0.692587
2041-34-12-1	1	0.000000	0	0.00000	0.818505
2706-004-12-1	1	6.374389	0	0.00000	2.758556
2706-011-12-1	1	0.000000	0	0.00000	0.073407
2706-023-12-1	0	0.000010	0	0.00000	0.111399
2706-030-12-1	1	0.000000	0	0.00000	1.450572

Table 2: testing_data

identifier	target	ENSG00000253497	ENSG00000238493	ENSG00000248122	ENSG00000175544
2041-07-12-1	0	1e-05	0	0.000000	3.828424
2706-025-12-1	1	0e+00	0	0.000000	0.986655
2706-065-12-1	0	1e-05	0	0.000000	0.039046
2706-071-12-1	0	1e-05	0	0.000000	0.272935
2706-091-12-1	0	1e-05	0	0.231253	0.706623
2706-098-12-1	0	1e-05	0	0.000000	1.469622
2706-112-12-1	1	0e+00	0	0.000000	0.310703
2706-164-12-1	0	1e-05	0	0.000000	0.982244
2706-172-12-1	1	0e+00	0	0.000000	0.145355
2706-213-12-1	1	0e+00	0	0.000000	1.056201

Table 3: validation_data

identifier	target	ENSG00000253497	ENSG00000238493	ENSG00000248122	ENSG00000175544
2706-002-12-1	1	0	0	0.000000	1.087608
2706-057-12-1	0	0	0	0.000000	0.334466
2706-061-12-1	0	0	0	0.000000	0.150133
2706-063-12-1	0	0	0	0.000000	3.247098
2706-076-12-1	1	0	0	0.000000	0.093304
2706-095-12-1	1	0	0	0.000000	0.550842
2706-117-12-1	1	0	0	0.260355	0.752071
2706-135-12-1	1	0	0	0.000000	1.831656
2706-147-12-1	0	0	0	0.000000	0.736398
2706-188-12-1	0	0	0	0.000000	0.829256

Table 4: target_categories

identifier	target_categorical	target
2041-04-12-1	Right	1
2041-08-12-1	Right	1
2041-13-12-1	Right	1
2041-18-12-1	Right	1
2041-21-12-1	Right	1
2041-34-12-1	Right	1
2706-004-12-1	Right	1

identifier	target_categorical	target
2706-011-12-1	Right	1
2706-023-12-1	Left	0
2706-030-12-1	Right	1

Recall .RData files are created using the command: save(variable_name, file="path/to/file") and loaded using load("path/to/file")

Alternatively, instead of using the .RData format, expression data can be imported through other approaches such as CSV or TSV. Here is another example of loading data from CSV file formats. Ensure each file is structured with the correct data and columns, as shown in the tables above.

```
# Load target data:
training_data <- read_csv("/path/to/file/training_data.csv")
validation_data <- read_csv("/path/to/file/validation_data.csv")
testing_data <- read_csv("/path/to/file/testing_data.csv")
target_categories <- read_csv("/path/to/file/target_categories.csv")
```

and then these variables should be combined into a list of tibbles using:

```
# Load the tibble package
library(tibble)

# Ensure each dataset is converted to a tibble and combine them into a list
demo_rnaseq_data <- list(
  training_data = as_tibble(training_data),
  validation_data = as_tibble(validation_data),
  testing_data = as_tibble(testing_data),
  target_categories = as_tibble(target_categories)
)
```

4 Data Preprocessing

```
# This function prepares a dataset for scikit-learn random forest machine learning by
# creating a matrix from the dataset excluding certain columns, and extracting a
# target vector. It isflexible and can handle different types of data sets such as
# training, testing, or validation.
# Data must be in the correct input format, see vignette or example dataset for details.

processed_training_data <- pyRforest::create_feature_matrix(
  dataset = demo_rnaseq_data$training_data,
  set_type = "training"
  )

processed_validation_data <- pyRforest::create_feature_matrix(
  dataset = demo_rnaseq_data$validation_data,
  set_type = "validation"
  )

processed_testing_data <- pyRforest::create_feature_matrix(
  dataset = demo_rnaseq_data$testing_data,
  set_type = "testing"
  )
```

5 Model Tuning

```
# This function uses scikit-learn's python based GridSearchCV to perform hyperparameter
# tuning and training of a RandomForestClassifier. It allows for customizable parameter
# grids and includes preprocessing steps of one-hot encoding and scaling. The function
# is designed to find the best hyperparameters based on accuracy. Please reference the
# scikit-learn GridSearchCV documentation for the full description of options,
# however our defaults are comprehensive.

# Default custom_parameter_grid (for binary or 2-class categorcial targets):
tuning_results <- pyRforest::tune_and_train_rf_model_grid(
                                  X = processed_training_data$X_training_mat,
                                  y = processed_training_data$y_training_vector,
                                  cv_folds = 5,
                                  scoring = 'roc_auc',
                                  seed = 4,
                                  n_jobs = 1,
                                  n_cores = 7
                                  )

print(tuning_results$grid_search$best_params_)
print(tuning_results$grid_search$best_score_)

# Default custom_parameter_grid for multiclass targets
# 'roc_auc_ovr' or 'accuracy' should be used over roc_auc:
tuning_results <- pyRforest::tune_and_train_rf_model_grid(
                                  X = processed_training_data$X_training_mat,
                                  y = processed_training_data$y_training_vector,
                                  cv_folds = 5,
                                  scoring = 'roc_auc_ovr',
                                  seed = 4,
                                  n_jobs = 1,
                                  n_cores = 7
                                  )

print(tuning_results$grid_search$best_params_)
print(tuning_results$grid_search$best_score_)

# If not running defaults and customization of parameter is desired:
custom_parameter_grid <- list(
  bootstrap = list(TRUE),
  class_weight = list(NULL),
  max_depth = list(5L, 20L, NULL),
  n_estimators = as.integer(seq(10, 90, 10)),
  max_features = list("sqrt", 0.2),
  # It is not recommended to change criterion, as gini is required by pyRforest functions.
  criterion = list("gini"),
  warm_start = list(FALSE),
  min_samples_leaf = list(1L, 50L),
  min_samples_split = list(2L, 200L)
)

# Train a model and perform hyperparameter tuning using GridSearchCV.
# This can be extremely time, memory, and processing-power consuming,
```

```
# and scales rapidly with dataset size.
# A small dataset such as the demo data may take <5 minutes on an M1 mac with 16gb ram,
# However, a larger dataset may take hours or even days depending on the dataset size
# and number of cores used.
# It is strongly recommended to parallelize computing by increasing n_cores.
# 1 less than the total number of cores on your processor is recommended.

tuning_results <- pyRforest::tune_and_train_rf_model_grid(
                                  X = processed_training_data$X_training_mat,
                                  y = processed_training_data$y_training_vector,
                                  cv_folds = 5,
                                  scoring = 'roc_auc', #use 'roc_auc_ovr' for multiclass targets
                                  seed = 123,
                                  param_grid = custom_parameter_grid,
                                  n_jobs = 1,
                                  n_cores = 7
                                  )

# After the model is tuned, print the best parameters and the best score.
# See scikit-learn documentation for interpreting scores.
# It is recommended to re-tune on different parameters if score is not adequate.

print(tuning_results$grid_search$best_params_)
print(tuning_results$grid_search$best_score_)
best_params<-tuning_results$best_params
```

```
# For a faster alternative to the grid search shown above this function uses
# scikit-learn's python based BayesSearchCV to perform hyperparameter
# tuning and training of a RandomForestClassifier. It allows for customizable parameter
# grids and includes preprocessing steps of one-hot encoding and scaling. The function
# is designed to find the best hyperparameters based on accuracy. Please reference the
# scikit-learn BayesSearchCV documentation for the full description of options,
# however our defaults are comprehensive.

# If running default custom_parameter_grid:
tuning_results <- pyRforest::tune_and_train_rf_model_bayes(
                                  X = processed_training_data$X_training_mat,
                                  y = processed_training_data$y_training_vector,
                                  cv_folds = 5,
                                  scoring = 'roc_auc', # use 'roc_auc_ovr' for multiclass targets
                                  seed = 4,
                                  n_jobs = 1,
                                  n_cores = 7
                                  )

print(tuning_results$grid_search$best_params_)
print(tuning_results$grid_search$best_score_)
best_params<-tuning_results$best_params

# If not running defaults and customization of parameter is desired:
custom_parameter_grid <- list(
  bootstrap = list(TRUE),
  class_weight = list(NULL),
  max_depth = list(5L, 20L, NULL),
```

```
  n_estimators = as.integer(seq(10, 90, 10)),
  max_features = list("sqrt", 0.2),
  # It is not recommended to change criterion, as gini is required by pyRforest functions.
  criterion = list("gini"),
  warm_start = list(FALSE),
  min_samples_leaf = list(1L, 50L),
  min_samples_split = list(2L, 200L)
)

# Train a model and perform hyperparameter tuning using BayesSearchCV,
# While faster and more efficient than the above GridSearchCV,
# This can still be moderately time, memory, and processing-power consuming,
# and scales up with dataset size.
# A small dataset such as the demo data may take <1 minutes on an M1 mac with 16gb ram,
# However, a larger dataset may take hours depending on the dataset size
# and number of cores used.
# It is strongly recommended to parallelize computing by increasing n_cores.
# 1 less than the total number of cores on your processor is recommended.

tuning_results <- pyRforest::tune_and_train_rf_model_bayes(
                                  X = processed_training_data$X_training_mat,
                                  y = processed_training_data$y_training_vector,
                                  cv_folds = 5,
                                  scoring = 'roc_auc', #use 'roc_auc_ovr' for multiclass targets
                                  seed = 123,
                                  param_grid = custom_parameter_grid,
                                  n_jobs = 1,
                                  n_cores = 7
                                  )

# After the model is tuned, print the best parameters and the best score.
# See scikit-learn documentation for interpreting scores.
# It is recommended to re-tune on different parameters if score is not adequate.

print(tuning_results$bayes_search$best_params_)
print(tuning_results$bayes_search$best_score_)
best_params<-tuning_results$best_params
```

6 Fit and Evaluate the Random Forest model

6.1 Validation scoring

```
# This function fits a Random Forest model using the provided hyperparameters
# and training data, then evaluates its performance on a validation set.

# Fit the results to the validation set to check performance:

fitting_results <- pyRforest::fit_and_evaluate_rf(
                               best_params = tuning_results$best_params,
                               X_train = processed_training_data$X_training_mat,
                               y_train = processed_training_data$y_training_vector,
                               X_val = processed_validation_data$X_validation_mat,
```

```
                                y_val = processed_validation_data$y_validation_vector
                                )

# Print the fitting results, provides accuracy, f1 score, precision, recall
# and roc_auc scores on the model as fitted to the validation set.
print(fitting_results)

# If performance is not adequate, you may want to reconstruct your model, tweak
# your data, and rerun step 7 before proceeding.

# If you want to manually assign your parameters you can do so by running this code:
# Note: you must include an L after all integers (eg 10L) to force them to remain as
# integers during the list to python dict conversion.
best_params_manual <- list(
  bootstrap = TRUE,
  class_weight = NULL,
  criterion = 'gini',
  max_depth = 10L, # Keep the L!
  max_features = 0.2, # Decimal numbers do not need the L
  min_samples_leaf = 2L, # Keep the L!
  min_samples_split = 2L, # Keep the L!
  n_estimators = 100L, # Keep the L!
  warm_start = FALSE,
  verbose = TRUE
)

# Use reticulate to convert the R list to a Python dict.
new_best_params <- reticulate::dict(best_params_manual)
```

6.2 Testing scoring

```
# This function is also used to evaluate the testing set when the model is finalized.
# Depending on your goals, you may want to initially skip scoring on this set
# and re-build your model on the reduced feature set and re-run previous steps
# before proceeding.
# For reporting purposes. You should not run this code until your model is locked down.

fitting_results_testing <- pyRforest::fit_and_evaluate_rf(
                                    best_params = tuning_results$best_params,
                                    X_train = processed_training_data$X_training_mat,
                                    y_train = processed_training_data$y_training_vector,
                                    X_val = processed_testing_data$X_testing_mat,
                                    y_val = processed_testing_data$y_testing_vector
                                    )

# Print the fitting results, provides accuracy, f1 score, precision, recall
# and roc_auc scores on the model as fitted to the validation set.
print(fitting_results_testing)
```

7 Permute the model and obtain feature importances

```
# This function fits a Random Forest model and calculates the true and permuted feature
# importances. It performs permutations on the target variable to generate permuted
# importances for comparison.
# This step is time consuming and scales with training set size and the number of
# permutations used. (Approx. 5 mins to 3 hours+).

feat_importances <- pyRforest::calculate_feature_importances(
                                    model = fitting_results$model,
                                    X_train = processed_training_data$X_training_mat,
                                    y_train = processed_training_data$y_training_vector,
                                    n_permutations = 1000
                                    )

# Print top features from model pre-Rank-based feature reduction
print(feat_importances$top_features)
```

8 Calculate Significance Metrics

8.1 Calculate Quantiles

```
# Calculate quantiles
# This function computes the mean, lower, and upper quantiles of permuted feature
# importance scores and compares them with the observed importance scores from the
# true values.
# It is used to assess the significance of feature importances from random forest
# models by comparing them against a distribution of importances obtained through
# permutation. Necessary for figures.
quantile_data <- pyRforest::calculate_quantiles(
                          truevalues = feat_importances$true_importances,
                          permutedvalues = feat_importances$permuted_importances,
                          alpha = 0.05
                          )
```

8.2 P-values for entire feature set

```
# Calculate p-value for the entire feature set
# This function calculates the proportion-based p-value for the entire feature set
# by comparing the observed sum of absolute deviations from the mean feature importance
# to those from permuted data. It is particularly useful in permutation tests to assess
# the statistical significance of the observed data.
pvalue_set <- pyRforest::calculate_full_set_pvalue(
                        permutedvalues = feat_importances$permuted_importances,
                        quantiledata = quantile_data
                        )
```

8.3 P-values for each rank

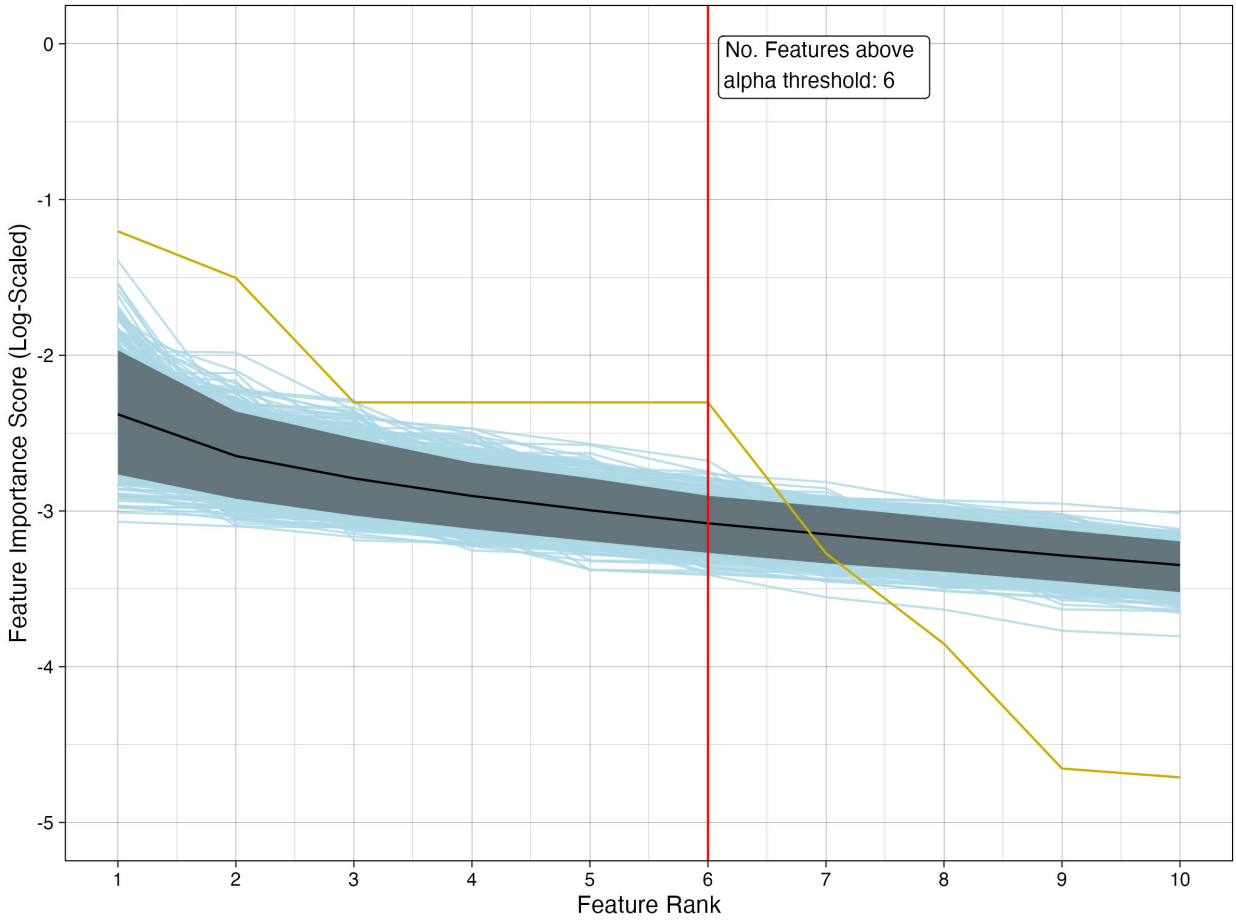
```
# Calculate p-value for each rank
# This function calculates p-values for each feature rank in the dataset by comparing
```

```r
# the observed feature importances against the distribution of importances in the
# permuted data.
# It returns ranks and their respective p-values and proportions up to a
# specified alpha threshold.
# Note: P-values returned as 0 actually represent p-values less than the smallest
# detectable limit. With 1,000 permutations, the smallest detectable p-value
# is 0.001 (1/n_permutations).
# Therefore, a reported p-value of 0 should be interpreted as p < 0.001.

pvalues_ranks <- pyRforest::calculate_ranked_based_pvalues(
                        truevalues = feat_importances$true_importances,
                        permutedvalues = feat_importances$permuted_importances,
                        alpha = 0.05
                        )

# Print top features from model post-Rank based feature reduction
print(pvalues_ranks)
```

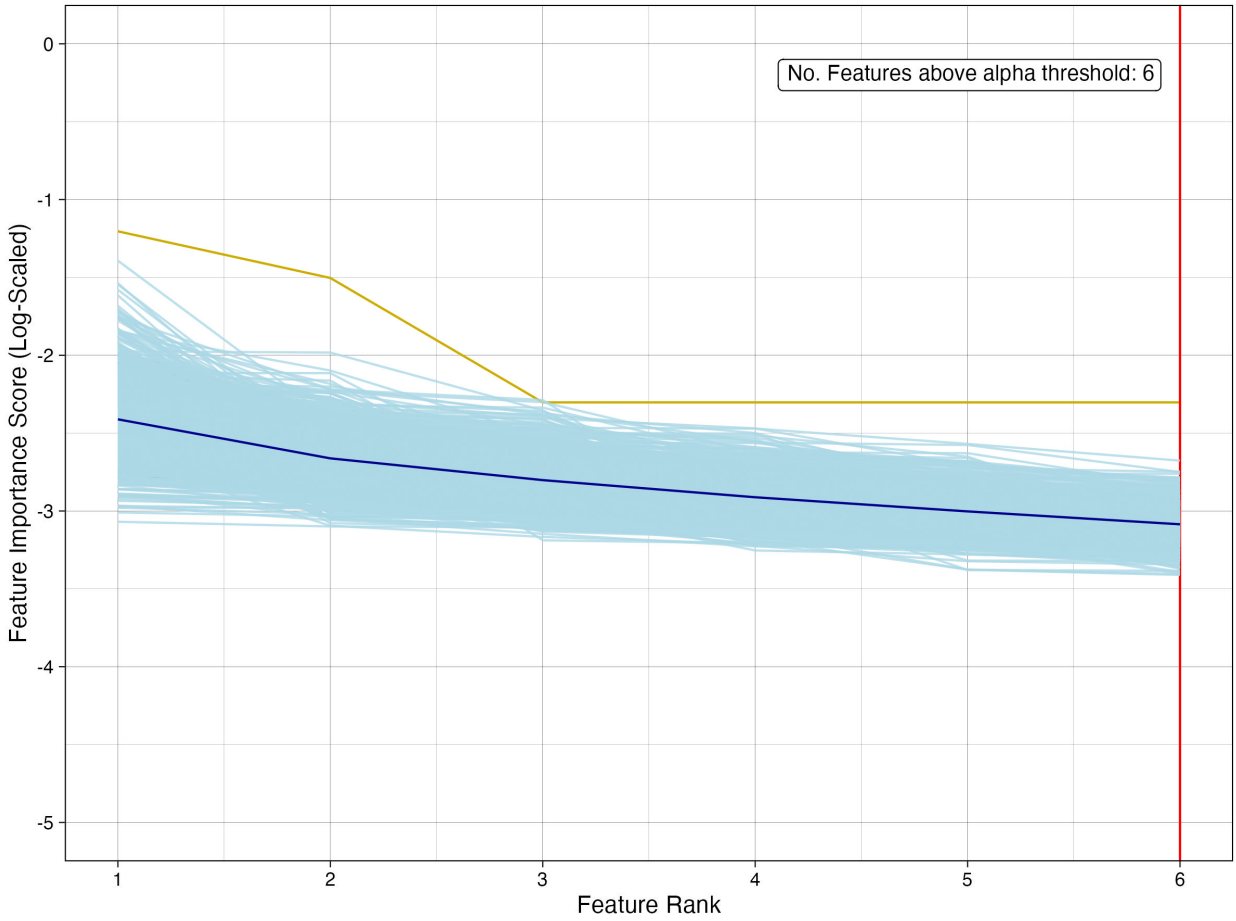
9 Generate Plots

9.1 Feature Importance Plots

```r
# Generate Full feature importance plot
# This function generates a figure which shows the true observed data plotted against the
# permuted data, by rank. The intersection of the true data with the upper quartile is
# shown, which we recommend as a significance cutoff. Note: There are ample parameters
# for controlling the axes scale, label location, and zoom, because of data variability,
# you will almost certainly have to adjust these to fit your plot.

fiplot_full <- pyRforest::generate_fi_rank_plot(
  permutedvalues = feat_importances$permuted_importances,
  quantiledata = quantile_data,
  xlimitmin = 1,
  xlimitmax = 10,
  ylimitmin= -5,
  ylimitmax = 0,
  labelhorizontaladjust = -0.05,
  labelverticaladjust = 1.5,
  focusedView = FALSE,
  logOn = TRUE
  )
# Tip:
# For best quality it is recommended to save plots using the ggsave() function
# e.g. ggsave("/path/fiplot_full.jpeg", plot = last_plot(),
#             width = 8, height = 6, dpi = 300)
print(fiplot_full)
```

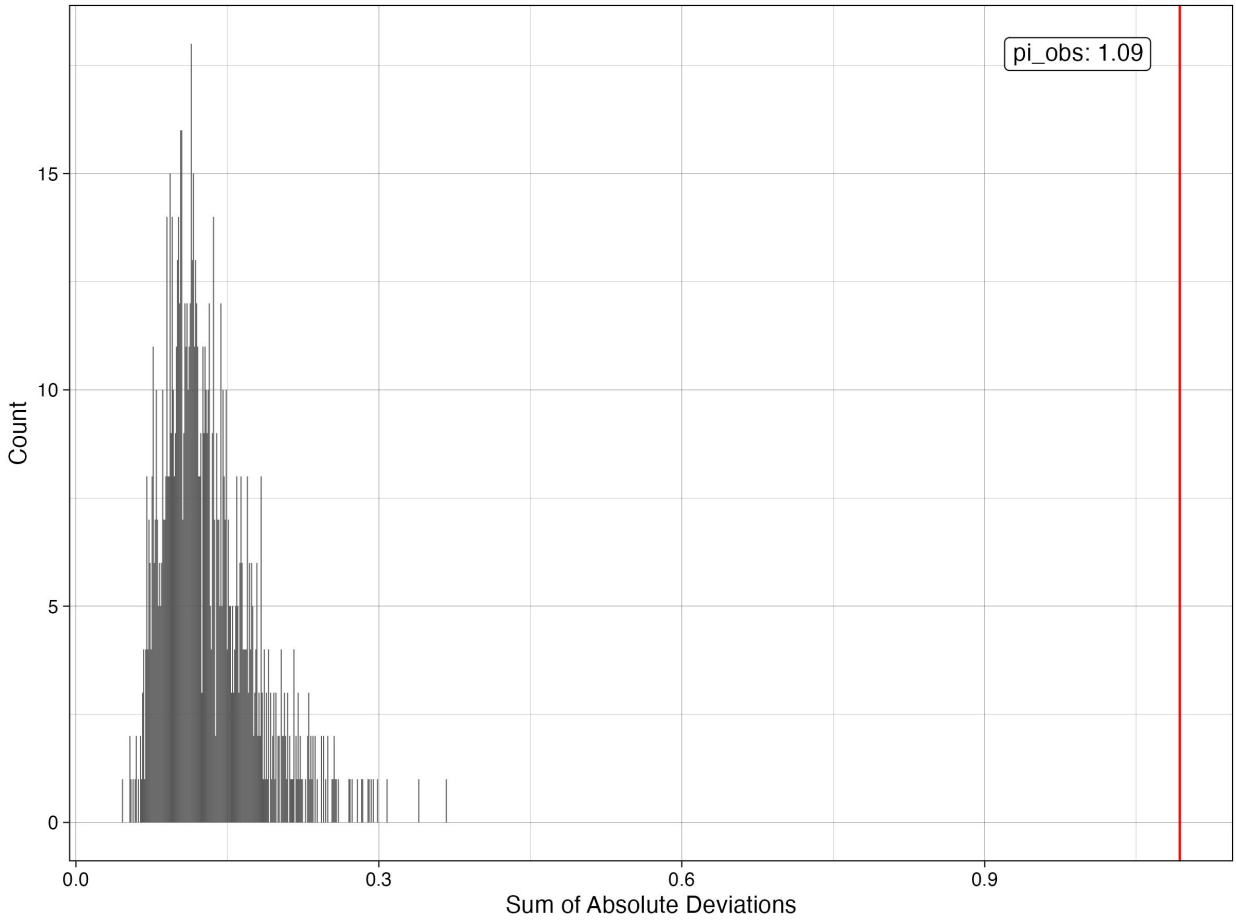



```
# Same plot as above, but zoomed to show only the significant results
fiplot_focused <- pyRforest::generate_fi_rank_plot(
  permutedvalues = feat_importances$permuted_importances,
  quantiledata = quantile_data,
  xlimitmin = 1,
  xlimitmax = 10,
  ylimitmin= -5,
  ylimitmax = 0,
  labelhorizontaladjust = 1.05,
  labelverticaladjust = 1.5,
  focusedView = TRUE,
  logOn = TRUE
  )

print(fiplot_focused)
```



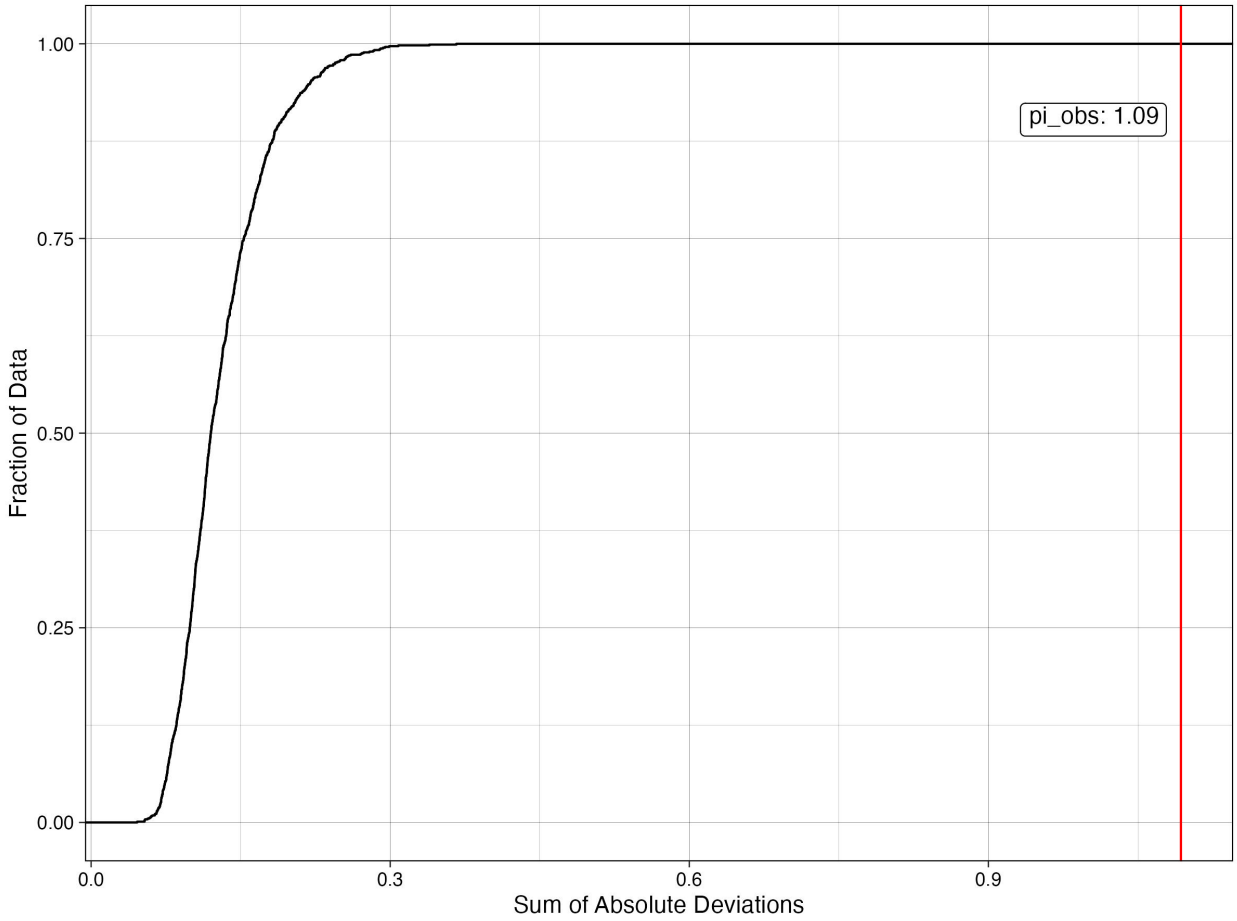
9.2 pi Histogram

```
# Generate pi Histogram plot
# Creates a histogram showing the sum of absolute deviations by count, the measures
# used to calculate the pi statistics used in the p-value calculation for the set.
pihist_plot <- pyRforest::generate_pi_histogram(
  permutedvalues = feat_importances$permuted_importances,
  quantiledata = quantile_data
  )

print(pihist_plot)
```



9.3 ECDF Plot

```
# Generate ECDF plot
# Creates a plot of the empirical cumulative density function showing the sum of
# absolute deviations by fraction of data.
ecdf_plot <- pyRforest::generate_pi_ECDF_plot(
  permutedvalues = feat_importances$permuted_importances,
  quantiledata = quantile_data)

print(ecdf_plot)
```



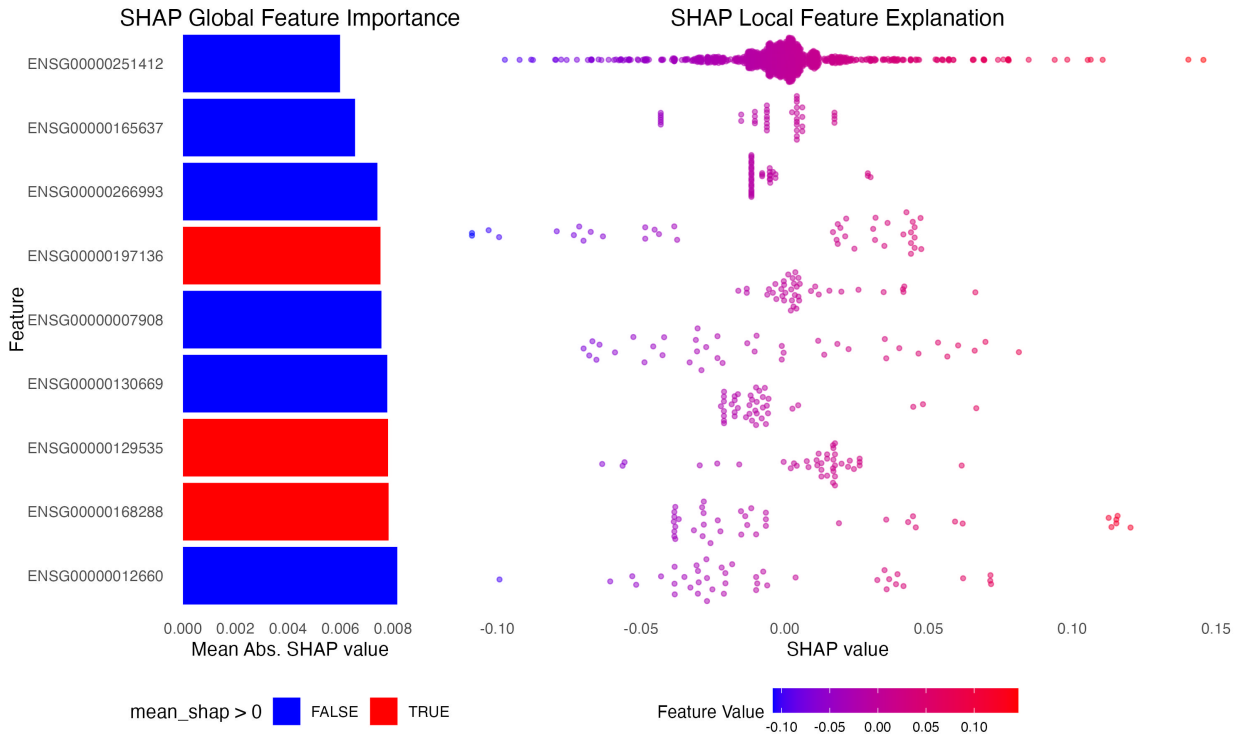
Advanced Features

Explore advanced functionalities of pyRforest such as SHAP value analysis, or use it to plug into additional gene ontology, enrichment and gprofiler modules.

10 SHAP Analysis

```
# Produce a list of SHAP values
# This function calculates SHAP values for a given dataset using a provided model.
# It then identifies significant features based on the SHAP values for the specified
# class.
# Additionally, it prepares a long-format data frame of individual SHAP values
# suitable for visualization.

shapvals <- pyRforest::calculate_SHAP_values(
  model = fitting_results$model,
  data = processed_training_data$X_training_mat,
  class_index = 1, # In demo, 1 represents Right
  shap_std_dev_factor = 0.5
  )

# SHAP analysis is ran for one direction at a time.
# If a models classes are: ['Left' 'Right'] their order represents the index (starting at 0) that shoul
```

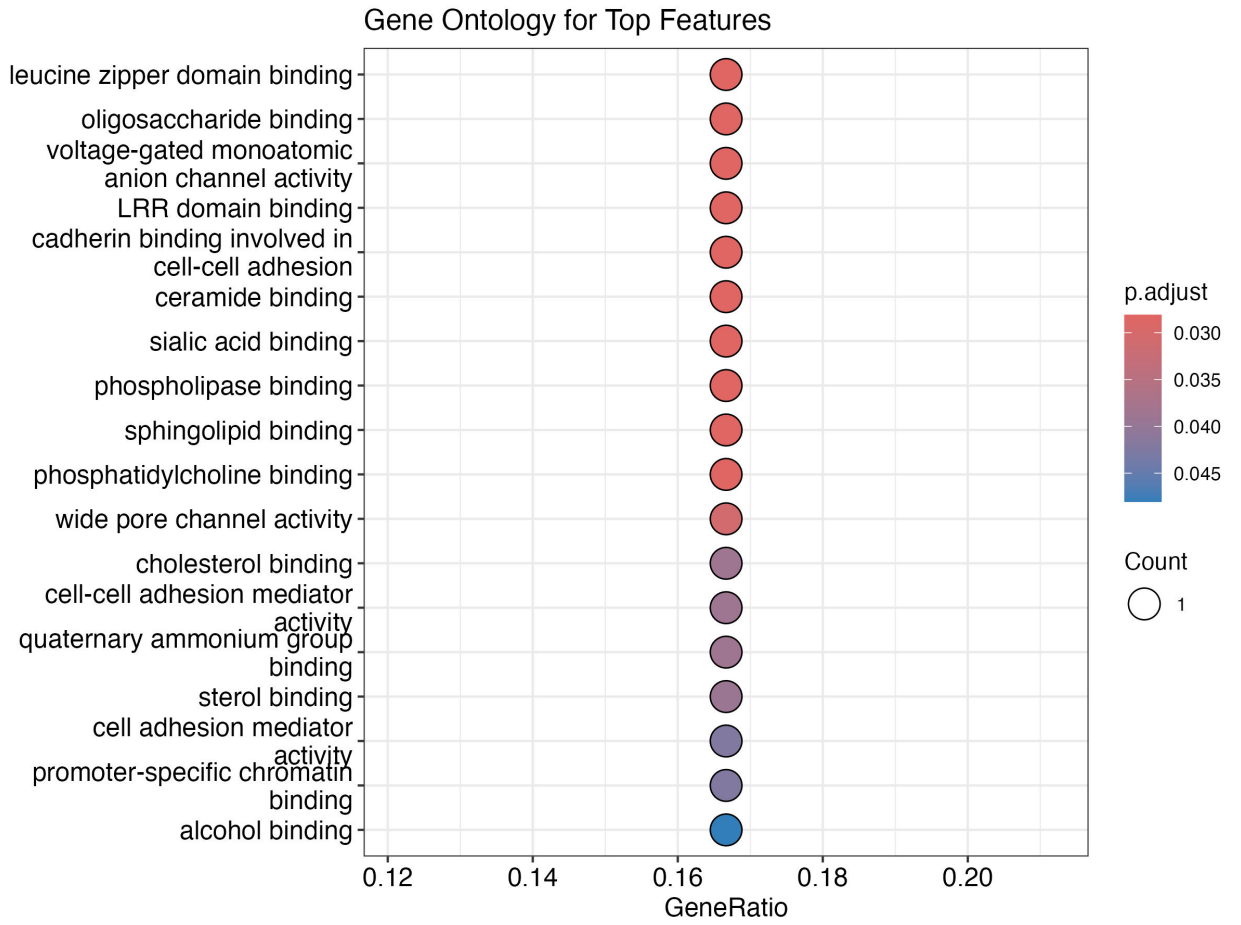
```
# If none passed default is 1 for  binary/two class or 0 for multiclass.
# This index represents the class for which SHAP values will be calculated,
# with feature contributions shown relative to that specific class.
# For multiclass you may want to run your shap analysis x times, one for each class.

# Running the function once with the class_index left at default will help explain.

# Produce the SHAP Plots
# Generates a combined bar and beeswarm plot to show direction,
# as well as global and local feature importances.
shapplot <- pyRforest::generate_shap_plots(
  mean_shap_values = shapvals$significant_features,
  long_shap_data = shapvals$long_shap_data
  )

print(shapplot)
```



11 GO Enrichment Analysis

```
library(clusterProfiler)
library(org.Hs.eg.db) # Assuming human data, see bioconductor for other organism dbs.
library(enrichplot)

# We have 3 lists of significant features from our data that can be explored further

# 1. The 'raw' list of siginificant features from the RF model:
print(feat_importances$top_features)
```

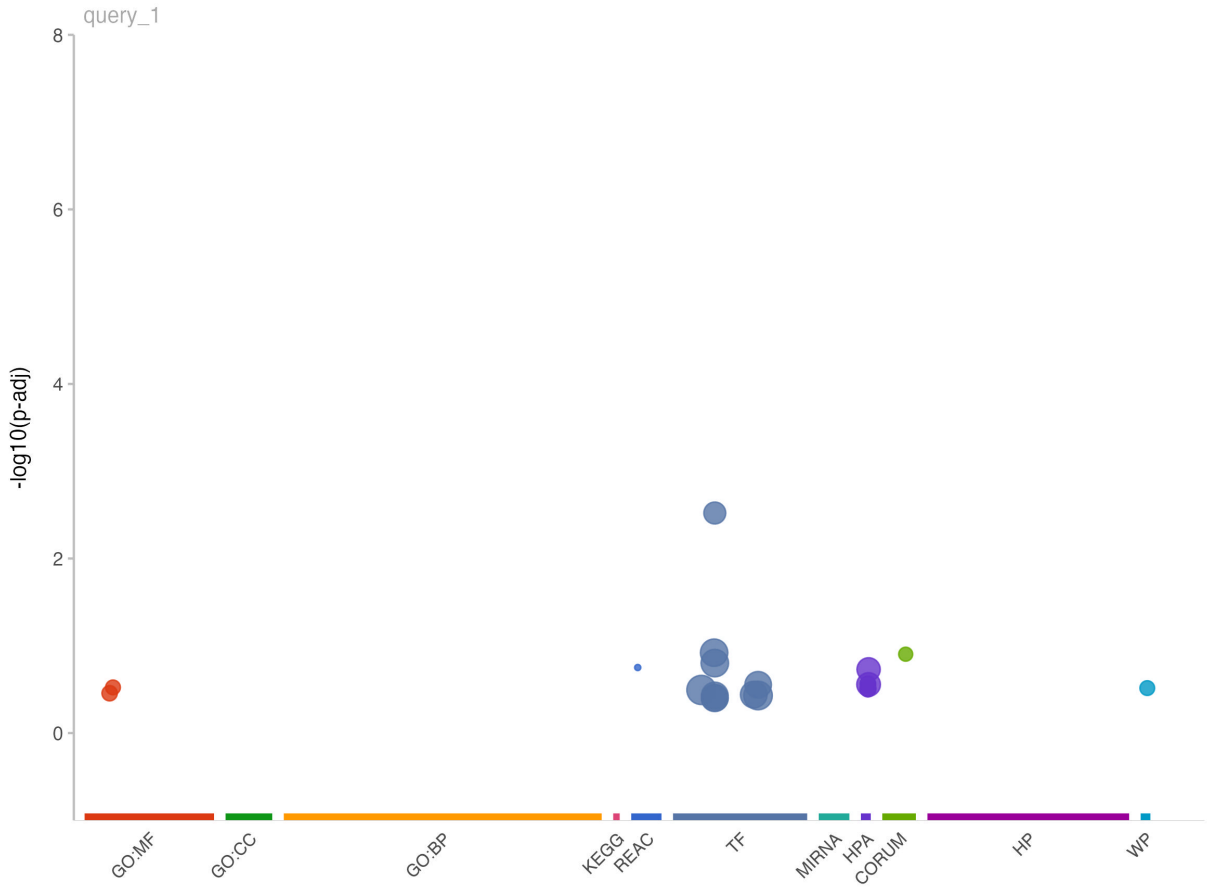
```r
# 2. The list of significant features after rank-based permutation feature reduction
# This is the most conservative, statistically stringent list:
print(pvalues_ranks)

# 3. The list of significant features from the SHAP analysis
# Can be further divided by directionality +/-:
print(shapvals$significant_features)

# Subsequent steps can be ran on any of these feature lists,
# although if the list is too small these steps may not produce results.
features_list <- shapvals$significant_features$feature # Features from SHAP

# Alternatively:
#features_list <- pvalues_ranks$feature # Post-rank based feature reduction list
#features_list <- feat_importances$top_features # Pre-feature reduction list

# Run the GO Enrichment Analysis
# From clusterProfiler: GO Enrichment Analysis of a gene set.
# Given a vector of genes, this function will return the enrichment GO categories.

ego <- clusterProfiler::enrichGO(
                gene = features_list,
                OrgDb = org.Hs.eg.db,
                keyType = "ENSEMBL",
                ont = "ALL", # consider BP, CC, and MF ontologies
                pAdjustMethod = "BH", # adjust p-values for multiple testing
                readable = TRUE)  # convert Entrez IDs back to gene symbols

# Plot the gene ontology
enrich_plot <- enrichplot::dotplot(ego, showCategory=20) +
                                    ggtitle("Gene Ontology for Top Features")
print(enrich_plot)
```



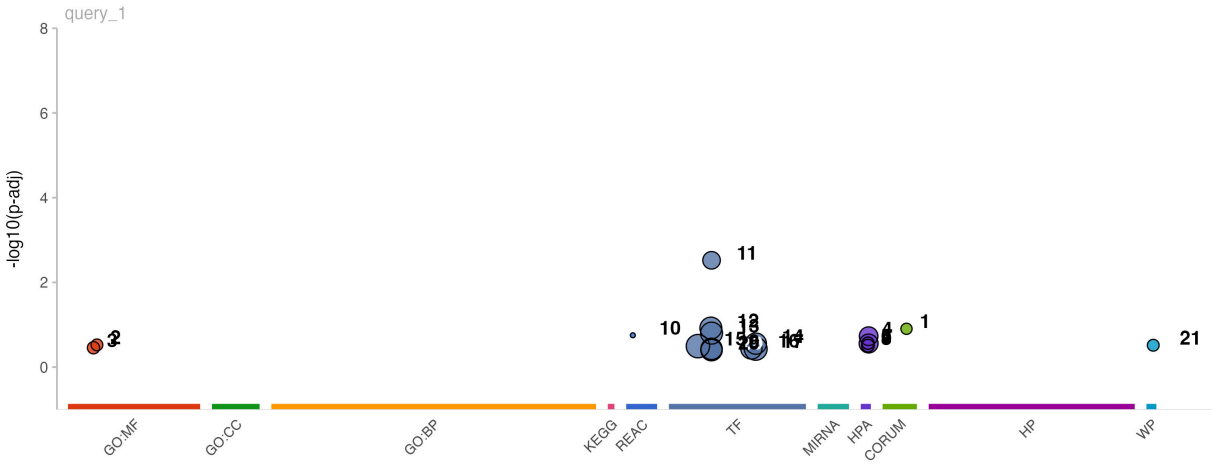
12 gProfiler Analysis

```
library(gprofiler2)

# Perform the enrichment analysis
# From gprofiler2: Interface to the g:Profiler tool g:GOSt
# See https://biit.cs.ut.ee/gprofiler/gost for functional enrichments analysis of
# gene lists.

gprofiler_results <- gprofiler2::gost(query = features_list, # List of gene vectors
                                      organism = "hsapiens",
                                      ordered_query = TRUE,
                                      user_threshold = 0.05
                                      )

# Create gProfiler Manhattan plot
# From gprofiler2: This function creates a Manhattan plot out of the results from the
# gprofiler2::gost() function.
gostplot_mh_plot <- gprofiler2::gostplot(gostres = gprofiler_results,
                                         capped = FALSE,
                                         interactive = F
                                         )
```

```r
print(gostplot_mh_plot)
```




```r
# Create gProfiler Manhattan plot with table of terms beneath it
# From gprofiler2: This function allows to highlight a list of selected terms on the
# Manhattan plot created with the gprofiler2::gostplot() function.
gostplot_mh_plot_with_table <- gprofiler2::publish_gostplot(
                            p = gostplot_mh_plot,
                            highlight_terms = gprofiler_results$result$term_id,
                            width = NA,
                            height = NA,
                            filename = NULL # For saving
                            )

print(gostplot_mh_plot_with_table)
```

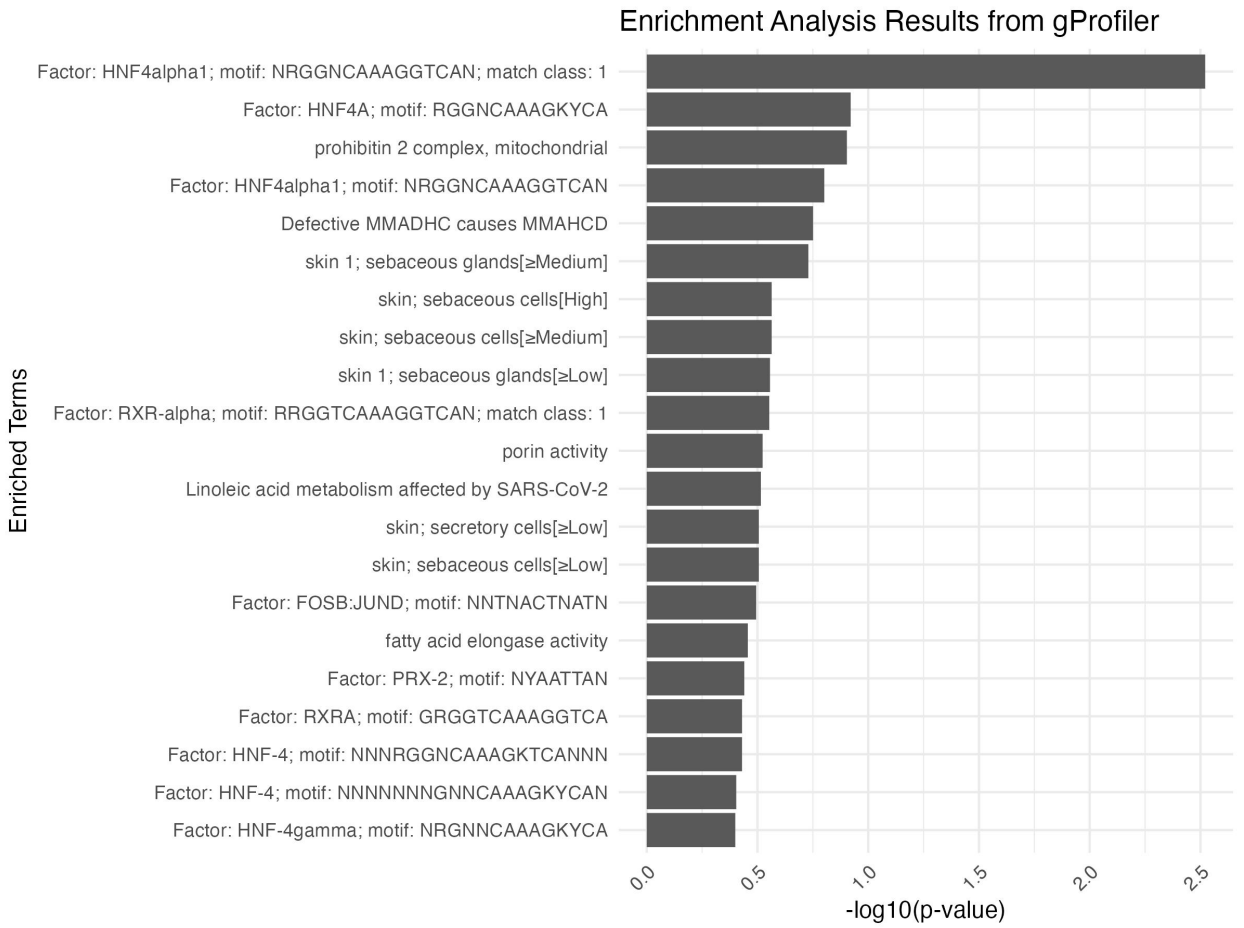


id	source	term_id	term_name	term_size		p_value
1	CORUM	CORUM:6289	prohibitin 2 complex, mitochondrial	5		1.2e-01
2	GO:MF	GO:0015288	porin activity	6		3.0e-01
3	GO:MF	GO:0009922	fatty acid elongase activity	7		3.5e-01
4	HPA	HPA:0461482	skin 1; sebaceous glands[≥Medium]	165		1.9e-01
5	HPA	HPA:0450613	skin; sebaceous cells[High]	7		2.7e-01
6	HPA	HPA:0450612	skin; sebaceous cells[≥Medium]	7		2.7e-01
7	HPA	HPA:0461481	skin 1; sebaceous glands[≥Low]	202		2.8e-01
8	HPA	HPA:0450621	skin; secretory cells[≥Low]	8		3.1e-01
9	HPA	HPA:0450611	skin; sebaceous cells[≥Low]	8		3.1e-01
10	REAC	REAC:R-HSA-3359473	Defective MMADHC causes MMAHCD	2		1.8e-01
11	TF	TF:M00411_1	Factor: HNF4alpha1; motif: NRGGNCAAAGGTCAN; match class: 1	86		3.0e-03
12	TF	TF:M02220	Factor: HNF4A; motif: RGGNCAAAGKYCA	1801		1.2e-01
13	TF	TF:M00411	Factor: HNF4alpha1; motif: NRGGNCAAAGGTCAN	1906		1.6e-01
14	TF	TF:M11774_1	Factor: RXR-alpha; motif: RRGGTCAAAGGTCAN; match class: 1	1066		2.8e-01
15	TF	TF:M08928	Factor: FOSB:JUND; motif: NNTNACTNATN	6228		3.2e-01
16	TF	TF:M10886	Factor: PRX-2; motif: NYAATTAN	1140		3.6e-01
17	TF	TF:M04489	Factor: RXRA; motif: GRGGTCAAAGGTCA	3917		3.7e-01
18	TF	TF:M00134	Factor: HNF-4; motif: NNNRGGNCAAAGKTCANNN	1147		3.7e-01
19	TF	TF:M03828	Factor: HNF-4; motif: NNNNNNNGNNCAAAGKYCAN	1165		3.9e-01
20	TF	TF:M09627	Factor: HNF-4gamma; motif: NRGNNCAAAGKYCA	1168		4.0e-01
21	WP	WP:WP4853	Linoleic acid metabolism affected by SARS-CoV-2	6		3.1e-01

g:Profiler (biit.cs.ut.ee/gprofiler)

```
# Assign gprofiler results
result_df <- gprofiler_results$result

# Filter for significant results
significant_gprofiler_results <- result_df[result_df$significant, ]

# Create a new column for -log10(p-value)
significant_gprofiler_results$log_p_value <- -log10(significant_results$p_value)

# Print significant gprofiler results:
print(significant_gprofiler_results)

# Plot significant gProfiler results
library(ggplot2)
```

```r
signif_gprofiler_resplot <- ggplot(
                                  significant_gprofiler_results,
                                  aes(x = reorder(term_name, log_p_value),
                                      y = log_p_value)) +
                                  geom_bar(stat = "identity") +
                                  theme_minimal() +
                                  coord_flip() +
                                  labs(x = "Enriched Terms", y = "-log10(p-value)",
                                    title = "Enrichment Analysis Results from gProfiler") +
                                  theme(axis.text.x = element_text(angle = 45, hjust = 1)
                                  )
print(signif_gprofiler_resplot)
```



13 Conclusion

pyRforest provides comprehensive tools for bioinformatics analysis, including features for machine learning, feature importance evaluation, and more. This vignette covers the basics to get you started. For more detailed documentation on the use of each function, please refer to the function help pages within the package using the ?pyRforest feature.

This package was created by Tyler Kolisnik, with support and assistance from Dr. Olin Silander, Dr. Adam Smith & Faeze Keshavarz.

Webpage: www.github.com/tkolisnik/pyRforest | Contact: tkolisnik@gmail.com