Variable importance and selection using random forests

Abderezzak Amimer , Madi Kassymbekov

17/04/2021

1. Presentation of the subject

This project is intended in proposing a tutorial for variable importance and variable selection in random forests using different R resources. The methods used in the following sections are purely designed for a prediction setting and not an inference.

1.3 Random Forests

Random forests are in the family of ensemble methods. The principle is fairly simple. It combines many regression or classification trees using several bootstrap samples from the training data and randomly selecting a subset of the explanatory variables at each node in order to split the data. Amongst the random subset, one variable is selected using CART or Inference method to split instances. Once each tree is fully grown, predictions are made by taking the average of all trees in the forest.

1.1 Variable importance

Variable importance can be defined as the contribution of each predictor to the model. It is usually represented as a ranking of the variables based on the effect they have on response variable studied. Knowing which variables significantly impacts the quality of the predictions helps analysts weed out those that are not necessary and help , in certain cases, improve the quality of the predictions.

1.2 Variable selection

Variable selection consists of removing certain predictors from a model in order to improve the quality of a prediction. Following Kohavi and John[1] and Guyon and Elisseff (2003)[2], three groups of methods are distinguishable. First, “wrapper”, which looks closely at the prediction performance of a model to make variable selection. Second, “filter”, the score of variable importance is not based on a model design and finally, “embedded” which combines both model estimation and variable selection.

Outline

This project is organized as follows. A description of important literature and the most recent algorithms used in R regarding variable importance and selection using trees. Next, an overview of the methods to be used in the tutorials. To continue, a section will be dedicated to review R resources that will be used on data sets found in different R libraries to which noise was added.This last part will be a demonstration of how to perform variable selection and compute variable importance using the references presented.

2. Literature review

2.1 Random forest

Random forests are based on decision trees with bootstrap aggregation called "bagging". It is a method used to generate multiple versions of a predictor to get an average predictor. Randomness is introduced in each predictor by making bootstrap replicates. As low bias is the feature of tree based models, high variance is one of its drawbacks dependent on the number of features that leads to larger trees and potential vulnerability to over fitting. However, compared to other tree models like CART and conditional inference tree, random forest models use bootstrap aggregation technique with random features to do prediction and afterwards uses weighted averaging which lowers the variance of predictions compared to classical decision tree models. The algorithm could be summarized by the following pseudo code of Breiman (2001)[3] :

1) For a training set of length N, sample N instances at random with replacement.
2) Grow a tree on the bootstrap training set using a specified number of random features.
3) Repeat step 1 and 2 for the set number of estimators.
4) Average predictions or take majority vote (depending on type of task).

2.2 Variable importance

Variable importance calculations are based the link that a predictor has with the response variable. Due to the nature of random forest algorithms, it's value can depend on certain hyperparameters. The essential ones are discussed here.

Sensitivity to n and p All learners are sensitive to the nature of the data and it's quality. Two important components to consider when fitting a model is the number of observations n and the number of explanatory variables p. Genuer et al. (2010)[4] ran an experiment using simulated data to understand how the variable importance index would vary when $n << p$. It was shown that as n decreases and becomes smaller compared to p, the variable importance index of significant variables would decrease and get closer to 0. The variability of the index also increases greatly and that is due to the intrinsic algorithm of random forests.

Sensitivity to mtry and ntree Mtry hyper parameter in the random forest algorithm is defined as the subset of covariates in the data that is used to perform the best split at different nodes. According to Genuer et al. (2010)[4], the increase of the number of variables in the subset leads to an increase in the magnitude of variable importance index of useful predictors. On the other hand, ntree, an other hyper parameter of the random forest algorithm, is defined as the number of trees. It's increase leads to a decrease of the standard deviation of the variable importance index. It is important to note, that although the standard deviation decreases, using a large number of trees can prompt over fitting on the training data.

Sensitivity to correlated covariates Many studies were performed to evaluate the impact of having correlated covariates on the variable importance index. Archer and Kimes (2008)[5] paper indicates that the increase in the number of correlated covariates makes it difficult to pick significant variables due to the dilution of the variable importance score which is confirmed by the experiments of Auret and Aldrich (2011)[6].

2.3 Variable selection

In this project, the focus will be put on the "wrapper" methods which base the variable selection on a score that includes the prediction performance. Usually, the performance measure used is the mean squared error for regression tasks and the misclassification rate for classification tasks.

3. Brief review of methods

In the following section, a brief description of the methods used to calculate variable importance and to perform relevant variable selection will be presented.

3.1 Variable importance calculation methods

Classical approach

First we start with variable importance calculation as it is the main index used to perform variable selection using random forests.

The idea behind this measure is that a covariate X is important if the prediction error increases when the relationship between X and Y is modified. To illustrate how to calculate the variable importance one can simply follow these steps for the classical approach which is presented by Breiman (2001)[3].

1) Fit a random forest on the training data.
2) Select an out of bag sample of observations that the model was not trained on (OOB) and calculate the mean squared error on each tree.
3) Select one covariate in the covariate space and permute the values of each instances in order to break the link between the response variable and the covariate selected.
4) Make new predictions using these permuted values and obtain a new mean squared error for OOB sample.
5) Perform the steps 3 and 4 for each covariate in the covariate space.
6) To calculate the variable importance, the sum of differences of the MSE between the permuted covariate and the non permuted covariate is taken and divided by the total number of OOB.

The following formula displays how to calculate variable importance :

$$VIMP(X_j) = \frac{1}{B}\sum_{b=1}^{B}[MSE(\hat{f}_b, OOB_b^j) - MSE(\hat{f}_b, OOB_b)]$$

where j is the covariate for which the value are permuted. b is the b_{th} OOB sample. B is the number of out of bag samples which is equal to the number of trees. $\hat{f}_b$ is the prediction model for the b_{th} sample.

Node assignment variation approach

Ishwaran H. (2007)[7] proposed a different way of breaking the link between Y and X. Instead of permuting the values of X_j they proposed to assign the node at random instead of choosing the best variable to split on. The second option, consists of systematically choosing the opposite split each time a split is performed with that variable. This adds noise to the model and the error is then used to calculate the difference in error with the original model.

Holdout approach

Ishwaran H. (2007)[7] also proposed a second method to estimate the importance of a variable when it comes to predict the response by holding out a group of variables when growing trees and comparing the OOB errors on the different forests when the group is held out vs when it's not.

Altmann Altmann (2010)[8] method is based on an approach that keeps the values and the correlation structure of the covariates intact while computing variable importance under a null hypothesis of no association between predictor and the response variable. Instead of permuting the values of X_j, it permutes the values of the outcome and then multiple random forests are trained and new variable importance are calculated. If the variable importance in the original model without permutation is significantly different than 0, then the variable is kept. Otherwise, it is deemed as non-important and can be discarded from the model. This initial approach is called the permutation approach and is a basic method. Altmann's modifications are implemented with the distribution of the variable importance under the null hypothesis. In other words, in order to allow less training of random forests and to get a conclusive test, it is important to make the right assumptions on the variable importance distribution and the author of the described method allows a parametric approach to estimate the P-values by fitting a specific probability distribution such as a normal, a log-normal etc.

Recurrent relative variable importance This approach is used in case the number of unimportant variables in a data set is large proposed by Szymczak et al (2016)[9]. The selection method is simply based on a ratio of different random forest fits. In other words, first, several random forests are generated based on the data set and parameter values differing only in the seed of the random number generating process. Each random forest is used to compute the variable importance using the classical method and the values obtained are divided by the absolute minimum importance observed in each run. This outputs relative values and each variable having an importance greater or equal to a predetermined factor is kept.

Cross-validated permutation variable importance This method randomly splits the data set into k sets of equal size. The method constructs k random forests, where the l-th forest is constructed based on observations that are not part of the l-th set. For each forest the fold-specific permutation variable importance measure is computed using all observations in the l-th data set: For each tree, the prediction error on the l-th data set is recorded. Then the same is done after permuting the values of each predictor variable.

The differences between the two prediction errors are then averaged over all trees. The cross-validated permutation variable importance is the average of all k-fold-specific permutation variable importances. For classification the mean decrease in accuracy over all classes is used and for regression the mean decrease in MSE. This method is proposed by Janitza et al.(2015)[10].

3.2 Variable selection methods

Variable selection in random forest is mainly based on the variable importance index. The following section explores and briefly describes the different ways to perform variable selection in order to get a better prediction performance.

Recursive feature elimination (RFE) and non recursive feature elimination (NRFE) RFE's objective is to find the smallest number of variables that help to get the best predictive model. It follows the following steps :

1. Train a random forest.
2. Compute the permutation importance measure.
3. Eliminate the less relevant variable(s).
4. Repeat steps 1 to 3 until no further variables remain.

This method is an improved version of the NRFE, which was proven to be less efficient on correlated data. The update of the ranking based on variable importance has proven to be more effective in presence of correlation withing the covariates. For that reason, NRFE will not be used in the following tutorials (Gregorutti et al (2017)[11]).

Boruta This method was developed to find all relevant variables within a classification framework. Kursa and Rudnicki (2010)[12] described the Boruta algorithm with the following steps:

The Boruta algorithm consists of the following steps: 1. Extend the training data by adding copies of all variables (the data is always extended by at least 5 duplicate attributes, even if the number of attributes in the original set is lower than 5).

2. Shuffle the added attributes to remove their correlations with the response.The goal here is to break the link between Y and X.
3. Run a random forest classifier on the extended data set and gather the Z scores computed. The Z score is calculated based on the loss of accuracy in classification task.
4. Find the maximum Z score among duplicate attributes, and then assign a hit to every attribute that scored better than the maximum Z score.
5. For each attribute with undetermined importance perform a two-sided test of equality with the maximum z score.
6. Deem the attributes which have importance significantly lower than the maximum z score as 'unimportant' and permanently remove them from the data set.
7. Deem the attributes which have importance significantly higher than the maximum z score as 'important'.
8. Remove all duplicate attributes.
9. Repeat the procedure until the importance is assigned for all the attributes, or the algorithm has reached the previously set limit of the random forest runs.

Vsurf The Vsurf method is both useful in regression and classification tasks. It is also used for interpretation and prediction. The algorithm in question is performed in two different steps and it is described as follows (Genuer et al. (2015)[13]).

Step 1 : Preliminary ranking and elimination.

a) This step consists of ranking the variables using the classical variable importance index. Genuer et al.(2015)[13] suggests using a typical 50 trees to estimate the variable importance index.

b) Based on a certain threshold, eliminate all variables for which the variable importance index is below the threshold. The threshold in question is estimated by calculating the standard deviation of variable importance. Other strategies could be used to find the best threshold.

Step 2 : Variable selection.

a) For interpretation. The algorithm uses a very simple method to select the most performing model. For $k = 1$ to m, a Random Forest model is built using m important variables following the ranking previously estimated where 1 is the most important variable. Using a different number of runs, typically 25, select the variables that lead to the smallest OOB error.

b) For prediction. In order to improve the prediction performance, start with the ordered sequence of variables selected in the interpretation step. From there, build different random forests by following the order of variable importance from the previous step and use a step wise algorithm to eliminate variables that increase the OOB prediction error by a previously selected threshold.

Jefferie S. Evans et al.(2015)[14] method for model selection (Parsimony) The purpose of the authors of the following method was to simplify as much as possible the complexity of the models. They used the permuted variable importance measure to do so. The method starts by fitting a random forest with all variables and estimates variable importance. The values are then ranked and standardized to a certain ratio. Then, it alliteratively subsets variables within a given ratio and fitting a model for each subset. The resulting model is compared to the original model which is kept constant. The performance is calculated on the OOB error which can include a penalty for the number of parameters in order to reduce the complexity of the model.

4. Review of R resources

The following section will provide a detailed overview of R resources implemented to perform algorithms that helps to estimate variable importance in random forests and to make variable selection.

4.1 Variable importance packages

To calculate variable importance, two main packages are picked for comparison.

The first one is the **randomForest** initially developed in Fortran by Leo Breiman and Adele Cutler and later on ported to R by Andy Liaw and Matthew Wiener. Two functions are of interest in this resource for variable importance. First, the **importance()** function and second, **varImpPlot()**.

4.1.1 importance() function :

The purpose of this function is to calculate the variable importance values for each variable following the classical method presented earlier. The user can customize the importance ranking estimation by modifying the type of calculation made when performing permutation. The first type uses the decrease in accuracy and the second type uses the mean decrease in node impurity.

A very important detail for this function is to make sure that when fitting the randomForest model, specify in the argument of **randomForest()** , that importance=TRUE in order to take full advantage of the proposed arguments of the importance function itself.

Function call : importance(x, type=NULL, class=NULL, scale=TRUE)

Arguments

x : The fitted randomForest object.

type : Can take two values, 1 if the importance measure is calculated with the mean decrease in accuracy and 2 if it is calculated with the mean decrease in node impurity.

class : For classification tasks, this argument allows to calculate the variable importance for the specific class selected in the argument.

scale : When this argument is equal to true, the variable importance measure is divided by the standard deviation of the variable importance vector.

4.1.2 varImpPlot() function

Using the previous calculated importance values, this function provides a visual representation of the variable importance. With the default settings, it displays two plots based on the two types of importance measures (Type= 1 : mean decrease in accuracy or Type = 2 Node impurity).

Function call :

varImpPlot(x, sort=TRUE, n.var=min(30, nrow(x$importance)), type=NULL, class=NULL, scale=TRUE, main=deparse(substitute(x)))

Arguments

x : The fitted randomForest object.

sort : Sorting the values in decreasing order if True.

n.var : User defined number of variables to show.

type : Can take two values, 1 if the importance measure is calculated with the mean decrease in accuracy and 2 if it is calculated with the mean decrease in node impurity.

class : For classification tasks, this argument allows to calculate the variable importance for the specific class selected in the argument.

scale : When this argument is equal to true, the variable importance measure is divided by the standard deviation of the variable importance vector.

main : User defined plot title.

The second package allowing to calculate variable importance is titled **RandomForestSRC**. The authors justified this package by implementing survival forests and also improving computational performance on certain functions. The main contributors are Ishwaran and Kogalur. The relevant functions for computing importance are **vimp.rfsrc()** and **holdout.vimp()**.

4.1.3 vimp.rfsrc()

Function call :

vimp.rfsrc(object, xvar.names, m.target = NULL, importance = c(“permute”, “random”, “anti”), block.size = 10, joint = FALSE, seed = NULL, do.trace = FALSE)

Arguments

object : The fitted randomForestSRC object.

xvar.names : User defined list of variable for which importance is calculated.

m.target : Label of the response variable for which the importance permutation will be calculated on. If nothing is specified, the code will select the default target variable fit on the random forest object.

importance : Three different types of importance values that can be calculated. If the user selects “permute”, the classical approach to calculating variable importance will be used. This approach was explained in the previous sections. If the user specifies “random”, to break the link between the target variable and the covariate in question, the nodes will be selected at random and , finally, the last option is to select “anti”. For this last approach, the covariate is split by sending systematically the instances in the opposite node.

block.size : If block.size can be set from 1 to ntree. When set to ntree, comparison of OOB error will be made on the ensemble forest. On the other hand, if block.size is set to 1, tho OOB error is calculated by tree. Smaller values lead to better accuracy, but take more computation time. The user can make a compromise by selecting different block sizes.

joint : Is a useful argument when the user defines a set of covariates in xvar.names for which they would like to find the joint importance values when the links between the response and the covariates is perturbed as a group.

seed : Specifying the random number generator seed.

do.trace : To print the estimated time of completion.

4.1.4 holdout.vimp()

Function call :

holdout.vimp(formula, data, ntree = function(p, vtry){1000 * p / vtry}, nsplit = 10, ntime = 50, sampsize = function(x){x * .632}, samptype = "swor", block.size = 10, vtry = 1)

formula : Model formula. The same type of formula to use when the user fits a randomforestsrc().

data : A data frame object containing the response and the covariates.

ntree : Number of trees for growing the forest.

nsplit : Number of splits per node, the default value is 10.

ntime : A constraint of ensemble calculations to a grid of ntime value.

sampsize : Size of subsampled data, it could be a specific number or a function of x.

samptype : Type of bootstrap used. "swor" which means sampling without replacement and "swr" means sampling with replacement.

vtry : number of variables to be held-out when growing a tree, typically 1.

block.size : If block.size can be set from 1 to ntree. When set to ntree, comparison of OOB error will be made on the ensemble forest. On the other hand, if block.size is set to 1, though OOB error is calculated by tree. Smaller values lead to better accuracy, but require more computation time. The user can make a compromise by selecting different block sizes.

Other packages are developped to make variable importance estimation but are less popular. **Vita** is one of them and contains two variable importance functions.

The first one is **PIMP** and it is based on Altmann's method.

4.1.5 : PIMP() function

Function call :

PIMP(X, y, rForest, S = 100, parallel = FALSE, ncores=0, seed = 123)

X : A data frame or a matrix of predictors.

y : A response vector. If the values are factor, the function assumes a classification task.

rForest : randomForest must be used with the argument importance set to true.

S : The number of permutation to perform on the response vector.

parallel : if set to true, the computation will be made in parallel.

ncores : How many child processors will be run simultaneously.

seed : seed for random number generator.

The second function implemented in **VITA** is **CVPVI** which is a cross validation approach.

4.1.6 : CVPVI() function

Function call :

CVPVI(X, y, k = 2, mtry= if (!is.null(y) && !is.factor(y)) max(floor(ncol(X)/3), 1) else floor(sqrt(ncol(X))), ntree = 500, nPerm = 1, parallel = FALSE, ncores = 0, seed = 123)

X : A data frame or matrix of covariates.

y : The response vector.

k : The number of folds used for cross validation.

mtry : Number of variables randomly sampled as candidates for each split for I_{th} forest. The default values differ based on the nature of the task. For classification, it is $\sqrt(p)$ and for regression it is $p/3$ where p is the number of covariates in the model.

ntree : Number of trees to grow per forest.

nPerm : Number of times the k_{th} data set are permuted per tree for assessing variable fold-specific permutation variable importance. Default is nPerm=1

parallel : if set to true, the computation will be made in parallel.

ncores : How many child processors will be run simultaneously.

seed : seed for random number generator.

Summary

We presented 6 functions for variables importance and they can all help distinguish wich covariates are the most useful. In the following sections, where the functions are applied to data sets, only **importance** , **vimp** and **holdout.vimp** will be demonstrated since they perform the most commonly used algorithms and have been tested rigorously.

The thre previous functions retained allow to perform calculations to display and indicate which covariates are the most useful in predicting the response variable. The **importance** function in the **randomForest** offers a simple way to get the details and it's usage is very straightforward. Thanks to the function **varImpPlot()** the user can easily display the result of their findings. On the other hand, the **vimp.rfsrc()** function from the **randomforestSRC** package offers much more flexibility and user parametrization. One of the most interesting features is the possibility to select the type of importance measure to compute and the number of trees used to obtain the OOB error comparison. Also, allowing for such customization in the function helps the user to alter the computation speed. For example, the analyst desiring to find out the importance of a group of variables could use the *joint* and specify those variable in the *x.var* argument to accelerate the process. Such flexibility is not offered in the **importance()** function. The last function covered for variable importance is the **holdout.vimp()** from the same package which offers important flexibility as well. The calculation methods are very different from one another and results for the latter function require a large number of trees in order to lower the variance of the importance estimates.

The following table displays the summary of each function presented above.

Function name	**main purpose**	**approach**	**Package**
importance()	variable importance estimate	Breaking the link between X_j and y by permutation of x_j values for each observation in the data set	randomForest
varImpPlot()	Visualization	2 graphs based on the type of error	randomForest
vimp.rfsrc()	variable importance estimate	Three different approaches to estimate variable importance. Classical permutation of X_j, random node assignment and opposite split	randomForestSRC

Function name	main purpose	approach	Package
holdout.vimp()	variable importance estimate	Holding out certain variables from the growing process and comparing the predictions to the ones made without holding them out	randomForestSRC
PIMP()	Variable importance for classification and regression	This function uses the Altmann's method to make variable selection. It permutes the response variable y to break its links with covariates.	VITA
CVPVI()	Variable selection for classification and regression using cross-validation	This method splits the dataset into k sets of equal size and constructs the same number of random forests. In each fold, variable permutation is completed to break the link between Y and X	VITA

4.2 Variable selection

For variable selection the following thre main R packages will be considered.

The first one is the **Boruta** package developed by Kursa and Rudnicki (2010). The main function for variable selection here is **Boruta** which applies the Boruta algorithm mentioned previously for variable selection.

4.2.1 Boruta() function :

Function call :

Boruta(x,y, pValue = 0.01, mcAdj = TRUE, maxRuns = 100, doTrace = 0, holdHistory = TRUE, getImp = getImpRfZ)

x : Data frame of covariates.

y : A vector of the response variable.

pValue : Confidence level to perform the two sided test on the z score of variable importance comparison between the copied covariate and the original covariate valus.

mcAdj : If set to be true, a multiple comparison method using Bonferroni in order to adjust the p-values.

maxRuns : The number of iterations of the algorithm. The algorithm stops when all variables are confirmed or the number of max runs is reached.

doTrace : verbosity level. 0 means no tracing, 1 means reporting decision about each attribute as soon as it is justified, 2 means the same as 1, plus reporting each importance source run, 3 means the same as 2, plus reporting of hits assigned to yet undecided attributes.

holdHistory : if set to true, the full trace of importance calculations is stored and returned in the object ImpHistory.

getImp : Allows to get the variable importance calculated using the **getImpRfz** which gathers Z-scores of mean decreasing accuracy measures.

formula : Alternatively, formula describing model to be analysed.

data : Data set to use for the algorithm.

4.2.2 VSURF() function :

Function call :

VSURF(x, y, ntree = 2000, mtry = max(floor(ncol(x)/3), 1), nfor.thres = 50, nmin = 1, nfor.interp = 25, nsd = 1, nfor.pred = 25, nmj = 1, RFimplem = “randomForest”, parallel = FALSE, ncores = detectCores()-1, clusterType = “PSOCK”, verbose = TRUE)

x or formula : A data frame or a matrix of predictors, the columns represent the variables. Or a formula describing the model to be fitted

y : A vector of the response variable for each instance. It has to be a factor for classification and numeric for regression.

mtry : A subsample of covariates selected randomly as candidates at each split. It is the standard parameter of random forests.

nfor.thres : The number of forests grown for the first step of variable elimination.

nmin : The coefficient by wich the minimal importance score is multiplied to set a threshold.

nfor.interp :The number of forests grown for the second step of variable elimination.

nsd : Number of times the standard deviation of the minimum value of err.interp is multiplied. err.interp is obtained by adding the min error of step 1 plus the coefficient nsd selected by this argument times the standard deviation of the out of bag error at the second step of the algorithm explained above.

nfor.pred : The number of forests grown for the third step of variable elimination.

nmj : Number of times the mean.jump is multipied. Which is calculated by taking the difference between OOB errors of one model and its first following model.

RFimplem : Choice of the package to use to fit the random forest. **randomForest** is set as the default option, but **ranger** and **Rborist**.

parallel : If set to true, VSURF function will run on multuple cores or in parallel.

ncores : Number of cores to use, the default value is set to the number of cores detected by R minus one.

clusterType : Type of the multiple cores cluster used to run the function.

verbose : If set to true, tracking of the progress will be printed in the console.

data : The data frame containing the variables in the model.

na.action : A function to specify the action to be taken if NAs are found. (NOTE: If given, this argument must be named, and as randomForest it is only used with the formula-type call.)

4.2.3 varSelRF() function :

Function call :

varSelRF(xdata, Class, c.sd = 1, mtryFactor = 1, ntree = 5000, ntreeIterat = 2000, vars.drop.num = NULL, vars.drop.frac = 0.2, whole.range = TRUE, recompute.var.imp = FALSE, verbose = FALSE, returnFirstForest = TRUE, fitted.rf = NULL, keep.forest = FALSE)

xdata : A data frame of a matrix of covariates. Missing values are not allowed.

Class : The response variable must.

c.sd : In the same idea of the 1SE rule, this factor multiplies the standard deviation error and ads it to the minimal error obtained to select the best model for variable selection.

mtryFactor : The factor that multiplies the $\sqrt(p)$ to select the number of covariates to be candidates for each split in the **randomForest()** function.

ntree : The number of tree for the initial forest.

ntreeIterat : The number of tree to use for all additional forests.

vars.drop.num : The number of variable to drop at each iteration.

vars.drop.frac : Instead of a number of variables to drop, this argument allows to specify a proportion of variables to drop.

whole.range : If set to true, the algorithm will keep dropping variables until only two are left and the model selects the best amongst all possible options. If false, the stopping criterion of c.sd error is respected.

recompute.var.imp : If set to true, the variable importance values will be recomputed at each iteration.

verbose : If set to true, tracking of the progress will be printed in the console.

returnFirstForest : If set to true, the first random forest fitted will be returned.

fitted.rf : A previously fitted random forest. (optional)

keep.forest : Save the random forest.

Summary

The following table displays the summary of each function presented above.

Function name	**main purpose**	**approach**	**Package**
Boruta()	Variable selection for classification	This function follows the Boruta method which , in summary, consists of a top-down search for relevent features by comparing original attributes importance with importance achieved at random	Boruta
VSURF()	Variable selection for classification and regression with a step for interpretation	Three steps selection algorithm. The first steo eliminates irrelevant variables from the dataset. The second allows to keep the ones for interpretation and the last step is used to improve prediction performance.	VSURF
varSelRF()	Variable selection for classification	Using the OOB error as minimization criterion, carry out variable elimination from random forest, by successively eliminating the least important variables (with importance as returned from random forest)	varSelRF

5. Guide on how to apply the previous methods and resources for a regression and a classification task.

The previous sections allowed to detail the available resources and methods used to compute and select the most important variables when it comes to fitting a random forest model. The next steps will provide an in-depth tutorial of these methods with concrete examples using simulated and real data.

5.1 Data sets overview

Two data sets will be explored and analysed in the following sections. First, for a classification task, the BreasCancer data will be used and second, for a regression task, the BostonHousing data will be used.

5.1.1 BreastCancer data set description

The first data set used is for a classification task and it is found in the mlbench package. The full description of the dataset can be found in this link **BreastCancer** DATA.

To load the data set, first install and import the **mlbench** package. If these basic R steps are not familiar, the following web demonstration is an excellent guide for beginners in R studio. **Datacamp**

The following block of code allows to install and load the dataset.

```
#Loading the already installed libray containing the dataset.
library("mlbench")

# Loading the dataset from mlbench package.
data("BreastCancer")

# removing instances with missing values
BreastCancer <- na.omit(BreastCancer)

# Summary of breast cancer type in the dataset
summary(BreastCancer$Class)
```

```
##    benign malignant
##       444       239
```

The data set contains 11 variables amongst which “class” is a binary variable defined by benign cancer and malignant cancer. Using the 10 covariates, the goal is to predict, for new instances, in which class they belong. There are 683 observations.

5.1.2 Bostonhousing data set description

The following block of code allows to load it.

```
data("BostonHousing2")

#The dataset description indicates that medv is incorret and cmedv should be used.
BostonHousing2$medv <- NULL

#Some variables represent the exact information. Tract , town , lon and lat.
#They all refer to the geographical location of the how in question.
```

```r
#Only lon and lat will be kept as they are numerical values. The others are
#character values.
BostonHousing2$town <- NULL
BostonHousing2$tract <- NULL
```

The second data set is also found in the **mlbench** package. This set is going to be used for a regression task. The goal is to adequately predict the median value of owner-occupied homes in 1970 around the Boston area. The dataset contains 15 covariates and 506 observations.

5.1.3 Train and test set splitting

In order to perform the required analysis to compare model performance, it is important to split the data between training and test sets. 20% of the data sets will be used to estimate generalization error.

```r
#classification data
set.seed(12345)
index = sort(sample(nrow(BreastCancer), nrow(BreastCancer)*.8))
train_class = BreastCancer[index , ]
train_class$Id <- NULL
test_class = BreastCancer[-index , ]
test_class$Id <- NULL

#Regression data
set.seed(12345)
index = sort(sample(nrow(BostonHousing2), nrow(BostonHousing2)*.8))
train_reg = BostonHousing2[index , ]
test_reg = BostonHousing2[-index , ]
```

Also, to make a better assessment of the ability of these methods to detect unimportant variables, some noise variables will be added.

```r
#Generating noise variables
num_noise_var = 5
set.seed(1234)
for ( i in 1: num_noise_var) {
  train_class[,10+i] <- rnorm(nrow(train_class) , mean = i , i)
  test_class[,10+i] <- rnorm(nrow(test_class) , mean = i , i)
  train_reg[,16+i] <- rnorm(nrow(train_reg) , mean = i , i)
  test_reg[,16+i] <- rnorm(nrow(test_reg) , mean = i , i)
}
```

5.2 Fitting a regular random forest without variable selection

To estimate variable importance, some methods require fitting a random forest before passing the data into the specific function allowing to do so. This initial section is dedicated to a simple random forest model without variable selection. It will help validate whether the variable selection approach is improving the quality of the predictions or if the variable selection methods decrease the overall prediction accuracy. In other words, this helps setting a benchmark for comparison. For the scope of the project, hyperparameters will be fixed from the start in order to compare the different methods with everything else being equal except, the intrinsic algorithm used to perform variable selection.

The first model to be fit is on the BreastCancer data and the good classification rate will be reported.

```r
#First load the package
library("randomForest")

#Fitting the randomForest, note that importance is set as true for future reference.
rf_class <- randomForest(Class ~ . , data = train_class , importance = TRUE )

#Predicting on the test set
pred_bench_class <- predict(rf_class , newdata = test_class , type = "response")

#good classification rate
Bench_class <- length(which(pred_bench_class == test_class$Class )) / nrow(test_class)

#Confusion matrix below present the results of the classification task
c_mat_1 <- table(pred_bench_class , test_class$Class)
```

The second model to fit is on the BostonHousing and the mean squared error with the mean absolute error will be reported.

```r
#Fitting the random forest
rf_reg <- randomForest(cmedv~. , data=train_reg , importance = TRUE)

pred_bench_reg <- predict(rf_reg , newdata = test_reg , type="response" )

MSE_bench <- mean((pred_bench_reg - test_reg$cmedv)^2)
MAE_bench <- mean(abs(pred_bench_reg - test_reg$cmedv))
```

The good classification rate obtained using the default parameters of the random forest is 0.9708.

On the other hand, the mean squared error for the regression task is 17.26 and the mean absolute error is 2.476.

5.3 Estimation of variable importance

Once a model is fit, a user could be interested in determining which variable was the most influent in predicting the response. The following section demonstrates how to estimate variable importance in both the classification task and the regression task.

5.3.1 Estimating importance function from randomForest package

The required package is already installed and loaded from the previous code chunk used to fit the different forests. First, the importance function will be used to estimate variable importance and second, it will be ploted to help visualize the results.

Before computing importance, it is crucial to indicate that to fully take advantage of the functions presented to estimate these values, the user has to select the *importance* argument in the randomForest function as true.

```r
# Computing using mean decrease in accuracy
imp_class_1 <- importance(rf_class , type = 1 , scale = TRUE )

# Computing using mean decrease in node impurity
imp_class_2 <- importance(rf_class , type = 2 , scale = TRUE)
```

```r
# Visualing the results using the plot function most important.
par(cex.main = 0.8 , cex.axis = 0.8 , cex.lab = 0.8 , cex.main = 0.8)
varImpPlot(rf_class , sort = TRUE , n.var = 5 ,
           main = "Variable importance : classification task" ,
           scale = TRUE)
```

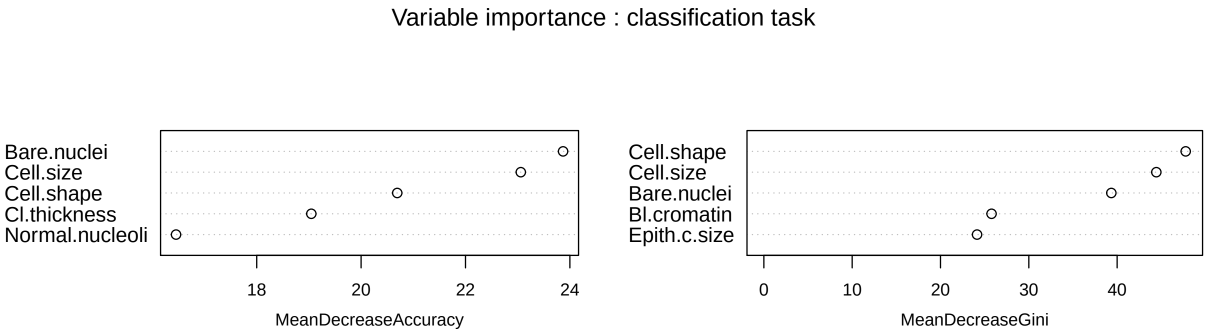


Figure 1: The variable importance calculated here are both from the importance function.

This previous figure displays the top five important variables based on the error decrease selected. On the left graph, type = 1 is chosen. The graph on the right, on the other hand uses type=2. Both methods selected the same four variables except for cell thickness which was only selected by the mean decrease in accuracy and normal nucleoli, which was only selected by the mean decrease in gini index. Although the method use the same algorithm, the simple modification in the type of error calculated leads to different results.

For the regression task, the same steps can followed in order to obtain the needed measure.

```r
# Computing variable importance for regression task : increase in MSE
imp_reg_1 <- importance(rf_reg , type = 1 , scale = TRUE)

# Computing variable importance for regression task : increase in Node purity
imp_reg_2 <- importance(rf_reg , type = 2 , scale = TRUE)

# Plotting the results
par(cex.main = 0.8 , cex.axis = 0.8 , cex.lab = 0.8 , cex.main = 0.8)
varImpPlot( rf_reg , n.var = 5 , main = "Variable importance : regression task" ,
            scale = TRUE)
```

For this task, the two methods agree on the first three variables, but not on the 4th and 5th.

This is the implemented function offered by randomForest which can be very convenient, but it only uses randomForest objects and is not very flexible. For that reason, it will no longer be used in he next sections.

5.3.2 Estimating importance measure from randomForestSRC package

This package allows much more flexibility and has more parametrization when it comes to computing variable importance. Before proceeding, a forest will be trained just like in the previous section.

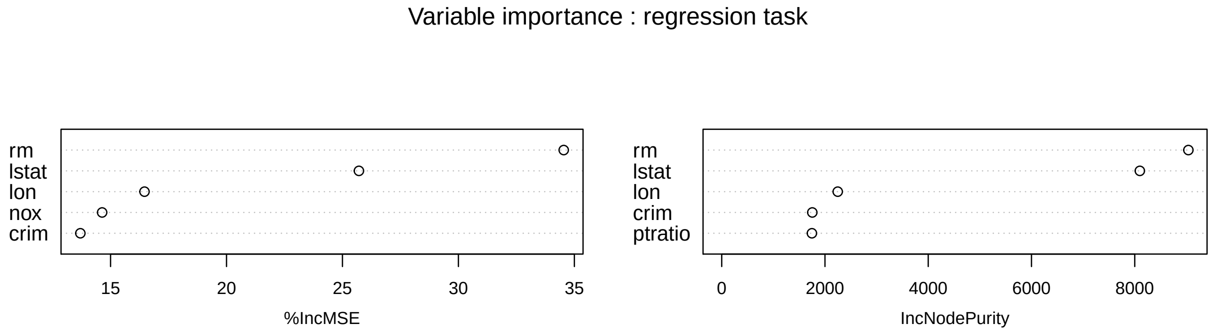


Figure 2: Like the previous figure, this displays the results from the VarImpPlot function.

```r
# First , loading the package
library("randomForestSRC")

#Classification
# Fitting a tree using the default parameters
set.seed(1234)
rfsrc_class <- rfsrc( Class ~ . , data = train_class , ntree = 500 )

#Predictions
pred_bench_src_class <- predict(rfsrc_class , newdata =  test_class)

#The output when using randomForest src is a list of objects. It only outputs probs.
#The following code transforms the output to the same format as the test data.
pred_bench_src_class <- ifelse(pred_bench_src_class$predicted[,1] > 0.5 ,
                               "benign" , "malignant")
pred_bench_src_class <- as.factor(pred_bench_class)

Bench_rfsrc_class <- length(which(pred_bench_src_class == test_class$Class )) /+
  nrow(test_class)
```

The performance of the model using randomForestSRC is very close to the older version : 0.9708

```r
# Fitting the regression model
set.seed(1234)
rfsrc_reg <- rfsrc( cmedv ~ . , data = train_reg , ntree = 500 )

#Making predictions
pred_bench_src_reg <- predict(rfsrc_reg , newdata = test_reg  )

#Evaluating performance
MSE_bench_src <- mean((pred_bench_src_reg$predicted - test_reg$cmedv)^2)
MAE_bench_src <- mean(abs(pred_bench_src_reg$predicted - test_reg$cmedv))
```

The performance of the models using randomForestSRC for the regression task is also very competitive and close to the older version. It's slightly worse. The mean absolute error is 2.58 and the mean squared error is 20.18

The following steps will allow us to estimate variable importance using these different grown forests and compare the different available procedures since the package allows more options. The first formula to explore is the ***vimp.rfsrc()**

```
#The vimp() function allows to select three different types of error computation.
#Classification task first

#Permutation
imp_src_class_1 <- vimp(rfsrc_class , importance = "permute" , block.size = 1 )
#The following code gets the overall importance of each variable instead
#of class specific
imp_src_class_1 <- sort(imp_src_class_1$importance[,1])

#Random
imp_src_class_2 <- vimp(rfsrc_class , importance = "random" , block.size = 1 )
imp_src_class_2 <- sort(imp_src_class_2$importance[,1])

#Opposite split
imp_src_class_3 <- vimp(rfsrc_class , importance = "anti" , block.size = 1 )
imp_src_class_3 <- sort(imp_src_class_3$importance[,1])

#Plot to compare the importance per method, NEEDS A LITTLE ATTENTION
par(mfrow = c(1,3) , cex.axis = 0.8 , cex.lab = 0.8 , cex.main = 0.8)
dotchart(imp_src_class_1[10:14] , main = "Permutation" , xlim = c(0 , 0.2))
dotchart(imp_src_class_2[10:14] , main = "Random" , xlim = c(0 , 0.2))
dotchart(imp_src_class_3[10:14] , main = "Opposite split" , xlim = c(0 , 0.2))
```

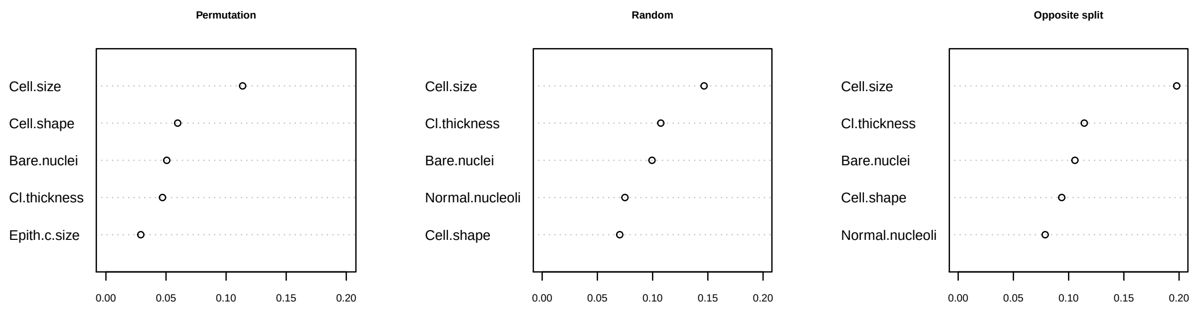


Figure 3: The methods for variable importance calculation vary from function to function. This one uses 3 types of different approaches to break the link with the response variable. The results are diplayed for all three, permute, random and anti.

For this dataset, the **vimp()** function seem to have a very similar result accross all methods used within the function to evaluate variable importance. The most important variable is the cell size according to all methods.

Now, for the regression task, the following code is applied, which is very similar to the previous one.

```
# Permutation on the regression data
imp_src_reg_1 <- vimp(rfsrc_reg , importance = "permute" , block.size = 1  )
imp_src_reg_1 <- sort(imp_src_reg_1$importance)
# Random
imp_src_reg_2 <- vimp(rfsrc_reg , importance = "random" , block.size = 1  )
```

```
imp_src_reg_2 <- sort(imp_src_reg_2$importance)
#opposite
imp_src_reg_3 <- vimp(rfsrc_reg , importance = "anti" , block.size = 1  )
imp_src_reg_3 <- sort(imp_src_reg_3$importance)

#Ploting
par(mfrow = c(1,3) , cex.axis = 0.8 , cex.lab = 0.8 , cex.main = 0.95)
dotchart(imp_src_reg_1[16:20] , main = "Permutation" , xlim = c(0 , 120))
dotchart(imp_src_reg_2[16:20] , main = "Random" , xlim = c(0 , 120))
dotchart(imp_src_reg_3[16:20] , main = "Opposite split" , xlim = c(0 , 120))
```

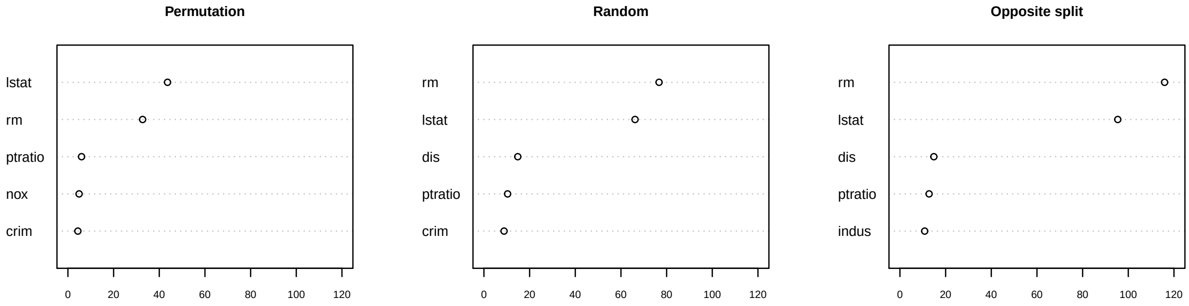
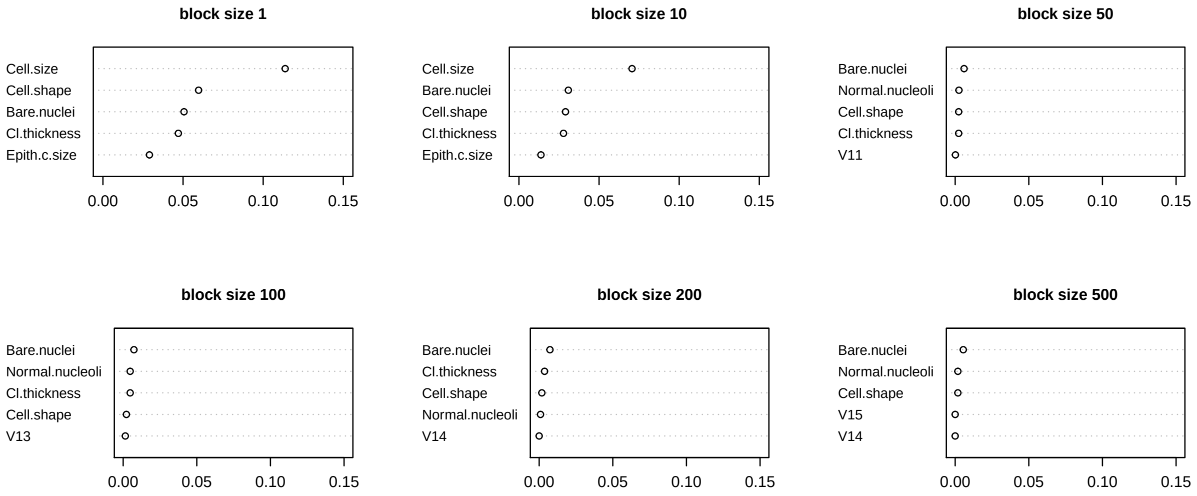


Figure 4: This figure diplays the results of importance calculation for the regression task

Similar to the results obtained using the classification dataset, the three methods lead to similar conclusion. The most important variables from all type of calculations are concording and there are no significant differences. An important modification that could be analysed is the number of tree in the block.size argument. This parameter is explained in the function definition in the previous section. The following chunk of code tests out tha variable importance values using different values of block.size.

```
#Block size varying [1 , 10 , 50 , 100 , 500]
block_size = c(1 , 10 , 50 , 100 , 200 , 500)
imp_block_size = list()
for (i in block_size){
  imp_block_size[[i]] = vimp(rfsrc_class , importance = "permute" , block.size = i)
  imp_block_size[[i]] = sort(imp_block_size[[i]]$importance[,1])
}

par(mfrow = c(2 , 3) , cex.axis = 1.1 , cex.lab = 0.8 , cex.main = 1.1)
# We need to find a way to plot them together
for( i in block_size) {
  dotchart(imp_block_size[[i]][10:14] ,
           main = paste("block size" , as.character(i)) , xlim = c(0 , 0.15))
}
```



The advantage of using a block size larger than one is to save on computing time when the number of observations and the number of variables is very large. In case the model is very complex, this option can lead to more efficient ways of estimating variable importance. But, as the block size increases, we can see a decrease in the method to evaluate the importance of each variable. This is due to the fact that , when the block size is equal to the total number of trees, the out of back error on permutation is compared to the whole forest instead of it's corresponding tree.

An other way to compute variable importance using randomForestSRC is by using the holdout method. It is possible due to it's implementation in the function **holdout.vimp()**. This function allows to holdout certain variables from growing trees in the random forest algorithm. The typical number of variables to holdout is 1.

```
# Computing variable importance for the classification task
imp_hold_class <- holdout.vimp( Class ~. , data= train_class , ntree = 500 ,
                                block.size = 1 , ntime = 100)

imp_hold_class <- sort(imp_hold_class$importance[,1])

# Computing cariable importance for the regression task
imp_hold_reg <- holdout.vimp(cmedv ~. , data= train_reg , ntree = 500 ,
                             block.size = 1 , ntime = 100)

imp_hold_reg <- sort(imp_hold_reg$importance)

# Plot of the results
par(mfrow = c(1,2) , cex.axis = 0.8 , cex.lab = 0.3 , cex.main = 0.75)
dotchart(imp_hold_class[10:14] , labels = rownames(imp_hold_class) ,
         main = "Classifcation task" , xlim = c(min(imp_hold_class[10:14]) - 0.001 ,
                                       max(imp_hold_class[10:14] + 0.001) ) )
dotchart(imp_hold_reg[16:20] , labels=rownames(imp_hold_reg) ,
         main = "Regression task" , xlim = c(min(imp_hold_reg[16:20])-0.5 ,
                                    max(imp_hold_reg[16:20])+ 0.5))
```

Using the holdout function, some new variables in the classifcation data set are shown as important compared to the previous method. Mitoses, for example, was not part of the most important ones. In order to easily compare the agreement between all the methods, a rank plot between diffrent pairs will help clarify.

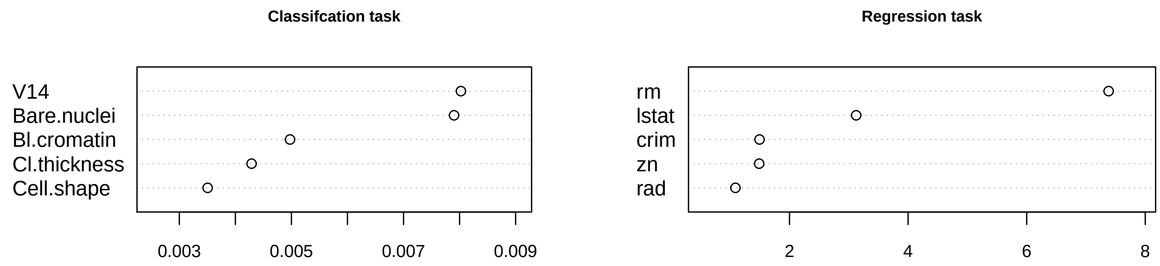


Figure 5: The hold out approach leaves some variables out from the process of growing trees. This method leads to different results compared to the previous two. The number of variables heldout here is 1 per iteration

This rank plot will only be completed on the classification task, but the exact same logic can be applied to the regression task as well. In addition, the importance measures used for this comparison will be the classical method using **importance** function from the **randomForest** package , the **vimp** function using the permutation method with block size set to 1 and the holdout method using the **holdout.vimp** from the randomForestSRC package. The analysis is restricted to those three for the sake of the tutorial, but further study could completed if necessary by varying different parameters of the functions.

```r
#Create the ranking
# 9 variables in the classification task
rank <- seq(1 , length(imp_class_1) , 1)

# sort the vectors that are not
covariates <- rownames(imp_class_1)
data_1 <- as.data.frame(cbind(covariates , imp_class_1))
rownames(data_1) <- seq(1 , nrow(data_1) , 1)
data_1$MeanDecreaseAccuracy <- as.numeric(as.character(data_1$MeanDecreaseAccuracy ))
data_1 <- data_1[order(data_1$MeanDecreaseAccuracy , decreasing = TRUE) , ]
data_1$classic_rank <- rank
data_1$MeanDecreaseAccuracy <- NULL

# Vimp function
imp_src_class_1 <- sort(imp_src_class_1 , decreasing = TRUE)
data_2 <-cbind(imp_src_class_1 , rank)
covariates_2 <- rownames(data_2)
data_2 <- as.data.frame(data_2 )
data_2$covariates <- covariates_2
data_2$imp_src_class_1 <- NULL
colnames(data_2) <- c("Vimp_rank" , "covariates")

# holdout function
imp_hold_class <- sort(imp_hold_class , decreasing = TRUE)
data_3 <-cbind(imp_hold_class , rank)
covariates_3 <- rownames(data_3)
data_3 <- as.data.frame(data_3)
data_3$covariates <- covariates_3
data_3$imp_hold_class <- NULL
colnames(data_3) <- c("Hold_rank" , "covariates")
```

```
#Merging
data_m <- merge(data_1 , data_2 , by="covariates")
data_m <- merge(data_m , data_3 , by="covariates")

# plot hold-classic , hold-vimp , vimp-classic
par(mfrow = c(1 , 3) , cex.main=0.8  , cex.axis = 1 , cex.lab = 1.1)
plot(data_m$Hold_rank , data_m$classic_rank ,
     xlab = "Holdout ranking" , ylab = "classic ranking")
axis(2, at = seq(1, 9, by = 1), las=0) ; axis(1, at = seq(1, 9, by = 1), las=0)
plot(data_m$Hold_rank , data_m$Vimp_rank ,
     xlab = "Holdout ranking" , ylab="Vimp ranking")
axis(2, at = seq(1, 9, by = 1), las=0) ; axis(1, at = seq(1, 9, by = 1), las=0)
plot(data_m$Vimp_rank , data_m$classic_rank,
     xlab ="Vimp ranking" , ylab="classic ranking")
axis(2, at = seq(1, 9, by = 1), las=0) ; axis(1, at = seq(1, 9, by = 1), las=0)
mtext("Ranking comparison of variable importance" , outer = TRUE , line = -1.5)
```

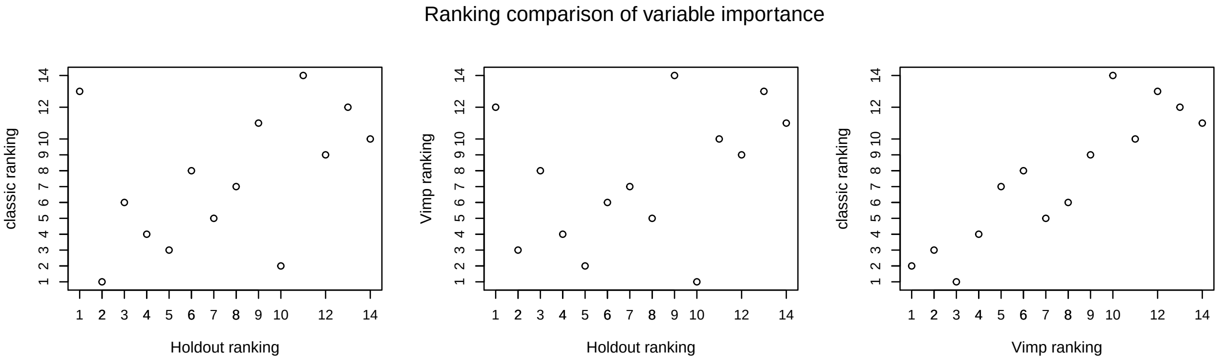


Figure 6: This figure ranks pairwise the importance using the thee methods to identify their agreement. If the dots form a positive linear trend, then the methods agree.

This pairwise comparison shows that the **holdout.vimp** function estimates importance very differently compared to the previous two methids since the rankings are not similar. On the other hand, the classic methond and the regular vimp function show a very close ranking. It can be seen on the right plot.

5.3.3 Summary

The previous demonstrations of variable importance calculations have indicated that the methods tend to agree with the most important variables and their ability to detect which ones have an impact on the response variable depends on the parameters used. For example, when changing the error type, the ranking changes as well when using the variable importance function. Modifiying how the link with the response variable is broken when using the vimp function also modifies the importance measures. In general, the two functions, **vimp** and **importance** provide similar conclusions whereas the **holdout.vimp** method does not agree with the results of the first two. Finally, when using the block.size argument on the vimp function, as the value for this parameter increases, the method's ability to distinguish important variables decreases. It can be very useful to reduce computing time when the number of instances and the number of variables is large, but in general context, there is a tradeoff in terms of quality.

To conclude this section, computing variable importance can be used for interpretation and get a rough idea of which variable impacts the response variable the most. For prediction purposes, this step is completed in order to discriminate variables from model based on different factors, generaly the ranking of variable importance. The following section is a tutorial on implementing methods that use variable importance to select which variables to keep in the model to perform predictions.

5.4 Variable selection

To evaluate wether the variable selection method is useful to improve the prediction capabilities of a model, a comparison with the bench mark set will be provided at the end.

```
# Loading the packages
library("Boruta")
library("VSURF")
library("varSelRF")
library("vita")
```

5.4.1 Boruta

```
# Using Boruta to make variable selection
Boruta_class <- Boruta(Class ~., data = train_class)

Boruta_reg <- Boruta(cmedv ~.   , data = train_reg)

# The following variables are the onese Boruta's method deemed important
print("Boruta's selection for classification task")
```

```
## [1] "Boruta's selection for classification task"
```

```
Boruta_class$finalDecision
```

```
##    Cl.thickness       Cell.size      Cell.shape   Marg.adhesion    Epith.c.size
##       Confirmed       Confirmed       Confirmed       Confirmed       Confirmed
##     Bare.nuclei     Bl.cromatin Normal.nucleoli         Mitoses             V11
##       Confirmed       Confirmed       Confirmed       Confirmed        Rejected
##             V12             V13             V14             V15
##        Rejected        Rejected        Rejected        Rejected
## Levels: Tentative Confirmed Rejected
```

```
print("Boruta's selection for regression task")
```

```
## [1] "Boruta's selection for regression task"
```

```
Boruta_reg$finalDecision
```

```
##       lon       lat      crim        zn     indus      chas       nox        rm
## Confirmed Confirmed Confirmed Confirmed Confirmed  Rejected Confirmed Confirmed
##       age       dis       rad       tax   ptratio         b     lstat       V17
## Confirmed Confirmed Confirmed Confirmed Confirmed Confirmed Confirmed  Rejected
##       V18       V19       V20       V21
##  Rejected  Rejected  Rejected  Rejected
## Levels: Tentative Confirmed Rejected
```

Boruta's method was able to discriminate all noise variables added to the dataset and only selected the ones already in it. In the classification dataset, no original variabel was removed. In the regression data set, only the chas variable was removed.

Now, random forests will be grown with the selected variables for both tasks.

```
#Fitting the model without rejected variables
# Classification and regression
boruta_rf_class <- randomForest(Class~. - V11 - V12 -V13 -V14 -V15 ,
                                data=train_class)

boruta_rf_reg <- randomForest(cmedv~.  -chas -V17 -V18 -V19 -V20 -V21 ,
                              data=train_reg)

#Prediction and performance measure
pred_boruta_class <- predict(boruta_rf_class , newdata = test_class , type = "response")

pred_boruta_reg <- predict(boruta_rf_reg , newdata = test_reg , type="response")

#good classification rate
boruta_rate_class <- length(which(pred_bench_class == test_class$Class )) /+
  nrow(test_class)

#Confusion matrix below present the results of the classification task
c_mat_2 <- table(pred_boruta_class , test_class$Class)

#MSE and MAE
MSE_Boruta <- mean(( pred_boruta_reg - test_reg$cmedv)^2)
MAE_Boruta <- mean(abs(pred_boruta_reg - test_reg$cmedv))
```

5.4.2 VSURF

The second function to demonstrate is from the **VSURF** package. This method uses three steps to make variable selection.

```
# using the default parameters to perform variable selection
vsurf_class <- VSURF(Class ~. , data=train_class , parallel=TRUE ,
                     ncores=4 , verbose=FALSE)
```

After performing the algorithm, displaying the list of variables selected at each step can be done by the following code.

First step : Threshold

```
# On the first step the following variables were removed
colnames(train_class[,vsurf_class$varselect.thres])
```

```
##  [1] "Cell.size"       "Bare.nuclei"     "Cell.shape"      "Normal.nucleoli"
##  [5] "Cl.thickness"    "Bl.cromatin"     "Epith.c.size"    "Marg.adhesion"
##  [9] "Mitoses"         "V11"
```

At this initial step, the only noise variable that passed the threshold is V11.

Second step : Interpretation

```
# On the first step the following variables were removed
colnames(train_class[,vsurf_class$varselect.interp])
```

```
## [1] "Cell.size"       "Bare.nuclei"     "Cell.shape"      "Normal.nucleoli"
## [5] "Cl.thickness"    "Bl.cromatin"     "Epith.c.size"
```

With this function, two original variables were deleted.

Third step : Prediction

```
# On the first step the following variables were removed
colnames(train_class[,vsurf_class$varselect.pred])
```

```
## [1] "Cell.size"       "Bare.nuclei"     "Cell.shape"      "Normal.nucleoli"
```

Finally, for predictions. This methods only selects four amongst all the 9 original variables.

For the regression task, the same approach step by step approach can be performed, but since the goal is to compare the prediction performance, only the last step is of interest for that task

```
# using the default parameters to perform variable selection
vsurf_reg <- VSURF(cmedv ~. , data=train_reg , parallel=TRUE , ncores=4 , VERBOSE = FALSE)
```

```
## Thresholding step
## Estimated computational time (on one core): 269 sec.
##
## Interpretation step (on 17 variables)
## Estimated computational time (on one core): between 123.3 sec. and  289 sec.
##
## Prediction step (on 9 variables)
## Maximum estimated computational time (on one core): 123.7 sec.
##   |                                                                      |
```

The selected variables for the prediction task are the following.

```
colnames(train_reg[,vsurf_reg$varselect.pred])
```

```
## [1] "b"    "nox"  "chas" "age"  "rad"
```

The next step is to predict and compute the peformance on the test set.

```
#Fiiting on data
rf_vsurf_class <- randomForest(Class~ Cell.size + Bare.nuclei + Cell.shape +
                                 Normal.nucleoli , data=train_class)

rf_vsurf_reg <- randomForest(cmedv ~ b + nox + chas + zn , data=train_reg )

#Pre on the classification task
#Prediction and performance measure
pred_vsurf_class <- predict(rf_vsurf_class , newdata = test_class , type ="response")
```

```
pred_vsurf_reg <- predict(rf_vsurf_reg , newdata = test_reg , type="response")

#good classification rate
vsurf_rate_class <- length(which(pred_vsurf_class == test_class$Class )) /+
  nrow(test_class)

#Confusion matrix below present the results of the classification task
c_mat_3 <- table(pred_vsurf_class , test_class$Class)

#MSE and MAE
MSE_vsurf <- mean(( pred_vsurf_reg - test_reg$cmedv)^2)
MAE_vsurf <- mean(abs(pred_vsurf_reg  - test_reg$cmedv))
```

Vsurf function had multiple interesting features and arguments to use that deepens the variable selection analysis. Now, the next function used to perform variable selection is called **varSelRF**

5.4.3 VarSelRF

```
#Loading the library
library("varSelRF")
```

The following block will demonstrate step by step how to implement it.

```
# This function takes x as a matrix
x_train_class <- train_class; x_test_class <- test_class
x_train_class$Class <- NULL ; x_test_class$Class <- NULL
y_train_class <- train_class$Class ; y_test_class <- test_class$Class

#Classification only.
varselrf_class <- varSelRF(x_train_class , y_train_class , ntree = 500 ,
                           recompute.var.imp = TRUE ,
                           keep.forest = TRUE , verbose = FALSE)

varselrf_class$selected.vars
```

```
## [1] "Bare.nuclei"     "Cell.shape"      "Cell.size"       "Cl.thickness"
## [5] "Normal.nucleoli"
```

This function selected four variables to make predictions and a model can now be fit used those only.

```
# Fitting the model using the selected variables
varsel_rf <- randomForest(Class~ Bare.nuclei + Cell.shape + Cell.size + Normal.nucleoli ,
                          data=train_class )

#Prediction and performance measure
pred_varsel_class <- predict(varsel_rf , newdata = test_class , type ="response")


#good classification rate
varsel_rate_class <- length(which(pred_varsel_class == test_class$Class )) /+
  nrow(test_class)

#Confusion matrix below present the results of the classification task
c_mat_4 <- table(pred_varsel_class , test_class$Class)
```

5.4.4 Performance summary

To summarize the results and the analysis of these functions used to select the best variables, we will compare their respective performance based on their prediction accuracy.

Classification task performance summary

Function	Good classification rate
Benchmark	0.9708
Benchmark SRC	0.9708
Boruta	0.9708
VSURF	0.9708
varSelRF	0.9708

On the classification data set, the number of initial covariates was 14. We added 5 noise variables to verify if the selection methods would identify them as important or not. All methods were able to remove them from the model. In this case, the resulting good classification rate was equal even if they did not select the exact same predictors. Boruta used all variables in the dataset except the added noise , vsurf approach selected 7 original variables which were presented above.This method also removed all noise. Finaly , varSelRF only used 4 variables to arrive at to the same performance. It selected Bare.nuclei, Cell.shape , cell.size and Normal.nucleoli. For model simplicity, it is better to use less variables. For that, variable selection in this context proved itself useful. The data set used was fairly simple as well. It didn't contain highly correlated covariates or an excessive number of covariates compared to the number of instances. If $n << p$ and if they were significantly correlated, then it might have had important impacts on the ability of the vanilla random forest to predict well without variable selection. Also, in the related work section, multiple studies showed the effects of such difficult data sets and their impoct on both basic prediction and variable selection.

For the regression task, the results obtained were different from one method to the other.

Regression task performance summary

Function	MSE	MAE
Benchmark	17.26	2.476
Benchmark SRC	20.18	2.58
Boruta	16.45	2.408
VSURF	61.45	5.42

The best performing variable selection approach is Boruta and it seems like the VSURF approach lead to the worse results. The Boruta approach , in this context, selected all original variables except chas and was able to delete all noise. On the other hand, the VSURF approach had cut a lot of variables. It only used 5 variables from the whole data set. This significantly impacted the performance and as presented on the table, the mean absolute error is more than twice higher than the one obtained by the basic approach.

Conclusion and overview

When fitting a random forest, it can be difficult to understand how the prediction is impacted by the choice of predictors without the concepts of variable importance and variable selection. The idea behind these methods is to break the link of the original values of each covariate with the response variable in order to estimate the negative impact on the quality of the prediction. This importance estimation is then used to perform variable selection and improve the predictions accuracy as well as a simplification of the model and the data complexity.

There are multiple approaches allowing to perfom such calculations and selections. We found that Boruta offers a robust performance on both classification and regression task while varSelRF was capable of providing great insight on interpretation and simplifying models to arrive at equivalant performance. Vsurf on it's end, is very computationally heavy and did not lead to better accuracy. These results were obtained in a setting where the data was very easy and it could be interesting to increase the difficulty of the situation by adding correlated covaraites and reducing the number of instances to identify if the methods still perfom as well.

These approaches should be rigorously tested as the performance can vary from data to data. For these tasks, the hyperparameters used were the same overall, but the number of trees, the number of variables as candidates to perform a split, the minimal size of nodes and far more parameters could be changed to assess the impact of such methods. This following tutorial was meant as a basic overview of each function, but the user has to make the best out of them by extensive hyperparameter tuning.

References

[1] Kohavi, R.;John, G.H.1997.Wrappers for feature selection.Elsevier:Artificial Intelligence p.273-324

[2] Guyon, I.;Elisseeff, A.2003.An introduction to Variable and Feature Selection.Journal of machine learning reserach

[3] Breiman, L.2001.Random forests.Cluver Kluwer Academic Publishers.Machine Learning,45,5-32

[4] Genuer, R.; Poggi, J-M.;Tuleau-Malot, C.2010.Variable Selection using random forests.Elsevier

[5] Archer, K.J; Kimes, R.V.2008.Empirical characterization of random forest variable importance measures. Computational Statistics & Data Analysis

[6] Auret, L; Aldrich, C.2011. Empirical comparison of tree ensemble variable importance measures.Chemometrics and intelligent laboratory systems p.157-170.

[7] Ishwaran, H.2007. Variable importance in binary regression trees and forests. Electronic journal of statistics.

[8] Altmann, A.; Tolosi, L.; Sander, O.; et al.2010.Permutation importance: a corrected feature importance measure. Bioinformatics;26:1340–7.

[9] Szymczak, S.; Holzinger, E.; Dasgupta, A.; et al.2016.r2VIM: a new variable selection method for random forests in genome-wide association studies. BioData Min;9:1.

[10] Janitza, A.; Boulesteix, A-L.2015. A computationally fast variable importance test for random forests for high-dimensiional data. Advances in Data Analysis and classification.

[11] Gregorutti, B.; Bertrand, M.; Philippe, S-P.2017.Correlation and variable importance in random forests.Springer Science + business media.

[12] Kursa, M.; Rudnicki, W.2010.Feature selection with the Boruta package.Journal of statistical software. V.36. Issue.11

[13] Genuer, R.;Poggi, J-M.;Tuleau-Malot, C.2015. VSURF: An R Package for Variable Selection Using Random Forests. The R Journal 7(2):19-33

[14] Murphy, M.A;Evans, J.S;Storfer,A.S.2010. Quantify Bufo boreas connectivity in Yellowstone National Park with landscape genetics. Ecology 91:252-261