

Introduction

Treat "forests" well. Not for the sake of nature, but for solving problems too!

Random Forest is one of the most versatile machine learning algorithms available today. With its built-in ensembling capacity, the task of building a decent generalized model (on any dataset) gets much easier. However, I've seen people using random forest as a black box model; i.e., they don't understand what's happening beneath the code. They just code.

In fact, the easiest part of machine learning is *coding*. If you are new to machine learning, the random forest algorithm should be on your tips. Its ability to solve—both regression and classification problems along with robustness to correlated features and variable importance plot gives us enough head start to solve various problems.

Most often, I've seen people getting confused in bagging and random forest. Do you know the difference?

In this article, I'll explain the complete concept of random forest and bagging. For ease of understanding, I've kept the explanation simple yet enriching. I've used MLR, data.table packages to implement bagging, and random forest with parameter tuning in R. Also, you'll learn the techniques I've used to improve model accuracy from ~82% to 86%.

Table of Contents

1. What is the Random Forest algorithm?
2. How does it work? (Decision Tree, Random Forest)
3. What is the difference between Bagging and Random Forest?
4. Advantages and Disadvantages of Random Forest
5. Solving a Problem
 - Parameter Tuning in Random Forest

What is the Random Forest algorithm?

Random forest is a tree-based algorithm which involves building several trees (decision trees), then combining their output to improve generalization ability of the model. The method of combining trees is known as an ensemble method. Ensembling is nothing but a combination of weak learners (individual trees) to produce a strong learner.

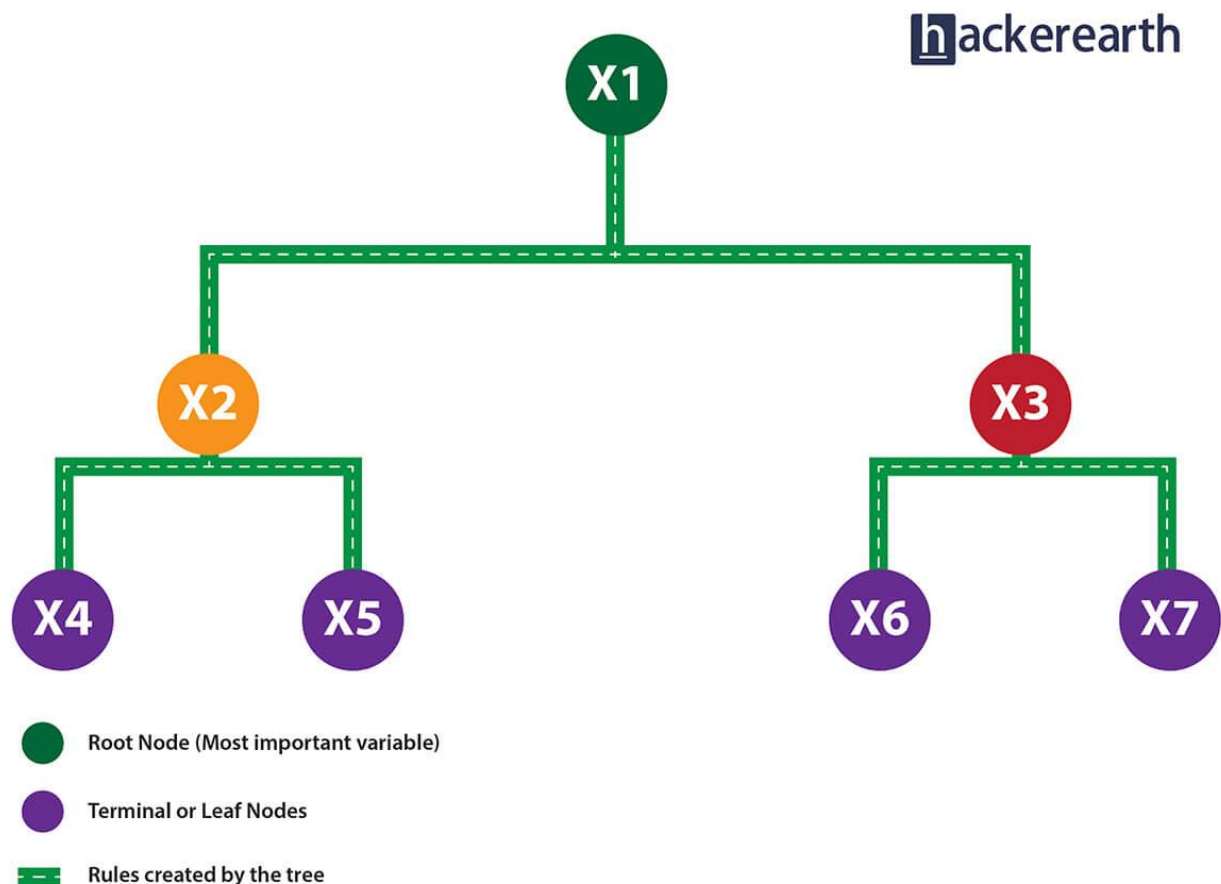
Say, you want to watch a movie. But you are uncertain of its reviews. You ask 10 people who have watched the movie. 8 of them said " the movie is fantastic." Since the majority is in favor, you decide to watch the movie. This is how we use ensemble techniques in our daily life too.

Random Forest can be used to solve regression and classification problems. In regression problems, the dependent variable is continuous. In classification problems, the dependent variable is categorical.

Trivia: The random Forest algorithm was created by [Leo Brieman](#) and [Adele Cutler](#) in 2001.

How does it work? (Decision Tree, Random Forest)

To understand the working of a random forest, it's crucial that you understand a **tree**. A tree works in the following way:

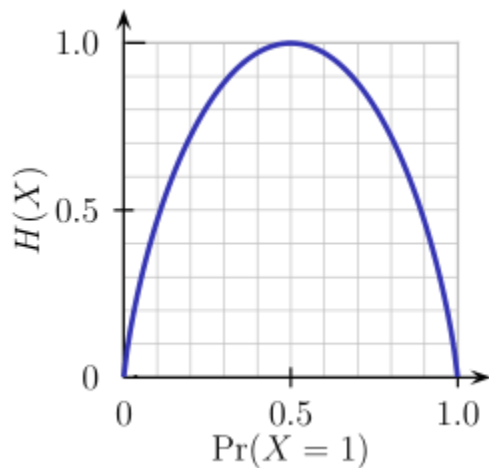


1. Given a data frame ($n \times p$), a tree stratifies or partitions the data based on rules (if-else). Yes, a tree creates rules. These rules divide the data set into distinct and non-overlapping regions. These rules are determined by a variable's contribution to the homogeneity or pureness of the resultant child nodes (X2,X3).

2. In the image above, the variable X1 resulted in highest homogeneity in child nodes, hence it became the root node. A variable at root node is also seen as the most important variable in the data set.3. But how is this homogeneity or pureness determined? In other words, how does the tree decide at which variable to split?

- In **regression trees** (where the output is predicted using the mean of observations in the terminal nodes), the splitting decision is based on minimizing RSS. The variable which leads to the greatest possible reduction in RSS is chosen as the root node. The tree splitting takes a **top-down greedy** approach, also known as *recursive binary splitting*. We call it "greedy" because the algorithm cares to make the best split at the current step rather than saving a split for better results on future nodes.
- In **classification trees** (where the output is predicted using mode of observations in the terminal nodes), the splitting decision is based on the following methods:
 - Gini Index** - It's a measure of node purity. If the Gini index takes on a smaller value, it suggests that the node is pure. For a split to take place, the Gini index for a child node should be less than that for the parent node.
 - Entropy** - Entropy is a measure of node impurity. For a binary class (a,b), the formula to calculate it is shown below. Entropy is maximum at $p = 0.5$. For $p(X=a)=0.5$ or $p(X=b)=0.5$ means, a new observation has a 50%-50% chance of getting classified in either classes. The entropy is minimum when the probability is 0 or 1.

$$\text{Entropy} = - p(a) \cdot \log(p(a)) - p(b) \cdot \log(p(b))$$



In a nutshell, every tree attempts to create rules in such a way that the resultant terminal nodes could be as pure as possible. Higher the purity, lesser the uncertainty to make the decision.

But a decision tree suffers from high variance. "High Variance" means getting high prediction error on unseen data. We can overcome the variance problem by using more data for training. But since the data set available is limited to us, we can use resampling techniques like bagging and random forest to generate more data.

Building many **decision trees** results in a **forest**. A random forest works the following way:

1. First, it uses the Bagging (Bootstrap Aggregating) algorithm to create random samples. Given a data set D1 (n rows and p columns), it creates a new dataset (D2) by sampling n cases at random with replacement from the original data. About 1/3 of the rows from D1 are left out, known as Out of Bag(OOB) samples.
2. Then, the model trains on D2. OOB sample is used to determine unbiased estimate of the error.
3. Out of p columns, $P \ll p$ columns are selected at each node in the data set. The P columns are selected at random. Usually, the default choice of P is $p/3$ for regression tree and $P = \sqrt{p}$ for classification tree.
4. Unlike a tree, no pruning takes place in random forest; i.e, each tree is grown fully. In decision trees, pruning is a method to avoid overfitting. Pruning means selecting a subtree that leads to the lowest test error rate. We can use cross validation to determine the test error rate of a subtree.
5. Several trees are grown and the final prediction is obtained by averaging or voting.

Each tree is grown on a different sample of original data. Since random forest has the feature to calculate OOB error internally, cross validation doesn't make much sense in random forest.

What is the difference between Bagging and Random Forest?

Many a time, we fail to ascertain that bagging is not same as random forest. To understand the difference, let's see how bagging works:

1. It creates randomized samples of the data set (just like random forest) and grows trees on a different sample of the original data. The remaining 1/3 of the sample is used to estimate unbiased OOB error.
2. It considers all the features at a node (for splitting).
3. Once the trees are fully grown, it uses averaging or voting to combine the resultant predictions.

Aren't you thinking, "If both the algorithms do same thing, what is the need for random forest? Couldn't we have accomplished our task with bagging?" NO!

The need for random forest surfaced after discovering that the bagging algorithm results in correlated trees when faced with a data set having strong predictors. Unfortunately, averaging several highly correlated trees doesn't lead to a large reduction in variance.

But how do correlated trees emerge? Good question! Let's say a data set has a **very strong predictor**, along with other **moderately** strong predictors. In bagging, a tree grown every time would consider the **very strong predictor** at its root node, thereby resulting in trees similar to each other.

The **main difference between** random forest and bagging is that random forest considers only a subset of predictors at a split. This results in trees with different predictors at top split, thereby resulting in **decorrelated trees** and more reliable average output. That's why we say random forest is robust to correlated predictors.

Advantages and Disadvantages of Random Forest

Advantages are as follows:

1. It is robust to correlated predictors.
2. It is used to solve both regression and classification problems.
3. It can be also used to solve unsupervised ML problems.
4. It can handle thousands of input variables without variable selection.
5. It can be used as a feature selection tool using its variable importance plot.
6. It takes care of missing data internally in an effective manner.

Disadvantages are as follows:

1. The Random Forest model is difficult to interpret.
2. It tends to return erratic predictions for observations out of range of training data. For example, the training data contains two variable x and y. The range of x variable is 30 to 70. If the test data has x = 200, random forest would give an unreliable prediction.
3. It can take longer than expected time to computer a large number of trees.

Solving a Problem (Parameter Tuning)

Let's take a data set to compare the performance of bagging and random forest algorithms. Along the way, I'll also explain important parameters used for parameter tuning. In R, we'll use MLR and data.table package to do this analysis. I've taken the [Adult dataset](#) from the UCI machine learning repository. You can download the data from [here](#).

This data set presents a binary classification problem to solve. Given a set of features, we need to predict if a person's salary is $\leq 50K$ or $\geq 50K$. Since the given data isn't well structured, we'll need to make some modification while reading the data set.

```
#set working directory
path <- "~/December 2016/RF_Tutorial"
setwd(path)

#load libraries
library(data.table)
library(mlr)
library(h2o)

#set variable names
setcol <- c("age", "workclass", "fnlwgt", "education", "education-num", "marital-status", "occupation",
"relationship", "race", "sex", "capital-gain", "capital-loss", "hours-per-week", "native-country", "target")

#load data
train <- read.table("adultdata.txt", header = F, sep = ",", col.names = setcol, na.strings = c(" ?"), stringsAsFactors = F)
test <- read.table("adulttest.txt", header = F, sep = ",", col.names = setcol, skip = 1, na.strings = c(" ?"), stringsAsFactors = F)
After we've loaded the data set, first we'll set the data class to data.table. data.table is the most powerful R package made
for faster data manipulation.
setDT(train)
setDT(test)
Now, we'll quickly look at given variables, data dimensions, etc.
dim(train)
dim(test)
str(train)
str(test)
As seen from the output above, we can derive the following insights:
```

1. The train data set has 32,561 rows and 15 columns.
2. The test data has 16,281 rows and 15 columns.
3. Variable target is the dependent variable.
4. The target variable in train and test data is different. We'll need to match them.
5. All character variables have a leading whitespace which can be removed.

We can check missing values using:

```
#check missing values
table(is.na(train))
FALSE TRUE
484153 4262
```

```
sapply(train, function(x) sum(is.na(x))/length(x))*100
table(is.na(test))
FALSE TRUE
242012 2203
```

```
sapply(test, function(x) sum(is.na(x))/length(x))*100`<br/>
```

As seen above, both train and test datasets have missing values. The sapply function is quite handy when it comes to performing column computations. Above, it returns the percentage of missing values per column. Now, we'll preprocess the data to prepare it for training. In R, random forest internally takes care of missing values using mean/ mode imputation. Practically speaking, sometimes it takes longer than expected for the model to run.

Therefore, in order to avoid waiting time, let's impute the missing values using median / mode imputation method; i.e. missing values in the integer variable will be imputed with median and factor variables will be imputed with mode (most frequent value). We'll use the impute function from MLR package, which is enabled with several unique methods for missing value imputation:

```
imp1 <- impute(data = train,target = "target",classes = list(integer=imputeMedian(), factor=imputeMode()))
imp2 <- impute(data = test,target = "target",classes = list(integer=imputeMedian(), factor=imputeMode()))
train <- imp1$data
test <- imp2$data
```

Being a binary classification problem, you are always advised to check if the data is imbalanced or not. We can do it in the following way:

```
setDT(train)[,.N/nrow(train),target]
  target  V1
1: <=50K 0.7591904
2: >50K  0.2408096
```

```
setDT(test)[,.N/nrow(test),target]
  target  V1
1: <=50K 0.7637737
2: >50K  0.2362263
```

If you observe carefully, the value of the target variable is different in test and train. For now, we can consider it a typo error and correct all the test values. Also, we see that 75% of people in train data have income <=50K. Imbalanced classification problems are known to be more skewed with a binary class distribution of 90% to 10%. Now, let's proceed and clean the target column in test data.

```
test[,target := substr(target,start = 1,stop = nchar(target)-1)]
```

We've used the substr function to return the substring from a specified start and end position. Next, we'll remove the leading whitespaces from all character variables. We'll use str_trim function from stringr package.

```
library(stringr)
char_col <- colnames(train)[sapply(train,is.character)]
for(i in char_col)
  set(train,j=i,value = str_trim(train[[i]],side = "left"))
```

Using sapply function, we've extracted the column names which have character class. Then, using a simple for - set loop we traversed all those columns and applied the str_trim function. Before we start model training, we should convert all character variables to factor. MLR package treats character class as unknown.

```
fact_col <- colnames(train)[sapply(train,is.character)]
for(i in fact_col)
  set(train,j=i,value = factor(train[[i]]))
```

```
for(i in fact_col)
```

```
  set(test,j=i,value = factor(test[[i]]))
```

Let's start with modeling now. MLR package has its own function to convert data into a task, build learners, and optimize learning algorithms. I suggest you stick to the modeling structure described below for using MLR on any data set.

```
#create a task
```

```
traintask <- makeClassifTask(data = train,target = "target")
```

```
testtask <- makeClassifTask(data = test,target = "target")
```

```
#create learner
```

```
bag <- makeLearner("classif.rpart",predict.type = "response")
```

```
bag.lrn <- makeBaggingWrapper(learner = bag,bw.its = 100,bw.replace = TRUE)
```

I've set up the bagging algorithm which will grow 100 trees on randomized samples of data with replacement. To check the performance, let's set up a validation strategy too:

```
#set 5 fold cross validation
```

```
rdesc <- makeResampleDesc("CV",iters=5L)
```

For faster computation, we'll use parallel computation backend. Make sure your machine / laptop doesn't have many programs running at backend.

```
#set parallel backend (Windows)
```

```
library(parallelMap)
```

```
library(parallel)
```

```
parallelStartSocket(cpus = detectCores())
```

For linux users, the function `parallelStartMulticore(cpus = detectCores())` will activate parallel backend. I've used all the cores here.

```
r <- resample(learner = bag.lrn , task = traintask, resampling = rdesc, measures = list(tpr,fpr,fnr,fpr,acc) ,show.info = T)
```

```
#[Resample] Result:
```

```
# tpr.test.mean=0.95 #fnr.test.mean=0.0505 #fpr.test.mean=0.487 #acc.test.mean=0.845
```

Being a binary classification problem, I've used the components of [confusion matrix](#) to check the model's accuracy. With 100 trees, bagging has returned an accuracy of 84.5%, which is way better than the baseline accuracy of 75%. Let's now check the performance of random forest.

```
#make randomForest learner
```

```
rf.lrn <- makeLearner("classif.randomForest")
```

```
rf.lrn$par.vals <- list(ntree = 100L, importance=TRUE)
```

```
r <- resample(learner = rf.lrn, task = traintask, resampling = rdesc, measures = list(tpr,fpr,fnr,fpr,acc), show.info = T)
```

```
# Result:
```

```
# tpr.test.mean=0.996 #fpr.test.mean=0.72 #fnr.test.mean=0.0034 #acc.test.mean=0.825
```

On this data set, random forest performs worse than bagging. Both used 100 trees and random forest returns an overall accuracy of 82.5 %. An apparent reason being that this algorithm is messing up classifying the negative class. As you can see, it classified 99.6% of the positive classes correctly, which is way better than the bagging algorithm. But it incorrectly classified 72% of the negative classes.

Internally, random forest uses a cutoff of 0.5; i.e., if a particular unseen observation has a probability higher than 0.5, it will be classified as $\leq 50K$. In random forest, we have the option to customize the internal cutoff. As the false positive rate is very high now, we'll increase the cutoff for positive classes ($\leq 50K$) and accordingly reduce it for negative classes ($\geq 50K$). Then, train the model again.

```
#set cutoff
```

```
rf.lrn$par.vals <- list(ntree = 100L, importance=TRUE, cutoff = c(0.75,0.25))
```

```
r <- resample(learner = rf.lrn, task = traintask, resampling = rdesc, measures = list(tpr,fpr,fnr,fpr,acc), show.info = T)
```

```
#Result: tpr.test.mean=0.934 #fpr.test.mean=0.43 #fnr.test.mean=0.0662 #acc.test.mean=0.846
```

As you can see, we've improved the accuracy of the random forest model by 2%, which is slightly higher than that for the bagging model. Now, let's try and make this model better.

Parameter Tuning: Mainly, there are three parameters in the random forest algorithm which you should look at (for tuning):

- **ntree** - As the name suggests, the number of trees to grow. Larger the tree, it will be more computationally expensive to build models.
- **mtry** - It refers to how many variables we should select at a node split. Also as mentioned above, the default value is $p/3$ for regression and \sqrt{p} for classification. We should always try to avoid using smaller values of mtry to avoid overfitting.

- **nodesize** - It refers to how many observations we want in the terminal nodes. This parameter is directly related to tree depth. Higher the number, lower the tree depth. With lower tree depth, the tree might even fail to recognize useful signals from the data.

Let get to the playground and try to improve our model's accuracy further. In MLR package, you can list all tuning parameters a model can support using:

```
getParamSet(rf.lrn)

#set parameter space
params <- makeParamSet(makeIntegerParam("mtry",lower = 2,upper = 10),makeIntegerParam("nodesize",lower = 10,upper = 50))

#set validation strategy
rdesc <- makeResampleDesc("CV",iters=5L)

#set optimization technique
ctrl <- makeTuneControlRandom(maxit = 5L)

#start tuning
tune <- tuneParams(learner = rf.lrn, task = traintask, resampling = rdesc, measures = list(acc), par.set = params, control = ctrl, show.info = T)
#[Tune] Result: mtry=2 : nodesize=23 : acc.test.mean=0.858
After tuning, we have achieved an overall accuracy of 85.8%, which is better than our previous random forest model. This way you can tweak your model and improve its accuracy.
```

I'll leave you here. The complete code for this analysis can be downloaded from [Github](#).

Summary

Don't stop here! There is still a huge scope for improvement in this model. Cross validation accuracy is generally more optimistic than true test accuracy. To make a prediction on the test set, minimal data preprocessing on categorical variables is required. Do it and share your results in the comments below.

My motive to create this tutorial is to get you started using the random forest model and some techniques to improve model accuracy. For better understanding, I suggest you read more on confusion matrix. In this article, I've explained the working of decision trees, random forest, and bagging.

Did I miss out anything? Do share your knowledge and let me know your experience while solving classification problems in comments below.

Question from past comments: How does a tree decide where to split ? How is Gini index calculated and how classification tree picks a variable to split the data set based on Gini index and Entropy.

Ans: Let me try and explain this in a simple way. Assume we have a data set with 3 variables: Target(0,1), Sex(M/F), Age (values between 18-65). Target is the dependent variable. The data set has 20 rows.

You run CART algorithm. How will it decide where to split ? CART uses Gini Index to decide the split. Since here, we have only two predictors, the tree will decide first variable like this:

For Sex Variable: First it will check, how many males are there in the data. Let's say 8. Out of all these males, how many have target as 1. lets say (3/8) Out of all these males, how many have target as 0, lets say (5/8)

Gini Index Formula = $1 - ((3/8)^2 + (5/8)^2)$ = some value (let's say 0.658)

Then it will check, how many females are there in the data. We get $20 - 8 = 12$ Out of all these females, how many have target as 1, lets say (4/12) Out of all these females, how many have target as 0, lets say (8/12)

Gini Index Formula = $1 - ((4/12)^2 + (8/12)^2)$ = some value (let's say 0.758)

Now, we'll weight the index values and sum them: $8/20 * 0.658 + 12/20 * 0.758$ = some value (X)

For Age variable, we'll follow the same procedure. Let's say, the tree identifies Age ≥ 23 as splitting value. The splitting will be like this:

First it will check, how many rows are present in the age variable which has value ≥ 23 . let's say 13 Out of all these people, how many have target as 1, lets say (4/13) Out of all these people, how many have target as 0, lets say (9/13)

Gini Index Formula = $1 - ((4/13)^2 + (9/13)^2)$ = somevalue1

Then it will check, how many rows are present in the age variable which has value < 23 . So, we have $20 - 13 = 7$ Out of all these people, how many have target as 1, lets say (2/7) Out of all these people, how many have target as 0, lets say (5/7)

Gini Index Formula = $1 - ((2/7)^2 + (5/7)^2)$ = somevalue2

Lower value of Gini index is used for splitting. Therefore in our case, if somevalue2 $<$ somevalue1, the tree will split at Age variable, otherwise it will split at Gender variable.