Using the SimpleEnsemble Package

Group 21

Contents

Introduction

`SimpleEnsemble` is designed to simplify the process of creating ensemble models. This vignette demonstrates how to use this package to build ensemble models. This package supports linear, logistic, lasso, ridge, elastic net, and support vector machines. It also provides various functionalities such as fine tuning linear models, find top K predictors, perform bagging with bootstrapping, create and combine ensemble models using a meta learner. The authors of this package are graduate students at Stony Brook University. Those are Deepti Mulbagal Venkatesh, Hamim Shabbir Halim, Keerthi Bhaskara Sirapu, and Saiteja Kalam.

The development of the "SimpleEnsemble" package adheres to principles of reproducibility and modularity, ensuring that the package is both dependable and versatile. While the package can be accommodated with much more feature base, our aim to satisfy the project requirements for the course AMS597:Statistical Computing at SBU. The structure of "SimpleEnsemble" is thoughtfully organized into four distinct modules, each serving a specific function within the ensemble modeling process:

1. **Models Module**: This core module houses all the machine learning models supported by the package. It provides a robust foundation for building predictive models, including linear regression, logistic regression, ridge regression, lasso regression, elastic net, and support vector machines.
2. **Bagging Module**: Dedicated to the implementation of the bagging technique, this module facilitates bootstrap sampling and aggregation. It allows users to enhance the stability and accuracy of their machine learning models by reducing variance and preventing overfitting.
3. **Ensemble Module**: This module is designed to integrate various predictive models into a cohesive ensemble framework. It provides functionality to combine predictions from multiple models, using techniques like averaging or more complex methods like stacking, thereby leveraging the strengths of individual models to improve overall performance.
4. **Preprocessing Module**: Essential for effective model training, this module includes the `pre.screen` function, which assists in selecting the top 'K' predictors. This function is useful when a dataset has significantly more predictors than the number of data points. This feature selection process is crucial for simplifying models, improving their interpretability, and reducing computational demands.

By compartmentalizing functionalities into these modules, "SimpleEnsemble" not only enhances the user experience through a clean and organized interface but also promotes ease of maintenance and scalability. Users can navigate the package intuitively, applying complex ensemble techniques with straightforward commands.

Models

The `fit.model` function is designed to be a versatile tool in the "SimpleEnsemble" R package. This function exemplifies the integration of traditional statistical methods with modern machine learning techniques, enabling users to fit a wide range of predictive models using custom parameters for tuning and optimization.

Overview of `fit.model`

The function allows for the fitting of several types of predictive models, including:

- **Linear Models:** Suitable for continuous outcomes where the relationship between predictors and the response is assumed to be linear.
- **Logistic Models:** Used for binary classification tasks where the outcome is categorical with two classes (e.g., success/failure).
- **Ridge and Lasso Models:** These are regularization methods that help in handling multicollinearity, model overfitting, and feature selection by introducing a penalty term to the loss function.
- **Elastic Net Models:** Combines penalties of lasso and ridge regression, ideal for situations where there are correlations among features.
- **Support Vector Machines (SVM):** Offers flexibility through the use of different kernels and can be used for both classification and regression tasks.

Function Parameters

- `X`: The predictor variables matrix, where each column represents a feature and each row represents an observation.
- `y`: The response or target variable, which can be continuous for regression tasks or binary for classification tasks.
- `model.type`: Specifies the type of model to fit. The function supports 'linear', 'logistic', 'ridge', 'lasso', 'elastic', and 'svm'.
- `alpha`: The mixing parameter relevant for elastic net, which balances the weight of ridge and lasso regularization components.
- `lambda`: The regularization parameter controlling the amount of shrinkage: larger values specify stronger regularization. If not specified, cross-validation is used to determine the optimal lambda automatically for ridge, lasso, and elastic net models.
- `intercept`: A logical value indicating whether to include an intercept term in the model.
- `kernel`: Specifies the type of kernel to be used in SVM models. Options include 'linear', 'polynomial', 'radial', and 'sigmoid'.

Error Handling

Comprehensive error handling ensures that the function inputs are valid, enhancing robustness: - Checks if `X` is a matrix and `y` is a vector. - Validates that `model.type` and `kernel` are among the supported options. - Checks the appropriateness of `alpha` values.

Examples of fit.model function in Models.R

Regression task, continuous response variable For all the models below (except linear and svm), lambda is found using cross validation, when explicitly not specified by user.

```
X <- as.matrix(mtcars[, 2:8])
y <- mtcars$mpg
print(y)
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

```
model.ridge <- fit.model(X, y, model.type = "ridge")
coef(model.ridge)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                       s0
## (Intercept) 27.069142638
## cyl         -0.585720915
## disp        -0.002665499
## hp          -0.017250972
## drat         1.351339797
## wt          -2.376116067
## qsec         0.132345002
## vs           0.487565400
```

```
model.lasso <- fit.model(X, y, model.type = "lasso", lambda = 0.002)
coef(model.lasso)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                      s0
## (Intercept) 26.09712532
## cyl         -0.85606108
## disp         0.01283925
## hp          -0.01725197
## drat         1.29171495
## wt          -4.19601293
## qsec         0.43822166
## vs          -0.24935142
```

```
model.linear <- fit.model(X, y, model.type = "linear")
coef(model.linear)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                      s0
## (Intercept) 25.98206526
## cyl         -0.86714516
## disp         0.01317795
## hp          -0.01723728
## drat         1.30129622
```

```
## wt          -4.22471558
## qsec         0.44815887
## vs          -0.28624157
```

```
model.elastic <- fit.model(X, y, model.type = "elastic")
coef(model.elastic)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                      s0
## (Intercept) 32.82412879
## cyl         -0.75425170
## disp         .
## hp          -0.01788099
## drat         0.76469180
## wt          -2.73426567
## qsec         0.03023098
## vs           0.14472287
```

```
model.svm <- fit.model(X, y, model.type = "svm")
#you can use this model object of SVM type to predict.
```

```
X <- as.matrix(mtcars[, 1:7])
y <- as.integer(mtcars$am)
print(y)
```

Classification task, binary response variable

```
##  [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1
```

```
model.logistic <- fit.model(X, y, model.type = "logistic")
coef(model.logistic)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                       s0
## (Intercept) 194.52403458
## mpg           3.18992763
## cyl          -6.80000770
## disp         -0.08775309
## hp            0.09269542
## drat         16.67559705
## wt           -5.95611667
## qsec        -14.74924507
```

```
model.ridge <- fit.model(X, y, model.type = "ridge")
coef(model.ridge)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                       s0
```

```
## (Intercept)  8.371281704
## mpg          0.123247870
## cyl         -0.236385209
## disp        -0.005524762
## hp           0.005688463
## drat         1.773412101
## wt          -1.019586354
## qsec        -0.727111814
```

```
model.lasso <- fit.model(X, y, model.type = "lasso")
coef(model.lasso)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                       s0
## (Intercept) 29.18937307
## mpg          0.15247627
## cyl          .
## disp        -0.01237948
## hp           .
## drat         2.32609113
## wt          -2.77927361
## qsec        -1.70567226
```

```
model.elastic <- fit.model(X, y, model.type = "elastic")
coef(model.elastic)
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                       s0
## (Intercept) 36.36491048
## mpg          0.36993778
## cyl         -0.75173484
## disp        -0.01807086
## hp           0.01442919
## drat         3.71069989
## wt          -2.67714685
## qsec        -2.48920561
```

```
model.svm <- fit.model(X, y, model.type = "svm")
#you can use this model object of SVM type to predict.
```

Bagging

The bagging function provided is designed to be a versatile tool for both regression and classification problems in R, capable of handling a variety of predictive modeling techniques including linear models, logistic regression, ridge regression, lasso regression, elastic-net models, and SVMs. The function employs the bootstrap aggregation (bagging) technique to improve model stability and accuracy by reducing variance and avoiding overfitting.

Key Features

Model Flexibility: The function supports multiple model types, making it broadly applicable for different types of predictive modeling tasks. Users can specify the model type according to their data and the problem context.

Regularization: For models like ridge, lasso, and elastic-net, parameters `alpha` and `lambda` are provided to control the regularization strength and the mix of L1 and L2 penalties, helping in feature selection and preventing overfitting.

Feature Importance: An importance threshold can be set, below which features can be excluded from the model. This is particularly useful in scenarios where reducing model complexity could enhance generalizability.

Multiple Combining Methods: The function allows for different methods to combine model predictions, one is averaging (method = "average") where the final predictions are average of all the prediction vectors from each model which results from a particular bootstrap sample. Another is The other is using a meta learner (method = "stacking"), where each prediction vector of a model (from a given bootstrap sample), is given as input to a support vector machines (svm) and the original response variable (y), is given as target variable to meta learner (svm). This will reduce the workload of the user.

Intercept Management: Users have the option to include or exclude the intercept in the model, providing additional flexibility depending on whether data are centered or if modeling constraints require its exclusion.

Variable Importance: The `bag()` function in the provided code snippet calculates variable importance based on the absolute values of coefficients derived from the models fitted within the bagging algorithm. This importance measurement is determined within the bootstrapping loop for models that include coefficients such as ridge, lasso, and elastic net models. A variable importance matrix (zero matrix) of size (nrow = no.of predictors * ncol = no.of bootstrap samples) is initiated. Breakdown of how variable importance is calculated:

1. **Bootstrapping and Model Fitting**:
    - The function performs bootstrapping, which involves repeatedly sampling from the dataset with replacement. For each bootstrap sample, a specified model is fitted.
2. **Importance Calculation**:
    - If the method is linear or logistic: a variable with a non-zero coefficient is considered important.
    - If the method is ridge, lasso, elastic: a variable is considered important if the coefficient is greater than a certain threshold (which is 1e-4.)
    - If a variable is important, the designated placeholder with be changed to 1 in variable importance matrix.
3. **Aggregation**:
    - After all bootstrapping iterations are complete, the variable importances across all samples are aggregated to provide a final importance measure for each variable. This is sum (on rows) of the importance measures across the bootstrapped models.

This approach allows for assessing which features contribute most significantly to the model, aiding in feature selection and understanding model behavior. The use of absolute coefficients as importance measures is particularly common in regularization models like lasso and ridge, where the magnitude of coefficients directly relates to the strength of each feature's effect after accounting for multicollinearity and overfitting control through penalties.

Examples of bag function in Bagging.R

```
data <- read.csv("H:/My Drive/Spring24/AMS597/SimpleEnsembleGroup21/SimpleEnsembleGroup21/data/Enigma.cs

# Define X and y
```

```r
X <- data[,1:13]
y <- as.vector(data$y)

# Split the data into training and testing sets
set.seed(123) # for reproducibility

# Create the training set indices using createDataPartition
library(caret)
```

Classification task - Binary response variable

```
## Loading required package: ggplot2

##
## Attaching package: 'ggplot2'

## The following object is masked from 'package:randomForest':
##
##     margin

## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:SimpleEnsemble':
##
##     bag
```

```r
training_samples <- createDataPartition(y, p=0.75, list=FALSE)

# Define training and testing sets
X_train <- as.matrix(X[training_samples, ])
y_train <- y[training_samples]
X_test <- as.matrix(X[-training_samples, ])
y_test <- y[-training_samples]
result1 <- SimpleEnsemble::bag(X_train, y_train, X_test,y_test, model.type = "lasso", R = 10)
print(paste("train accuracy:", result1$combined.train.metric))
```

```
## [1] "train accuracy: 0.911330049261084"
```

```r
print(paste("test accuracy:", result1$combined.test.metric))
```

```
## [1] "test accuracy: 0.908695652173913"
```

```r
print(result1$importances)
```

```
##  [1] 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```
result1 <- SimpleEnsemble::bag(X_train, y_train, X_test,y_test, model.type = "lasso", R = 10, combine.m
print(paste("train accuracy:", result1$combined.train.metric))
```

```
## [1] "train accuracy: 0.391190959142278"
```

```
print(paste("test accuracy:", result1$combined.test.metric))
```

```
## [1] "test accuracy: 0.392173913043478"
```

```
print(result1$importances)
```

```
##  [1] 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```
# Load the data
data <- read.csv("H:/My Drive/Spring24/AMS597/SimpleEnsembleGroup21/SimpleEnsembleGroup21/data/Questionl


# Define X and y
X <- data[, -which(names(data) == "w4")]
y <- as.vector(data$y)

# Split the data into training and testing sets
set.seed(123) # for reproducibility
training_samples <- createDataPartition(y, p=0.75, list=FALSE)

# Define training and testing sets
X_train <- as.matrix(X[training_samples, ])
y_train <- y[training_samples]
X_test <- as.matrix(X[-training_samples, ])
y_test <- y[-training_samples]


result1 <- SimpleEnsemble::bag(X_train, y_train, X_test,y_test, model.type = "elastic", R = 10)
print(paste("train RMSE:", result1$combined.train.metric))
```

Regression task - Continuous response variable

```
## [1] "train RMSE: 75.3067953451363"
```

```
print(paste("test RMSE:", result1$combined.test.metric))
```

```
## [1] "test RMSE: 62.0899716446451"
```

```
print(result1$importances)
```

```
##  [1] 10  0  1 10  0  1  0  0  0  0  6  9  0 10
```

```r
result1 <- SimpleEnsemble::bag(X_train, y_train, X_test,y_test, model.type = "elastic", R = 10, combine
print(paste("train RMSE:", result1$combined.train.metric))
```

```
## [1] "train RMSE: 47.8579960089462"
```

```r
print(paste("train RMSE:", result1$combined.test.metric))
```

```
## [1] "train RMSE: 44.8406673781767"
```

```r
print(result1$importances)
```

```
##  [1] 10  0  0 10  0  1  0  0  0  0  4 10  0 10
```

Top K features

The `pre.screen()` function is a powerful tool for feature selection designed to operate with different types of response variables and apply several statistical methods for determining feature relevance. The function is versatile, supporting methods suitable for both continuous and categorical data, and it's structured to handle binary responses with specific methods.

Function Description

The `pre.screen()` function selects the top `k` most relevant features from a given dataset based on the specified method of feature selection. It supports four main methods: **PCA (Principal Component Analysis)** Suitable for: Continuous X Response (y): Works with both continuous and binary, mainly to reduce dimensionality without consideration of y. Details: PCA does not inherently consider the relationship between predictors and the response variable; it's primarily used for dimensionality reduction or feature extraction based on variance. **Random Forest** Suitable for: Both continuous and binary X Response (y): Works with both continuous and binary responses. Details: Random Forest is versatile as it can handle various types of data and compute feature importance based on the decrease in node impurity. **Correlation** Suitable for: Continuous X Response (y): Continuous only. Details: Correlation measures the strength and direction of a linear relationship between two continuous variables. Not suitable for categorical data unless converted to dummy variables which can distort the analysis. **Chi-Square** Suitable for: Categorical/binary X Response (y): Categorical/binary only. Details: Chi-square tests are used to determine whether there's a significant association between two categorical variables. It requires frequency counts from categorical/binary data.

Parameters

- **X**: A matrix of predictor variables.
- **y**: A vector representing the response variable.
- **k**: The number of features to select. Must not exceed the number of columns in `X`.
- **method**: The method used for feature selection. Must be one of "correlation", "chi-square", "pca", or "randomForest".

Returns

A list containing: - **selected.indices**: Indices of the selected features. - **selected.data**: The subset of `X` corresponding to the selected features.

Error Handling

The function includes robust error handling to ensure that the inputs are appropriate for the selected method: - Checks if `X` is a matrix and `y` is a vector. - Validates that the number of features to select does not exceed the number of available predictors. - Ensures that the specified method is supported. - Verifies data types and structures are suitable for the selected method, e.g., numeric predictors for correlation, categorical predictors for chi-square.

Examples for pre.screen function in Preprocessing.R

```
data <- read.delim("https://www.ams.sunysb.edu/~pfkuan/Teaching/AMS597/Data/leukemiaDataSet.txt", heade
data$Group <- ifelse(data$Group == "ALL", 1, 0)
```

```
y <- data$Gene2238
X <- as.matrix(data[, -c(1,2238)])
prescreen.result.corr <- pre.screen(X, y, 10, method = "correlation")
print("correlation")
```

continuous X, continuous y

```
## [1] "correlation"
```

```
print(prescreen.result.corr$selected.indices)
```

```
##  [1] 2237 2109 2329 2200 2681 1550 1358   69 2865 1194
```

```
prescreen.result.pca <- pre.screen(X, y, 10, method = "pca")
print("pca")
```

```
## [1] "pca"
```

```
print(prescreen.result.pca$selected.indices)
```

```
##  [1] 2663 2992 2993 2049 2662 2069 2246 3570  631 2994
```

```
prescreen.result.rf <- pre.screen(X, y, 10, method = "randomForest")
print("random forest")
```

```
## [1] "random forest"
```

```
print(prescreen.result.rf$selected.indices)
```

```
##  [1] 2237 2681   69 1559 1017  383 2109  631 2329  272
```

```r
y <- data$Group
X <- as.matrix(data[, -c(1)])
prescreen.result.rf.cont <- pre.screen(X, y, 10, method = "randomForest")
print("random forest")
```

continuous X, binary y

```
## [1] "random forest"
```

```r
print(prescreen.result.rf.cont$selected.indices)
```

```
##  [1]  956  456  874  436 3441 2481 1182 3038 2789  979
```

```r
data <- read.delim("https://www.ams.sunysb.edu/~pfkuan/Teaching/AMS597/Data/leukemiaDataSet.txt", heade
data$Group <- ifelse(data$Group == "ALL", 1, 0)
y <- as.vector(data$Gene1)
X <- as.matrix(data[, -c(1,2)])
prescreen.result <- pre.screen(X, y, 10, method = "chi-square")
```

```
## Error in pre.screen(X, y, 10, method = "chi-square"): Error: Chi-square method requires both predict
```

Ensemble Learning

The `ensemble()` function in R is designed to facilitate ensemble learning by combining predictions from multiple predictive models. This function allows users to specify different types of models, such as linear regression, logistic regression, ridge regression, lasso regression, elastic net, and support vector machines (SVMs). The predictions from these models are then combined using either a simple averaging method or a more complex stacking method using an SVM as a meta-learner.

Function Details

Parameters:

- **X_train**: A numeric matrix containing the predictor variables. This matrix is required for fitting all the specified models.
- **y_train**: A numeric vector of the response variable which the models predict.
- **X_test**: A numeric matrix containing the predictor variables. This matrix is required for predicting values all the specified models.
- **y_test**: An optional numeric vector of the response variable, to evaluate model.
- **model.specs**: A list of lists where each inner list specifies the type of model to be fitted and its parameters. Each list must specify:
  - `model`: Type of model (e.g., "linear", "logistic", etc.).
  - `alpha`, `lambda`: Regularization parameters, applicable for ridge, lasso, and elastic net models. These should be set to `NULL` if not used.
  - `intercept`: Indicates whether to include an intercept in the model.
  - `kernel`: Specifies the kernel type for SVM models.
- **combine.method**: Method to combine model predictions. Options are:
  - `"average"`: Calculates the simple average of predictions from all models.
  - `"stacking"`: Uses an SVM to learn how to optimally combine the model predictions.

Return:

- A vector of predictions based on the chosen method for combining model outputs.

Error Handling:

- The function checks if `X` is a matrix and `y` is a vector, throwing an error if these conditions are not met.
- It validates whether all specified models in `model.specs` are supported.

Examples for ensembele function in Ensemble.R

```
# Load the data
data <- read.csv("H:/My Drive/Spring24/AMS597/SimpleEnsembleGroup21/SimpleEnsembleGroup21/data/Enigma.cs

# Define X and y
X <- data[,1:13]
y <- as.vector(data$y)

# Split the data into training and testing sets
set.seed(123) # for reproducibility

# Create the training set indices using createDataPartition
library(caret)
training_samples <- createDataPartition(y, p=0.75, list=FALSE)

# Define training and testing sets
X_train <- as.matrix(X[training_samples, ])
y_train <- y[training_samples]
X_test <- as.matrix(X[-training_samples, ])
y_test <- y[-training_samples]

model.specs <- list(
  list(model="ridge", lambda=0.1),
  list(model="lasso", lambda=0.1),
  list(model="svm", kernel="radial")
)

predictions <- SimpleEnsemble::ensemble(X_train, y_train, X_test, y_test, model.specs)
print(paste("train accuracy:", predictions$combined.train.metric))
```

Classification task

```
## [1] "train accuracy: 0.926398145465083"
```

```
print(paste("test accuracy:", predictions$combined.test.metric))
```

```
## [1] "test accuracy: 0.917391304347826"
```

```
predictions <- SimpleEnsemble::ensemble(X_train, y_train, X_test, y_test, model.specs, combine.method =
print(paste("train accuracy:", predictions$combined.train.metric))
```

```
## [1] "train accuracy: 0.374674007534048"
```

```
print(paste("test accuracy:", predictions$combined.test.metric))
```

```
## [1] "test accuracy: 0.383478260869565"
```

```
# Load the data
data <- read.csv("H:/My Drive/Spring24/AMS597/SimpleEnsembleGroup21/SimpleEnsembleGroup21/data/QuestionI


# Define X and y
X <- data[, -which(names(data) == "w4")]
y <- as.vector(data$y)

# Split the data into training and testing sets
set.seed(123) # for reproducibility
training_samples <- createDataPartition(y, p=0.75, list=FALSE)

# Define training and testing sets
X_train <- as.matrix(X[training_samples, ])
y_train <- y[training_samples]
X_test <- as.matrix(X[-training_samples, ])
y_test <- y[-training_samples]

model.specs <- list(
  list(model="ridge", lambda=0.1),
  list(model="lasso", lambda=0.1),
  list(model="svm", kernel="radial")
)

result1 <- SimpleEnsemble::ensemble(X_train, y_train, X_test, y_test, model.specs)
print(paste("train RMSE:", result1$combined.train.metric))
```

Regression task

```
## [1] "train RMSE: 144.549103205019"
```

```
print(paste("test RMSE:", result1$combined.test.metric))
```

```
## [1] "test RMSE: 97.3994947402617"
```

```
result1 <- SimpleEnsemble::ensemble(X_train, y_train, X_test, y_test, model.specs, combine.method = "av
print(paste("train RMSE:", result1$combined.train.metric))
```

```
## [1] "train RMSE: 94.3544568415787"
```

```r
print(paste("test RMSE:", result1$combined.test.metric))
```

```
## [1] "test RMSE: 153.906009527393"
```