randomForestSRC: Getting Started with randomForestSRC Vignette

Hemant Ishwaran, Min Lu and Udaya B. Kogalur

2021-10-15

Introduction

randomForestSRC is a CRAN compliant R-package implementing Breiman random forests (Breiman 2001) in a variety of problems. The package uses fast OpenMP parallel processing to construct forests for regression, classification, survival analysis, competing risks, multivariate, unsupervised, quantile regression and class imbalanced $q$-classification. The package is constantly being worked on and many new kinds of applications, forests and tree constructions will be added to it in the near future.

The package was developed by Hemant Ishwaran and Udaya Kogalur and is the descendent of their original (and now retired) parent package randomSurvivalForest for fitting survival data. Originally, Breiman’s random forest (RF) was only available for regression and classification. Random survival forests (Ishwaran et al. 2008) (RSF) was invented to extend RF to the setting of right-censored survival data.

randomForestSRC has evolved over time so that it can now construct many interesting forests for different applications. But then what exactly is a forest — and what exactly is a random forest?

Basically, a forest is an example of an ensemble, which is a special type of machine learning method that averages simple functions called base learners. The resulting averaged learner is called the ensemble. RF uses trees for the base-learner and builds on the ensemble concept by injecting randomization into the learning process — this is where the random in random forests comes from. Specifically, randomization is introduced in two forms. First, a randomly drawn bootstrap sample of the data is used to grow a tree (actually there is nothing special about the bootstrap, and other types of sampling are used). Second, during the grow stage at each node of the tree, a randomly selected subset of variables is chosen as candidates for splitting (this is called random feature selection). The purpose of this two-step randomization is to decorrelate trees, which reduces variance due to bagging (Breiman 1996). Furthermore, RF trees are typically grown very deeply; in fact, Breiman’s original RF classifier called for growing a classification tree to purity (one observation per terminal node). The use of deep trees, a bias reduction technique, when combined with reduced variance due to averaging and randomization, enables RF to approximate rich classes of functions while maintaining low generalization error.

Why Do Ensembles Work? A More Technical Explanation

The last part might not be so clear and it might seem like a mystery why averaging simple base-learners like trees leads to the excellent performance often reported with ensembles. Here we attempt to provide a more technical explanation for why this happens. This explanation applies to all kinds of ensembles and not just RF.

For simplicity, we consider the regression case. Let $\{\varphi_1, \ldots, \varphi_B\}$ be a collection of learners where each learner $\varphi_b : X \rightarrow \mathbb{R}$ is trained on the same learning data set $L = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$. The goal is to estimate the regression function $f(X)$ which is the conditional mean of the scalar outcome $Y$ conditional on the feature vector $X \in X'$. It is assumed that each learner is trained separately from one another. Because the learners are trained on the same data they cannot be independent, however we will assume they share the same distribution. This assumption holds for RF.
Define the ensemble estimator as the averaged value of the learners

\[ \hat{f}_{\text{ens}}(x) = \frac{1}{B} \sum_{b=1}^{B} \varphi_b(x). \]

For example if the base-learners are trees, then \( \hat{f}_{\text{ens}} \) is a tree ensemble like RF. Let \((X, Y)\) be an independent test data point with the same distribution as the learning data. The generalization error for an estimator \( \hat{f} \) is

\[ \text{Err}(\hat{f}) = \mathbb{E}_{X,Y} \left[ Y - \hat{f}(X) \right]^2. \]

Assuming a regression model \( Y = f(X) + \varepsilon \) where \( X \perp \varepsilon \) and \( \text{Var}(\varepsilon) = \sigma^2 \), using a bias-variance decomposition, we have

\[ \text{Err}(\hat{f}) = \sigma^2 + \mathbb{E}_X \left\{ \text{Bias}(\hat{f} \mid X)^2 + \text{Var}(\hat{f} \mid X) \right\} \]

where the two terms on the right are the conditional bias and conditional variance for \( \hat{f} \). Using this notation, we can establish the following result (Ueda and Nakano 1996).

**Theorem** If \( \{\varphi, \varphi_1, \ldots, \varphi_B\} \) are identically distributed learners constructed from \( L \), then

\[ \text{Err}(\hat{f}_{\text{ens}}) = \sigma^2 + \mathbb{E}_X \left\{ \text{Bias}(\varphi \mid X)^2 + \frac{1}{B} \text{Var}(\varphi \mid X) + \left(1 - \frac{1}{B}\right) \text{Cov}(X) \right\} \]

where \( \text{Cov}(X) = \text{Cov}(\varphi_b, \varphi_{b'}) \mid X \).

To understand the above Theorem, keep in mind that the number of learners, \( B \), is at our discretion and can be selected as large as we want (of course in practice this decision will be affected by computational cost, but let’s not worry about that for now). Therefore with a large enough collection of learners we can expect the generalization error to closely approximate the limiting value

\[ \lim_{B \to \infty} \text{Err}(\hat{f}_{\text{ens}}) = \sigma^2 + \mathbb{E}_X \left\{ \text{Bias}(\varphi \mid X)^2 + \text{Cov}(X) \right\}. \]

Notice that the variance has completely disappeared! This is very promising. The ideal generalization error is \( \sigma^2 \), so in order to achieve this value, we need our base-learners to have zero bias. However, the problem is the term \( \text{Cov}(X) \), which is the average covariance between any two learners. As bias decreases, learners must naturally become more complex, but this has the counter effect of increasing covariance (to reduce bias we need to use all the features for splitting, etc., but all of this makes learners more correlated with each other).

This explains why RF is the way it is. Here the base learners are randomized trees: the randomization is what reduces correlation. Also RF uses deep trees: a deep overfit tree is what reduces bias. Thus, RF balances these two terms and we can summarize the result by saying RF works because it is a variance reduction technique for low bias learners.

**Quick Start**

**Quick Installation**

Like many other R packages, the simplest way to obtain `randomForestSRC` is to install it directly from CRAN via typing the following command in R console:

```
install.packages("randomForestSRC", repos = "https://cran.us.r-project.org")
```

For more details, see `help(install.packages)`. For other methods, including building the package from our GitHub repository, see installation (Hemant Ishwaran, Lu, and Kogalur 2021a).
A Quick Example of Regression

```r
library(randomForestSRC)
# New York air quality measurements. Mean ozone in parts per billion.
airq.obj <- rfsrc(Ozone ~ ., data = airquality)
print(airq.obj)
```

```
> 1 Sample size: 153
> 2 Was data imputed: no
> 3 Number of trees: 500
> 4 Forest terminal node size: 5
> 5 Average no. of terminal nodes: 19.592
> 6 No. of variables tried at each split: 2
> 7 Total no. of variables: 5
> 8 Resampling used to grow trees: swor
> 9 Resample size used to grow trees: 97
> 10 Analysis: RF-R
> 11 Family: regr
> 12 Splitting rule: mse *random*
> 13 Number of random split points: 10
> 14 (OOB) R squared: 0.7745644
> 15 (OOB) Error rate: 245.3191

In the above output, line 5 displays the number of terminal nodes per tree averaged across the forest; line 8 displays the type of bootstrap, where `swor` refers to sampling without replacement and `swr` refers to sampling with replacement; line 9 displays the sample size for line 8 where for `swor`, the number equals to about 63.2% observations, which matches the ratio of the original data in sampling with replacement; line 10 and 11 show the type of forest where `RF-R` and `regr` refer to regression; line 12 displays splitting rule which matches the inputted argument `splitrule` and line 13 shows the number of random splits to consider for each candidate splitting variable which matches the inputted argument `nsplit`.

Model performance is displayed in the last two lines of the output in terms of out-of-bag (OOB) prediction error. A more detailed explanation for OOB is forthcoming. In the above regression model, this is evaluated as the cross-validated mean squared error (MSE) estimated via the out-of-bag data shown in line 15. Since MSE is lack of scale invariance and interpretation, standardized MSE, defined as the MSE divided by the variance of the outcome, is used and converted to R squared or the percent of variance explained by a random forest model which has an intuitive and universal interpretation, shown in line 14.

**Variable Importance (VIMP) and Partial Plot** For variable selection, estimated variable importance (VIMP) of each predictor (Breiman 2001) can be adopted, which utilizes a prediction-based approach by estimating prediction error attributable to the predictor (see VIMP vignette for more details). The VIMP can be interpreted as the increase of the standardized MSE in percentage when the corresponding predictor is randomly permuted into a noise variable. Positive VIMP values identify variables that are predictive after adjusting for all the other variables. Standard errors and P values can be generated by a bootstrapping, subsampling or delete-d-jackknife procedure (Ishwaran and Lu 2019; Hemant Ishwaran, Lu, and Kogalur 2021b). Another useful tool for interpreting the results from a RF analysis is the partial dependence plot which displays the predicted conditional mean of the outcome as a function of variable Month. In particular we see that the level of ozone is the highest around August from the right figure below.

```r
oo <- subsample(airq.obj, verbose = FALSE)
# take a delete-d-jackknife procedure for example
vimpCI <- extract.subsample(oo)$var.jk.sel.Z
```

randomForestSRC Vignettes
Overview of the Package

Building a random forest involves growing a binary tree using user supplied training data and parameters. As shown in the figure below, data types must be real valued, discrete or categorical. The response can be right-censored time and censoring information, or any combination of real, discrete or categorical information. The response can also be absent entirely.
The forest created by the package contains many useful values which can be directly extracted by the user and parsed using additional functions. Below we give an overview of some of the key functions of the package.

1. **rfsrc()**

   This is the main entry point to the package and is used to grow the random forest using user supplied training data. We refer to the resulting object as a RF-SRC grow object.

2. **rfsrc.fast()**

   A fast implementation of `rfsrc` using subsampling.

3. **quantreg()**

   Univariate and multivariate quantile regression forest for training and testing. Different methods available including the Greenwald-Khanna algorithm (Greenwald and Khanna 2001), which is especially suitable for big data due to its high memory efficiency.

4. **predict.rfsrc(), predict()**

   Used for prediction (and restoring a forest). Predicted values are obtained by dropping the user supplied test data down the grow forest. If no data is supplied, restores the original RF-SRC grow object. Restoration using the `predict` function makes it possible for users to acquire information from the grow forest without the computational expense of having to regrow a new forest. Information users might fight useful includes terminal node membership, in-sample values used to grow a tree, variable splitting behavior by tree, distance and proximity of training data, variable importance and finally performance values for specific, or groups of trees.

5. **sidClustering()**

   Clustering of unsupervised data using Staggered Interaction Data (Mantero and Ishwaran 2021). Also implements the artificial two-class approach of Breiman (Breiman 2002).

6. **vimp(), subsample(), holdout.vimp()**

   Used for variable selection:
   - `vimp()` calculates variable importance (VIMP) from a RF-SRC grow/predict object by noising up the variable (for example by permutation). Note that VIMP can also be requested directly in the grow or predict call.
   - `subsample()` constructs VIMP confidence intervals via subsampling.
   - `holdout.vimp()` calculates importance of a variable when it is removed from the model.

7. **imbalanced()**

   q-classification and G-mean VIMP for class imbalanced data (O’Brien and Ishwaran 2019).

8. **impute()**

   Fast imputation of data. Both `rfsrc()` and `predict.rfsrc()` are capable of imputing missing data (although this will be deprecated in the future). However, it is faster and more effective to pre-impute data. This function provides an efficient and fast interface for this.

9. **partial()**

   Used to extract the partial effects of a variable or variables.
Growing a Forest

A forest is specified by a model. Each model requires a slightly different formula specification, and uses model-specific split rules. This results in model-specific terminal node statistics, ensembles, and a model-specific prediction error algorithm. Below the formula table are basic examples of the different models available. For simplicity, we assume the data has already been loaded. More detailed examples are provided in other vignettes, this is just a broad over-view.

<table>
<thead>
<tr>
<th>Family</th>
<th>Example Grow Call with Formula Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Survival</td>
<td>rfsrc(Surv(time, status) ~ ., data = veteran)</td>
</tr>
<tr>
<td>Competing Risk</td>
<td>rfsrc(Surv(time, status) ~ ., data = wihs)</td>
</tr>
<tr>
<td>Regression Quantile Regression</td>
<td>rfsrc(Ozone ~ ., data = airquality)</td>
</tr>
<tr>
<td></td>
<td>quantreg(mpg ~ ., data = mtcars)</td>
</tr>
<tr>
<td>Classification Imbalanced Two-Class</td>
<td>rfsrc(Species ~ ., data = iris)</td>
</tr>
<tr>
<td></td>
<td>imbalanced(status ~ ., data = breast)</td>
</tr>
<tr>
<td>Multivariate Regression</td>
<td>rfsrc(Multivar(mpg, cyl) ~ ., data = mtcars)</td>
</tr>
<tr>
<td>Multivariate Mixed Regression</td>
<td>rfsrc(cbind(Species,Sepal.Length) ~ ., data = iris)</td>
</tr>
<tr>
<td>Multivariate Quantile Regression</td>
<td>quantreg(cbind(mpg, cyl) ~ ., data = mtcars)</td>
</tr>
<tr>
<td>Multivariate Mixed Quantile Regression</td>
<td>quantreg(cbind(Species,Sepal.Length) ~ ., data = iris)</td>
</tr>
<tr>
<td>Unsupervised sidClustering Breiman</td>
<td>rfsrc(data = mtcars) sidClustering(data = mtcars, method = &quot;sh&quot;)</td>
</tr>
</tbody>
</table>

Split Rules

In the following table, the first rule denotes the default split rule for each model specified by the option `splitrule`. The default split rule is applied when the user does not specify a split rule. The package uses the data set and formula specification to determine the model. Note that the multivariate and unsupervised split rules are a composite rule based on the default split rules for regression and classification. Each component of the composite is normalized so that the magnitude of any one y-outcome does not influence the statistic. A new Mahalanobis splitting rule has been added for multivariate regression with correlated real-valued outcomes.

<table>
<thead>
<tr>
<th>Family</th>
<th>splitrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Survival</td>
<td>logrank, bs.gradient, logrankscore</td>
</tr>
<tr>
<td>Competing Risk</td>
<td>logrankCR, logrank</td>
</tr>
<tr>
<td>Regression Quantile Regression</td>
<td>mse la.quantile.regr, quantile.regr, mse</td>
</tr>
<tr>
<td>Classification Imbalanced Two-Class</td>
<td>gini, auc, entropy gini, auc, entropy</td>
</tr>
<tr>
<td>Multivariate Regression Classification</td>
<td>mv.mse, mahalanobis mv.gini mv.mix mv.mse</td>
</tr>
<tr>
<td>Multivariate Mixed Regression</td>
<td>mv.mix</td>
</tr>
</tbody>
</table>
All models allow the use of randomized splitting specified by the option `nsplit`. When set to a non-zero positive integer, a maximum of these number of split points are chosen randomly for each of the candidate splitting variables when splitting a tree node. This significantly reduces the cost from having to consider all possible split-values. This can sometimes also improve performance, for example the choice `nsplit = 1` implements extremely randomized trees (Geurts, Ernst, and Wehenkel 2006). Traditional deterministic splitting (all split values considered) is specified by `nsplit = 0`.

There is also a pure random splitting rule, `splitrule = 'random'`, where splitting is completely independent of the y-value. This obviously has poor prediction power but can be useful for other purposes (for example, fast tuning for big data or rough but fast imputation for large data).

All models also allow the user to define a custom split rule statistic. Some basic C-programming skills are required. Examples for all the families reside in the C source code directory of the package in the file splitCustom.c. Note that recompiling and re-installing the package is necessary after modifying the source code.

**Terminal Node Statistics**

In the following table, the terminal node statistics (TNS) produced by the five models are summarized. For survival, the TNS is the Kaplan-Meier estimator and the Nelson-Aalen cumulative hazard function (CHF) at the time points of interest specified by the user, or as determined by the package if not specified. Competing risk also has two TNS’s: the cause-specific cumulative hazard estimate (CSCHF), and the cause-specific cumulative incidence function (CSCIF). Regression and classification TNS’s are the mean and class proportions respectively. For quantile regression, quantiles for each of the requested probabilities. For a multivariate model (including quantile regression), there are TNS’s for each response, whether it is real valued, discrete or categorical. The unsupervised model has no TNS, as the analysis is responseless.

<table>
<thead>
<tr>
<th>Family</th>
<th>Terminal Node Statistics, Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Survival</td>
<td>Kaplan-Meier survival, Nelson-Aalen CHF, mortality</td>
</tr>
<tr>
<td>Competing Risk</td>
<td>cause-specific CHF, cause-specific CIF, event-specific expected number of years lost</td>
</tr>
<tr>
<td>Regression Quantile Regression</td>
<td>mean, mean quantiles, moments, mean</td>
</tr>
<tr>
<td>Classification Imbalanced Two-Class</td>
<td>class proportions, class proportions, Bayes classifier</td>
</tr>
<tr>
<td>Classification Mixed Regression</td>
<td>class proportions, class proportions, q-classifier</td>
</tr>
<tr>
<td>Multivariate Regression Multivariate Classification</td>
<td>per response: mean, mean per response: class proportions, Bayes classifier same as above for Regression, Classification per response: quantiles, mean</td>
</tr>
<tr>
<td>Unsupervised sidClustering Breiman (Shi-Horvath)</td>
<td>none same as Multivariate Mixed Regression same as Classification</td>
</tr>
</tbody>
</table>
Prediction

Each model returns a predicted value for each data point which is calculated from the TNS. The predicted value is model specific and in the table is highlighted in italics. For survival, it is mortality defined as the sum of the CHF over the event (death) times (Ishwaran et al. 2008). For competing risks, for each event, the expected number of life years lost due to the event specific cause (Ishwaran et al. 2014). For regression, the mean value of the y-outcome. For classification, the estimated class probability for each class. Also returned for convenience is the Bayes classifier which is the classifier with maximal probability calculated using the estimated class probability. For two-class imbalanced, the q-classifier is returned and not the Bayes classifier. For a multivariate model, there are TNS’s for each response, whether it is real valued, discrete or categorical. The unsupervised model has no TNS, as the analysis is responseless. For sidClustering, this is similar to a multivariate model.

In-Sample and Out-of-Sample (In-Bag and Out-of Bag)

Remember that each tree is grown from a random subset of the data. Thus, the package will return both out-of-sample and in-sample predicted values from the forest, where the former are calculated using the hold out data for each tree, and the latter are from the data used to train the tree (see Breiman 2001 for more details). These values are stored in $predicted.oob and $predicted respectively. The out-of-sample values $predicted.oob should be used for inference on the training data. This is because they are cross-validated and will not over-fit the data. It is generally never recommended to use $predicted from the grow forest. In general, out-of-sample (out-of-bag, OOB) values should always be the preferred choice for analysis on the training data. We will say something more about OOB data in the illustration given at the end.

Prediction Error

In the following table, the error rate calculation for the five models is summarized. The error rate is stored in $err.rate from the forest object. For survival, it is based on Harrell’s C-index (1 minus concordance) using mortality for comparison. For Competing Risk, Harrell’s C-index using cause-specific number of years lost for comparison. For regression, performance is based on mean-squared error. For classification, performance is based on the conditional and over-all misclassification rate. Performance using the Brier score is available using option perf.type = ’brier’. The OOB AUC (area under the ROC curve) value is provided upon printing a classification forest and is obtained using the function get.auc() (see more details about Brier score and OOB AUC in our variable importance vignette). For two-class imbalanced, performance is based on G-mean by default. Other performance values are also available. For the unsupervised case, there is no prediction error implemented.

<table>
<thead>
<tr>
<th>Family</th>
<th>Prediction Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Survival</td>
<td>Harrell’s C-index (1 minus concordance)</td>
</tr>
<tr>
<td>Competing Risk</td>
<td>Harrell’s C-index (1 minus concordance)</td>
</tr>
<tr>
<td>Regression Quantile Regression</td>
<td>mean-squared error mean-squared error</td>
</tr>
<tr>
<td>Classification Imbalanced</td>
<td>misclassification, Brier score G-mean, misclassification, Brier score</td>
</tr>
<tr>
<td>Two-Class</td>
<td></td>
</tr>
<tr>
<td>Multivariate Regression</td>
<td>per response: same as above for Regression per response: same as above for Classification per response: same as above for Regression, Classification same as Multivariate Regression</td>
</tr>
<tr>
<td>Multivariate Classification</td>
<td></td>
</tr>
<tr>
<td>Multivariate Mixed Regression</td>
<td></td>
</tr>
<tr>
<td>Multivariate Quantile Regression</td>
<td></td>
</tr>
<tr>
<td>Unsupervised sidClustering</td>
<td>none same as Multivariate Mixed Regression same as Classification</td>
</tr>
<tr>
<td>Breiman (Shi-Horvath)</td>
<td></td>
</tr>
</tbody>
</table>
Illustration

Here we provide a detailed illustration that covers some of the points discussed above. A simulated classification training data set is shown on the left of the figure Tree Decision Boundary. The data set has two real valued features, \( x_1 \) and \( x_2 \). The response is a class label that can take on four values. The class sizes are equal. Each class covers a circular area. The circles touch at a single point—the origin. On the right of the figure is a tree produced from the training data with associated split points labeled.

Tree Decision Boundary

The tree was grown using a bootstrapped data set, however as mentioned other types of resampling can be used (by default the package uses 0.632 sampling without replacement). Bootstrapping draws a random set of values with replacement, where the number of cases drawn is the same as the training sample size. Bootstrapping results in well-defined training and testing subsets of the data. The bootstrap, some of which are duplicates, represents the in-bag (IB) training data, and contains approximately 0.632 unique cases. The remaining individuals define the out-of-bag (OOB) test set. See Breiman (1996, 2001) for a detailed description of these concepts. Only the bootstrap data is used to train the model and to define terminal node statistics. In this model, the terminal node statistic is the resulting frequency distribution of the class labels in a terminal node. This distribution is shown under each terminal node in the right of the figure. The decision boundary on \( x_1 \) and \( x_2 \) formed by the tree is superimposed on the data space along with the six terminal node labels.

The OOB subset for a tree does not play a role in determining its topology. Each individual in the OOB subset for a tree is passive and assigned a unique terminal node membership and terminal node statistic. An OOB ensemble statistic for each individual is formed by combining the terminal node statistics from all trees in which an individual is an OOB member. The class with the maximum frequency in the OOB ensemble statistic serves as the OOB predicted class label for the member. This procedure of taking the maximum probability is what is called the Bayes classifier and is the default classifier used in machine learning (we note that for class imbalanced data there is a much better classifier called the q-classifier which is discussed in one of the advanced vignettes Classifiers for Class Imbalanced Data, Hemant Ishwaran et al. (2021)).
In the figure labeled Forest Decision Boundary a single test data point centered at the origin is sent into the previously grown forest for prediction. Each tree provides a unique terminal node membership and statistic for this test individual. The resulting forest ensemble statistic yields a class with the maximum frequency as the predicted class label for this test data point. We note that the probability of all classes are roughly equal. This is because the training data set has one case in each class at the origin. The maximum class frequency for this example is yellow, though we can ascribe this to Monte Carlo effects. When the x-variable space is densely covered with test data points, the predicted values for the data space reveal the forest decision boundary. The boundary is also shown in the lower half of the figure. The Supplementary Code contains the R-function for simulating the data in this example.

Forest Decision Boundary

Formal Description of In-Bag (IB) and Out-Of-Bag (OOB) Ensemble: Forest Weights

To summarize the above points more mathematically, let \( F = \{ T_1, \ldots, T_B \} \) denote the forest where \( T_b = T_b(\mathcal{L}) \) is the \( b \)th RF tree. The terminal nodes of a tree are rectangular regions which form a partition of \( \mathcal{X} \), and thus each \( x \in \mathcal{X} \) must be a member of a unique terminal node, which we denote by \( R_b(x) \) for tree \( T_b \). Hereafter we shall suppress the dependence on \( \mathcal{L} \) and the sample size for notational clarity.

For the moment we will pretend that a tree is constructed using the entire data set, \( \mathcal{L} \). After this we will describe how resampling plays a role in defining the IB and OOB ensemble. Let \( Y \in \{1, \ldots, C\} \) denote the classification outcome which is a value from \( C \geq 2 \) possible classes. The target function we wish to estimate is

\[
\psi_c(x) = \mathbb{P}\{Y = c | X = x\}, \quad 1 \leq c \leq C.
\]

The tree predictor for \( \psi_c(x) \) is the relative frequency of the \( c \) class labels inside the terminal node for \( x \). It can be shown that for \( T_b \), the tree learner for estimating \( \psi_c(x) \) is

\[
\varphi_{b,c}(x) = \sum_{i=1}^n W_{i,b}(x) I\{Y_i = c\}, \quad W_{i,b}(x) = \frac{1_{\{x \in R_b(X_i)\}}}{\sum_{j=1}^n 1_{\{x \in R_b(X_j)\}}}
\]
where $0 \leq W_{i,b}(x) \leq 1$ and $\sum_{i=1}^{n} W_{i,b}(x) = 1$ are convex weights. Thus, the ensemble estimator for $\psi_c(x)$ is

$$\mathcal{P}_c(x) = \frac{1}{B} \sum_{b=1}^{B} \mathcal{P}_b,c(x) = \frac{1}{B} \sum_{b=1}^{B} \sum_{i=1}^{n} W_{i,b}(x) I\{Y_i = c\} = \sum_{i=1}^{n} W_i(x) I\{Y_i = c\},$$

where

$$W_i(x) = \frac{1}{B} \sum_{b=1}^{B} \frac{n_{i,b} \{x \in R_b(x_i)\}}{\sum_{j=1}^{n} n_{j,b} \{x \in R_b(x_j)\}}$$

are the forest weights which are convex values that sum to 1.

Now let us return to issue of resampling. Recall that $T_b$ is constructed using the IB bootstrap sample and the terminal node statistics (TNS’s) are calculated using only this data. Thus the IB tree learner excludes OOB data. This is in contrast to the tree learner just described which assumes all data is used. To address this, define integer values $n_{i,b} \geq 0$ recording the bootstrap frequency of case $i$ for $T_b$. For later use we also need to track OOB membership which we can do by noticing if $n_{i,b} = 0$. Define values $I_{i,b} \in \{0,1\}$ indicating OOB membership for case $i$:

$$I_{i,b} = \begin{cases} 1 & \text{if } n_{i,b} = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $i$ is OOB in $T_b$ iff $I_{i,b} = 1$. To define the IB ensemble we modify the definition of the forest weights to permit only those data points that are in-bag. Define the IB forest weights

$$W_{i,IB}(x) = \frac{1}{B} \sum_{b=1}^{B} \frac{n_{i,b} \{x \in R_b(x_i)\}}{\sum_{j=1}^{n} n_{j,b} \{x \in R_b(x_j)\}}.$$

We use the notation IB in the above expression to indicate the dependence on the bootstrap. For example, the denominator is the bootstrap sample size in $T_b$ for the terminal node containing $x$. Also notice that the weight from a specific tree $T_b$ is zero unless $i$ is in-bag in $T_b$; i.e. $n_{i,b} > 0$. The IB ensemble estimator for $\psi_c(x)$ is

$$\mathcal{P}_{c,IB}(x) = \frac{1}{B} \sum_{b=1}^{B} \mathcal{P}_{b,c,IB}(x) = \sum_{i=1}^{n} W_{i,IB}(x) I\{Y_i = c\}.$$

Now we describe the OOB ensemble which is defined using OOB data. The OOB predicted value for class label $c = 1, \ldots, C$ for case $i = 1, \ldots, n$ uses only those trees for which $i$ is OOB and is defined as

$$\mathcal{P}_{i,c}^{OOB} = \frac{1}{B} \sum_{b=1}^{B} \mathcal{P}_{i,b,c}^{OOB} = \sum_{j=1}^{n} W_{i,j}^{OOB} I\{Y_j = c\}$$

where $W_{i,j}^{OOB}$ is the forest weight for case $j$ in which $i$ is excluded and is defined by

$$W_{i,j}^{OOB} = \frac{1}{\sum_{b=1}^{B} I_{i,b}} \sum_{b=1}^{B} \left[ I_{i,b} n_{j,b} \{x_i \in R_b(x_j)\} \right] \left[ \sum_{k=1}^{B} n_{k,b} \{x_i \in R_b(x_k)\} \right].$$

The value in the square brackets is the tree weight for case $j$ when $i$ is OOB for tree $b$.

**R code** The following shows how to directly use forest weights to obtain IB and OOB predicted values for classification using the package. This is meant for illustration as these values are of course automatically provided by the package and are stored in $\text{predicted}$ and $\text{predicted.oob}$. The predicted class labels (using the Bayes rule) are stored in $\text{class}$ and $\text{class.oob}$.
o <- rfsrc(Species~.,iris)
phat.inb <- o$predicted
phat.oob <- o$predicted.oob
fwt.inb <- predict(o, forest.wt="inbag")$forest.wt
fwt.oob <- predict(o, forest.wt="oob")$forest.wt
phat.fwt.inb <- do.call(cbind, lapply(levels(o$yvar), function(lbl) {
  apply(fwt.inb, 1, function(wt) {
    sum(wt * (o$yvar == lbl))
  })
}))
phat.fwt.oob <- do.call(cbind, lapply(levels(o$yvar), function(lbl) {
  apply(fwt.oob, 1, function(wt) {
    sum(wt * (o$yvar == lbl))
  })
}))

print(head(data.frame(
  IB=phat.inb,
  OOB=phat.oob,
  IB.fwt=phat.fwt.inb,
  OOB.fwt=phat.fwt.oob), 20))

> IB.setosa IB.versicolor IB.virginica OOB.setosa OOB.versicolor OOB.virginica
> 1 1.000 0.000 0 1.0000000 0.000000000 0
> 2 0.998 0.002 0 0.9947368 0.005263158 0
> 3 1.000 0.000 0 1.0000000 0.000000000 0
> 4 1.000 0.000 0 1.0000000 0.000000000 0
> 5 1.000 0.000 0 1.0000000 0.000000000 0
> 6 0.996 0.004 0 0.9885057 0.011494253 0
> 7 1.000 0.000 0 1.0000000 0.000000000 0
> 8 1.000 0.000 0 1.0000000 0.000000000 0
> 9 0.996 0.004 0 0.9891892 0.010810811 0
>10 1.000 0.000 0 1.0000000 0.000000000 0
>11 1.000 0.000 0 1.0000000 0.000000000 0
>12 1.000 0.000 0 1.0000000 0.000000000 0
>13 1.000 0.000 0 1.0000000 0.000000000 0
>14 1.000 0.000 0 1.0000000 0.000000000 0
>15 0.992 0.008 0 0.9767442 0.023255814 0
>16 0.976 0.024 0 0.9390863 0.060913706 0
>17 0.998 0.002 0 0.9947090 0.005291005 0
>18 1.000 0.000 0 1.0000000 0.000000000 0
>19 0.982 0.018 0 0.9516129 0.048387097 0
>20 1.000 0.000 0 1.0000000 0.000000000 0
> IB.fwt.1 IB.fwt.2 IB.fwt.3 OOB.fwt.1 OOB.fwt.2 OOB.fwt.3
> 1 1.000 0.000 0 1.0000000 0.000000000 0
> 2 0.998 0.002 0 0.9947368 0.005263158 0
> 3 1.000 0.000 0 1.0000000 0.000000000 0
> 4 1.000 0.000 0 1.0000000 0.000000000 0
> 5 1.000 0.000 0 1.0000000 0.000000000 0
## Supplementary Code

Generalized code for simulating the spheres used in the figure Tree Decision Boundary in Illustration:

```r
get.spheres <- function(class.size=c(5000, 5000, 5000, 5000),
                        class.radius=c(1.0, 1.0, 1.0, 1.0)) {
  ## Four spheres in 3D.
  class.count <- 4;

  ## notice that forest weights are convex (sum to 1)
  print(rowSums(fwt.inb, na.rm = TRUE))

  > [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [149] 1 1

  print(rowSums(fwt.oob, na.rm = TRUE))

  > [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  > [149] 1 1
```

## Notice that forest weights are convex (sum to 1)
x.count <- 3

## We place the spheres centered on the z-axis only.
class.center = matrix(c(-1, 0, 0,
                       0, 1, 0,
                       1, 0, 0,
                       0, -1, 0),
nrow=3, ncol=class.count)

## Create the blank data frame.
spheres = matrix(0, nrow=sum(class.size), ncol=3+1)
x.names <- as.list(rep(0, 3+1))
for (j in 1:3) x.names[[j]] <- paste("X", as.character(j), sep="")
x.names[[3+1]] <- "outcome";

spheres <- data.frame(spheres)
names(spheres) <- x.names

## Use spherical co-ordinates. We "push" the points away from the
## origin by taking the square root of rho.
for (i in 1:class.count) {
  theta <- runif(class.size[i])
  phi <- runif(class.size[i])
  rho <- sqrt(runif(class.size[i]))

  x = class.radius[i] * cos(2*pi*theta) * sin(pi*phi) + class.center[1,i]
  y = class.radius[i] * rho * sin(2*pi*theta) * sin(pi*phi) + class.center[2,i]
  z = class.radius[i] * rho * cos(pi*phi) + class.center[3,i]

  spheres[((i-1)*class.size[i]) + 1 : (i*class.size[i]), 1:3] <- cbind(x, y, z)
  spheres[((i-1)*class.size[i]) + 1 : (i*class.size[i]), 3+1] <- i
}

## Artificial train and test point overrides at origin.
for (i in 1:class.count) {
  if (class.size[i] >= 1.0) {
    spheres[((i-1)*class.size[i]) + 1],] <- c(0, 0, 0, i)
  }
}

## Make the y-variable a factor.
spheres$outcome <- as.factor(spheres$outcome)

return (spheres)
}
Cite this vignette as

References


**Basics**
randomForestSRC is a fast OpenMP and memory efficient package for fitting random forests (RF) for univariate, multivariate, unsupervised, survival, competing risks, class imbalanced classification and quantile regression.
A basic grow call is of the form:

```r
rfsrc(formula, data, ntree, mtry, nodesize)
```
Grow your RF through `rfsrc`, specify your model in `formula`, provide your data frame in `data` and tune your model via `ntree`, `mtry`, `nodesize`.

**Specify a formula**

<table>
<thead>
<tr>
<th>Type</th>
<th><code>rfsrc</code> Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Survival</td>
<td><code>rfsrc(Surv(time, status) ~ ., data = veteran)</code></td>
</tr>
<tr>
<td>Competing Risk</td>
<td><code>rfsrc(Surv(time, status) ~ ., data = whis)</code></td>
</tr>
<tr>
<td>Regression</td>
<td><code>rfsrc(Ozone ~ ., data = airquality)</code></td>
</tr>
<tr>
<td>Quantile Regression</td>
<td><code>quantreg(mpg ~ ., data = mtcars)</code></td>
</tr>
<tr>
<td>Classification</td>
<td><code>rfsrc(Surv(time, status) ~ ., data = veteran)</code></td>
</tr>
<tr>
<td>Imbalanced Two-Class</td>
<td><code>rfsrc(Multivar(mp, cyl) ~ ., data = mtcars)</code></td>
</tr>
<tr>
<td>Multivariate</td>
<td><code>rfsrc(cbind(Species, Sepal.Length) ~ ., data = iris)</code></td>
</tr>
<tr>
<td>Regression</td>
<td><code>quantreg(cbind(mpg, cyl) ~ ., data = mtcars)</code></td>
</tr>
<tr>
<td>Quantile Regression</td>
<td><code>quantreg(cbind(Species, Sepal.Length) ~ ., data = iris)</code></td>
</tr>
<tr>
<td>Unsupervised</td>
<td><code>sidClustering(data = mtcars)</code></td>
</tr>
<tr>
<td>Breiman (Shi-Horvath)</td>
<td><code>sidClustering(data = mtcars, method = &quot;sh&quot;)</code></td>
</tr>
</tbody>
</table>

**Clean up and impute data**

- Choose your variables in `formula` and grow a tree.
- Your outcome(s) will be saved in `o$y` and your predictors are in `o$x` from `data` without missing values.
- To impute your data, use:
  ```r
  o <- rfsrc(y ~ a + z, data = dta, ntree = 1)
  o <- impute(y ~ a + z, data = dta)
  o <- rfsrc(y ~ a + z, data = dta, na.action = "na.impute")
  ```

**Tune mtry and nodesize**

```r
tune <- tune(quality ~ ., wine)
```
Find the optimal mtry and nodesize tuning parameter for a random forest using out-of-bag (OOB) error

```r
o$optimal mtry nodesize
1 5
```

**Grow**

Convenient interface for growing a CART tree

```r
rfsrc.cart(formula, data, ntree = 1, mtry = ncol(data), bootstrap = "none")
```
Fast OpenMP parallel computing of random forests

```r
rfsrc(formula, data, ntree = 500, mtry = NULL, ntry = NULL, nodesize = NULL, nodedepth = NULL, splitrule = NULL, nsplit = 10, importance = c("false", "true", "none", "permute", "random", "anti"), ensemble = c("all", "obh", "imbag"), bootstrap = c("by.root", "none", "by.user"), sampype = c("swor", "swr"), samp = NULL, membership = FALSE, na.action = c("na.omit", "na.impute"), nimpuse = 1, ntime = 250, cause, proximity = FALSE, distance = FALSE, forest.wt = FALSE, xvar.wt = NULL, yvar.wt = NULL, split.wt = NULL, case.wt = NULL, forest = TRUE, var.used = c("false", "all.trees", "by.tree"), split.depth = c("false", "all.trees", "by.tree"), seed = NULL, do.trace = FALSE, statistics = FALSE, ...)
```

**Inference from the Forest**

**Ensemble Predicted Value for Training Data**

```r
o <- rfsrc(Ozone ~ ., data = airquality)
o$predicted
```
Inbag and out-of-bag (OOB) predicted values for the training dataset are in `o$predicted` and `o$predicted.oob`

**Other Ensemble Values for Training Data**

- For classification problem, we also have `$class` and `$class.oob`
- For survival problem, we have `$survival` and `$survival.oob`

**Prediction Error for Assessing Model Performance**

```r
o <- rfsrc(Species ~ ., data = iris, block.size=1)
o$err.rate returns error rate; print(o) lists OOB error rate in the bottom; plot(o) plots OOB error rate along with number of trees; get.auc(y, prob) obtains the value of AUC (area under the ROC curve)
```
get.mv.error obtains error rate from a multivariate random forest

**Function Details**

- `rfsrc`: Fast approximate random forests using subsampling with forest options set to encourage computational speed
- `rfsrc.anonymous`: Random forests carefully modified so as not to save the original training data when sharing
- `sidClustering`: Clustering of unsupervised data
- `quantreg`: Univariate or multivariate quantile regression forest and returns its conditional quantile and density values
- `univariate`: Solutions to the two-class imbalanced problem
- `get.auc`: Obtains area under the ROC curve

CC by Min Lu. Learn more with the randomForestSRC homepage. randomForestSRC version 2.12.0. Updated: 2021-08
**Visualization**

- `plot.survival` plots various survival estimates
- `plot.competing.risk` plots summary curves from a competing risk analysis
- `plot.quantreg` plots quantiles obtained from a quantile regression forest

**Tree Visualization**

- `get.tree` extract a single tree from a forest and plot it on your browser
- `mtcars.unspv <- rfsrc(data = mtcars)`
- `plot(get.tree(mtcars.unspv, 5))`

**Split Statistics**

- `stat.split` acquires split statistic information. The end-cut preference (ECP) splitting property can be plotted

**Predict on New Data**

- `o.pred <- predict(object = o, newdata)`
- Predicted values for the new dataset are in `o.pred$predicted`
- `get.mv.predicted` returns predicted value for multivariate regression analysis

**Restore**

Restoration using the `predict` function makes it possible for users to acquire information from the grow forest without the computational expense of having to regrow a new forest.

Examples of restore are as follows (extract: proximity, variable splitting behavior, performance over specific trees)

```r
o <- rfsrc(Ozone ~ ., data = airquality)
predict(o, proximity = TRUE)$proximity
predict(o.obj, var.used = "by.tree")$var.used
```

**Variable Selection and Hunting**

- `var.select(formula, data, method)` Variable selection or hunting by setting method
  - `md` Minimal depth (default)
  - `vh` Variable hunting
  - `vh.vimp` Variable hunting with VIMP

**Partial Plot**

- `plot.variable(o, xvar.names)`

**Partial Dependence Plot**

- `plot.variable(o, xvar.names, partial = TRUE) and partial`

**Variable Selection**

**Variable Importance (VIMP)**

```r
o <- rfsrc(Species ~ ., iris, importance = TRUE)
```

- `o$importance` returns permutation VIMP
- `plot(o)` plots VIMP when setting `importance` to "permute" or "TRUE" in `rfsrc` or using `vimp`

**Continuous predictor:**

- `get.partial.plot.data` is a handy function that parses the output from "partial.rfsrc" in format suitable for plots

**Minimal Depth**

- `max.subtree` extracts minimal depth and maximal subtree information used for variable selection and identifying interactions between variables