Random Forest in R

JR Ferrer-Paris

02/08/2021

Abstract

This document will walk you through examples to fit random forest models for classification using the *randomForest* package and two different datasets. This is part of the UNSW codeRs workshop: *Introduction to Classification Trees and Random Forests in R* at https://github.com/UNSW-codeRs/workshop-random-forests

UNSW codeRs is a student and staff run community dedicated for 'R' users for anyone who wants to further develop their coding skills. It is our goal to create a safe and open space for members to share and gain new experiences relating to R, coding and statistics.

https://unsw-coders.netlify.app/

Overview

The random forest algorithm seeks to improve on the performance of a single decision tree by taking the average of many trees. Thus, a random forest can be viewed as an **ensemble** method, or model averaging approach.

Random forests use **bagging** and **variable randomization** to create different conditions for each tree. As a result, the average of these trees tends to be more accurate overall than a single tree.

What data do we need?

The same as any regular classification decision tree!

- Y: The output or response variable is a categorical variable with two or more classes (in R: factor with two or more levels)
- X: A set of predictors or features, might be a mix of continuous and categorical variables, they should not have any missing values

Load data

Here we will work again with two examples.

First, we will use the *iris* dataset from base R. This dataset has 150 observations with four measurements (continuous variables) for three species (categorical variable with three categories):

```
data(iris)
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

As a second example we will use the Breast Cancer dataset from the *mlbench* package. This dataset has 699 observations with 9 nominal or ordinal variables describing cell properties and the output or target variable is the class of tumor with two possible values: bening or malignant:

```
require(mlbench)
```

```
## Loading required package: mlbench
```

```
data(BreastCancer)
str(BreastCancer)
```

```
## 'data.frame':    699 obs. of  11 variables:
##  $ Id             : chr  "1000025" "1002945" "1015425" "1016277" ...
##  $ Cl.thickness   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 5 5 3 6 4 8 1 2 2 4 ...
##  $ Cell.size      : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 1 1 2 ...
##  $ Cell.shape     : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 2 1 1 ...
##  $ Marg.adhesion  : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 5 1 1 3 8 1 1 1 1 ...
##  $ Epith.c.size   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 2 7 2 3 2 7 2 2 2 2 ...
##  $ Bare.nuclei    : Factor w/ 10 levels "1","2","3","4",..: 1 10 2 4 1 10 10 1 1 1 ...
##  $ Bl.cromatin    : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
##  $ Normal.nucleoli: Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
##  $ Mitoses        : Factor w/ 9 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
##  $ Class          : Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

What package to use

Random Forests are implemented in several packages:

- *randomForest*: Breiman and Cutler's Random Forests for Classification and Regression
- *ranger*: A Fast Implementation of Random Forests
- *party*: A Laboratory for Recursive Partytioning
- *RandomForestsGLS*: Random Forests for Dependent Data

- *randomForestSRC*: Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC)
- *Rborist*: Extensible, Parallelizable Implementation of the Random Forest Algorithm
- etc.

Some helper packages are:

- *randomForestExplainer*: Explaining and Visualizing Random Forests in Terms of Variable Importance
- *vip*: Variable Importance Plots
- *varImp*: Variable Importance Plots
- *caret*: Classification and Regression Training
- *tidymodels*: tidy model framework (model workflows)

Load packages

Here we will work with package *randomForest*.

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

Fit a model

iris dataset

Let's start with a familiar dataset. Fitting the model is very straightforward.

```
set.seed(3)
rf_model = randomForest(Species ~ ., data = iris, ntree=30)
```

The `print` method shows basic information about the fitted model:

```
print(rf_model)
```

```
##
## Call:
##  randomForest(formula = Species ~ ., data = iris, ntree = 30)
##                Type of random forest: classification
##                      Number of trees: 30
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 4.67%
## Confusion matrix:
##            setosa versicolor virginica class.error
## setosa         50          0         0        0.00
## versicolor      0         47         3        0.06
```

```
## virginica       0          4        46        0.08
```

We will go through these output in more detail, but first take a look at the "OOB estimate of error rate", this shows us how accurate our model is. $accuracy = 1 - error\ rate$. OOB stands for "out of bag" - and bag is short for "bootstrap aggregation". So OOB estimates performance by comparing the predicted outcome value to the actual value across all trees using only the observations that were not part of the training data for that tree.

Tree complexity Ok, now let's look at the top of the output: `Number of trees: 30`. So a forest is made up of trees, right? Let's see tree number three:

```
getTree(rf_model, 3, labelVar=TRUE)
```

```
##    left daughter right daughter    split var split point status prediction
## 1              2              3 Petal.Length        2.45      1       <NA>
## 2              0              0         <NA>        0.00     -1     setosa
## 3              4              5  Petal.Width        1.65      1       <NA>
## 4              6              7 Petal.Length        4.95      1       <NA>
## 5              8              9  Petal.Width        1.75      1       <NA>
## 6              0              0         <NA>        0.00     -1 versicolor
## 7             10             11 Petal.Length        5.45      1       <NA>
## 8             12             13 Sepal.Length        5.80      1       <NA>
## 9             14             15 Petal.Length        4.85      1       <NA>
## 10            16             17  Petal.Width        1.55      1       <NA>
## 11             0              0         <NA>        0.00     -1  virginica
## 12             0              0         <NA>        0.00     -1  virginica
## 13             0              0         <NA>        0.00     -1 versicolor
## 14             0              0         <NA>        0.00     -1  virginica
## 15             0              0         <NA>        0.00     -1  virginica
## 16             0              0         <NA>        0.00     -1  virginica
## 17             0              0         <NA>        0.00     -1 versicolor
```

This is not very user friendly... but normally we would focus on the forest rather than the individual trees.

Key question here is, why are these trees so complex compared to the ones we saw before? We can control the complexity of the trees with `nodesize` and `maxnode` parameters. For example a large node size and small number of nodes will result in simpler/shorter trees, but this could affect OOB error rates:

```
rf2 = randomForest(Species ~ ., data = iris, ntree=30, nodesize=20, maxnodes=5)
print(rf2)
```

```
##
## Call:
##  randomForest(formula = Species ~ ., data = iris, ntree = 30,      nodesize = 20, maxnode
##                Type of random forest: classification
```

```
##                       Number of trees: 30
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 4.67%
## Confusion matrix:
##            setosa versicolor virginica class.error
## setosa         50          0         0        0.00
## versicolor      0         47         3        0.06
## virginica       0          4        46        0.08
```

```
getTree(rf2, 3, labelVar=TRUE)
```

```
##   left daughter right daughter    split var split point status prediction
## 1             2              3 Petal.Length        2.75      1       <NA>
## 2             0              0         <NA>        0.00     -1     setosa
## 3             4              5 Sepal.Length        6.15      1       <NA>
## 4             6              7  Petal.Width        1.70      1       <NA>
## 5             8              9  Petal.Width        1.75      1       <NA>
## 6             0              0         <NA>        0.00     -1 versicolor
## 7             0              0         <NA>        0.00     -1  virginica
## 8             0              0         <NA>        0.00     -1 versicolor
## 9             0              0         <NA>        0.00     -1  virginica
```

Variable randomization We can also control the complexity by tweaking the`number of variables sampled in each split` with the parameter `mtry`. By default `randomForest` will test one third of the variables in each split and choose the best one. Here we can force it to use four variables each time:

```
rf3 = randomForest(Species ~ ., data = iris, ntree=30, mtry=4)
print(rf3)
```

```
##
## Call:
##  randomForest(formula = Species ~ ., data = iris, ntree = 30,      mtry = 4)
##                Type of random forest: classification
##                      Number of trees: 30
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 6%
## Confusion matrix:
##            setosa versicolor virginica class.error
## setosa         50          0         0        0.00
## versicolor      0         46         4        0.08
## virginica       0          5        45        0.10
```

Tuning hyper-parameters These three hyper-parameters (`maxnode`, `nodesize` and `mtry`) can be tuned to get better results:

```
for (ns in c(5,25)) {
  for (mn in c(5,15)) {
    for (mt in c(2,4)) {
      rf1 <- randomForest(Species ~ ., data = iris, nodesize=ns, maxnodes=mn, mtry=mt, ntree
      cat(sprintf("nodesize=%02d maxnodes=%02d mtry=%s OOB-error=%0.4f\n", ns, mn,mt, rf1$er
    }
  }
}
```

```
## nodesize=05 maxnodes=05 mtry=2 OOB-error=0.0333
## nodesize=05 maxnodes=05 mtry=4 OOB-error=0.0467
## nodesize=05 maxnodes=15 mtry=2 OOB-error=0.0533
## nodesize=05 maxnodes=15 mtry=4 OOB-error=0.0400
## nodesize=25 maxnodes=05 mtry=2 OOB-error=0.0533
## nodesize=25 maxnodes=05 mtry=4 OOB-error=0.0533
## nodesize=25 maxnodes=15 mtry=2 OOB-error=0.0533
## nodesize=25 maxnodes=15 mtry=4 OOB-error=0.0400
```

However this is subject to variability due to randomness, and a better fine-tuning requires several replicate runs of each combinations. Some functions can be used to do this more efficiently, for example `randomForest::tuneRF` or the workflow in packages `caret` or `tidymodels`.

Breast cancer dataset

Let's look at the more complex example of breast cancer dataset to look at other aspects of the randomForest object.

```
set.seed(83)

BC.data <- subset(BreastCancer[,-1],!is.na(Bare.nuclei))

rf_model = randomForest(Class ~ ., data = BC.data,importance=TRUE)
print(rf_model)
```

```
##
## Call:
##  randomForest(formula = Class ~ ., data = BC.data, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 2.64%
## Confusion matrix:
##           benign malignant class.error
## benign       431        13  0.02927928
## malignant      5       234  0.02092050
```

Variable importance Random Forest estimates variable importance in two different ways: one is by comparing the accuracy of the original prediction with the accuracy of a prediction based on a randomly shuffled (permuted) variable. If a variable is important then the model's accuracy will suffer a large drop when it is randomly shuffled. But if the accuracy is similar, then the variable is not important to the model.

The second measure is the total decrease in node impurities from splitting on the variable, averaged over all trees. For classification, the node impurity is measured by the Gini index. It's basically a measure of diversity or dispersion - a higher gini means the model is classifying better.

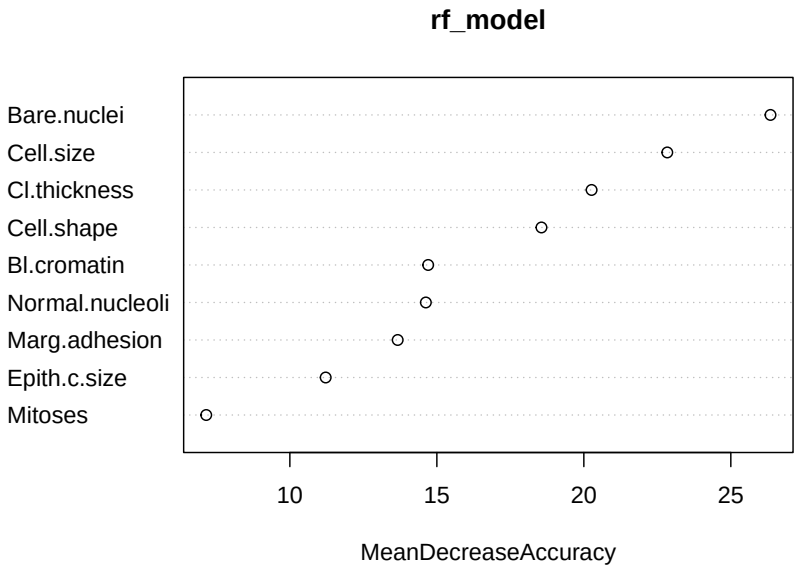
```
importance(rf_model)
```

```
##                    benign malignant MeanDecreaseAccuracy MeanDecreaseGini
## Cl.thickness    15.978870 21.025937             20.263316        13.796357
## Cell.size       17.360307 14.849966             22.838147        91.871818
## Cell.shape       8.775332 16.733826             18.558343        62.929939
## Marg.adhesion    7.592527 12.550380             13.667630         6.892496
## Epith.c.size     8.769258  8.866896             11.218363        25.301932
## Bare.nuclei     19.871647 24.743863             26.348608        54.277415
## Bl.cromatin     10.148544 12.009782             14.704710        31.690796
## Normal.nucleoli 12.950851 10.620751             14.626091        21.788466
## Mitoses          7.193980  1.368730              7.154672         2.085496
```

Notice for example that `Bl.cromatin` and `Normal.nucleoli` have similar values of mean decrease in overall accuracy when permuted, but they have different importance for each class.

The `varImpPlot` function can be used to produce a plot, use `type=1` for `MeanDecreaseAccuracy` and `type=2` for `MeanDecreaseGini`.

```
varImpPlot(rf_model,type=1)
```



Confusion matrix and votes Now let's look at the prediction for each observation. The confusion matrix compares observed with predicted values and shows the classification error for each class.

```
rf_model$confusion
```

```
##           benign malignant class.error
## benign       431        13  0.02927928
## malignant      5       234  0.02092050
```

The classification error is very low, but we might be interested in exploring those cases of misclassification (we don't want a malign tumor to be misclassified). Each observation has a number of OOB predictions, these are considered "votes" for a class, and the observation is assigned to the class with the majority of votes. Let's look at the *votes* object for this model:

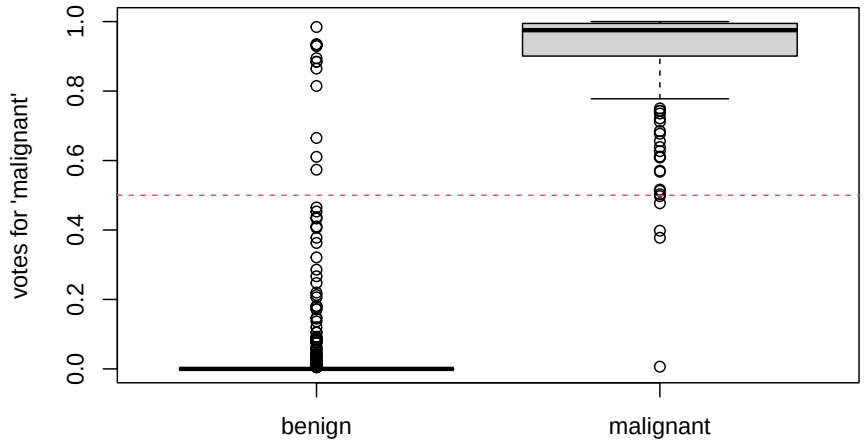
```
head(rf_model$votes)
```

```
##       benign malignant
## 1 1.00000000 0.0000000
## 2 0.18539326 0.8146067
## 3 1.00000000 0.0000000
## 4 0.07058824 0.9294118
## 5 1.00000000 0.0000000
## 6 0.00000000 1.0000000
```

We can compare the votes for the second class ('malignant') and compare them

between the two classes:

```
boxplot(rf_model$votes[,2]~rf_model$y,xlab="",ylab="votes for 'malignant'")
abline(h=0.5,lty=2,col=2)
```



Using a lower threshold (for example 0.4 or 0.3) to classify a tumor as malignant would reduce the false negative rate, but increase false positive rate.

There are other machine-learning methods that focus on improving the performance for these hard-to-classify observations, but this would be a subject for another workshop!

Post-scriptum

Additional resources

- Davis David **Random Forest Classifier Tutorial: How to Use Tree-Based Algorithms for Machine Learning**
- Evan Muzzall and Chris Kennedy **Introduction to Machine Learning in R**
- Victor Zhou **Random Forests for Complete Beginners**
- Bradley Boehmke & Brandon Greenwell **Hands-On Machine Learning with R**
- Julia Kho **Why Random Forest is My Favorite Machine Learning Model**

- JanBask Training **A Practical guide to implementing Random Forest in R with example**

```
sessionInfo()
```

Session information:

```
## R version 4.1.0 (2021-05-18)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Big Sur 10.16
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_AU.UTF-8/en_AU.UTF-8/en_AU.UTF-8/C/en_AU.UTF-8/en_AU.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] randomForest_4.6-14 mlbench_2.1-3
##
## loaded via a namespace (and not attached):
##  [1] compiler_4.1.0    magrittr_2.0.1    tools_4.1.0       htmltools_0.5.1.1
##  [5] yaml_2.2.1        stringi_1.7.3     rmarkdown_2.9     highr_0.9
##  [9] knitr_1.33        stringr_1.4.0     xfun_0.24         digest_0.6.27
## [13] rlang_0.4.11      evaluate_0.14
```