# Text Classification- Random Forests

Promothesh Chatterjee*

## Random Forests

Random forest uses multiple decision trees and combines their output to improve the model. Say, you want to buy a particular stock for investment, since you are not very sure about it's prospects, you ask 10 financial advisors about it. Different advisors ask different questions to understand your thought process, and then, based on your responses 8 of them said that the stock has great prospects given your investment goals. Since the majority is in favor, you decide to buy the stock. This is an example of ensemble method.

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.

### How does Random Forest work?

Random Forest uses a bagging (Bootstrap Aggregating) algorithm for generating random samples. Basically, Random Forest allows each individual tree to randomly sample from the dataset with replacement, resulting in different trees. For instance, if our data was [1, 2, 3, 4, 5, 6], random samples generate might be [1, 1, 2, 5, 5, 6]. Repetition might occur due to sampling with replacement.

In text data, Random Forest takes a DTM (with n documents and p terms), and uses it to generate a new DTM by sampling cases at random with replacement. By sampling with replacement, some observations may be sampled multiple times and some observations may never be sampled. This essentially treats your data as a population of interest. The document rows that are left out from sampling are known as Out of Bag(OOB) samples. The model trains on the new DTM. The Out of Bag sample is used to determine unbiased estimate of the error.

Similarly, out of p columns, x columns are selected randomly at each node in the DTM. Typically, x= sqrt(p) is used for classification tree. Many trees are generated (each on a randomly generated sample of the data) and the final prediction is obtained by averaging or voting. When the algorithm considers an individual decision tree, the split in the tree is based on a random selection of predictors. Random forests forcibly exclude some predictor variables when building individual trees. This is done to prevent a particularly strong predictor from dominating all the trees. Thus, individual decision trees will not bear strong correlations with each other, making the average predictions more reliable.

### Implementing Random Forest in R

```
library(tidyverse)
library(e1071)
library(caret)
library(quanteda)
library(irlba)
library(randomForest)

# Load up the .CSV data and explore in RStudio.
hotel_raw <- read_csv("Data/hotel-reviews.csv")
```

```
set.seed(1234)
hotel_raw<-hotel_raw[sample(nrow(hotel_raw), 5000), ]# take a small sample
summary(hotel_raw)
```

```
##    User_ID          Description        Browser_Used       Device_Used
##  Length:5000        Length:5000        Length:5000        Length:5000
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##  Is_Response
##  Length:5000
##  Class :character
##  Mode  :character
```

```
# Check for missing data
sum(!complete.cases(hotel_raw))
```

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

Since we have three relevant variables that are coded as character variables, let's convert them to Factor as they would relevant for future analysis.

```
hotel_raw$Is_Response <- as.factor(hotel_raw$Is_Response) # tells whether happy or not happy
hotel_raw$Device_Used <- as.factor(hotel_raw$Device_Used)
hotel_raw$Browser_Used <- as.factor(hotel_raw$Browser_Used)
```

First, let's take a look at distribution of the class labels (i.e., what proportion of consumers are happy vs. not happy).

```
hotel_raw %>%
count(Is_Response)%>%
  mutate(freq=n/sum(n))
```

```
## # A tibble: 2 x 3
##   Is_Response     n  freq
##   <fct>       <int> <dbl>
## 1 happy        3471 0.694
## 2 not happy    1529 0.306
```

Before looking at the text reviews, let's just examine the distribution of the length of the reviews.

```
hotel_raw<-hotel_raw %>%
  mutate(ReviewLength=nchar(Description))
```

Now let's check if review length differs based on whether consumers are happy or not with the hotel. We might suspect that people who are not happy might post longer reviews. Without any statistical test, our conjecture seems to be correct.

```
hotel_raw%>%
  group_by(Is_Response) %>%
  summarise(AvLength=mean(ReviewLength))
```

```
## # A tibble: 2 x 2
##   Is_Response AvLength
##   <fct>          <dbl>
## 1 happy           805.
## 2 not happy      1062.
```

Before we proceed, let's think about cross-validation (i.e.,create training and test sets for assessing how the results of a statistical analysis will generalize to an independent data set).(For the idea behind these check out: https://www.codecademy.com/articles/training-set-vs-validation-set-vs-test-set and https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f). In a true project we typically would do a three-way split: training, validation, and test.

Our data does have some class imbalance (not serious) but it is non-trivial, so let's use the caret package to do a random stratified train/test split so that we can have proper proportion of happy/not happy class labels.

For a wonderful article on class imbalance read: https://www.analyticsvidhya.com/blog/2017/03/imbalanced-classification-problem/

Let's now create a 70%/30% stratified split. We also set the random seed for reproducibility.

```
library(caret)
set.seed(12345)
hotel_record <- createDataPartition(hotel_raw$Is_Response, times = 1,
                                    p = 0.7, list = FALSE)
hotel_train <- hotel_raw[hotel_record,]
hotel_test <- hotel_raw[-hotel_record,]
```

So we have created the train and test datasets, let's make sure that the proportion of happy/not-happy is similar to the original dataset.

```
hotel_train %>%
count(Is_Response)%>%
  mutate(freq=n/sum(n))
```

```
## # A tibble: 2 x 3
##   Is_Response     n  freq
```

```
##   <fct>       <int> <dbl>
## 1 happy        2430 0.694
## 2 not happy    1071 0.306
```

```
hotel_test %>%
count(Is_Response)%>%
  mutate(freq=n/sum(n))
```

```
## # A tibble: 2 x 3
##   Is_Response     n  freq
##   <fct>       <int> <dbl>
## 1 happy        1041 0.694
## 2 not happy     458 0.306
```

We can see that both train and test datasets have similar proportion of class labels.

**Text Preprocessing** Let's tokenize and preprocess the reviews from the hotel dataset and take a look at one of the reviews.

```
library(quanteda)
hotel_train_token2 <- tokens(hotel_train$Description, what = "word",
                        remove_numbers = TRUE, remove_punct = TRUE,
                        remove_symbols = TRUE) %>%
                        tokens_tolower()
```

Let's also remove the stopwords and perform word stemming, and create document frequency matrix model with TFIDF.

```
hotel_train_dfm<-hotel_train_token2 %>%
  tokens_remove(stopwords(source = "smart")) %>%
  tokens_wordstem() %>%
  dfm() %>%
    dfm_trim( min_termfreq = 10, min_docfreq = 2) %>%
    dfm_tfidf()
```

Now let's append the feature data frame with class labels.

```
hotel_train_dfm1 <- convert(hotel_train_dfm, to ="data.frame")
hotel_train_df <- cbind(Label = hotel_train$Is_Response, data.frame(hotel_train_dfm1))
```

**Stratified K-Fold Cross-validation** K-Fold Cross-validation and Stratified K-Fold Cross-validation are both techniques used in machine learning to assess the performance of a model and ensure that it can generalize well to unseen data.

In **K-Fold Cross-validation**, the dataset is divided into `K` equal (or as close to equal as possible) sized subsets or "folds". The model is then trained and validated `K` times, each time using `K-1` folds for training and the remaining fold for validation. For instance, in the first iteration, the first fold might be used as the validation set and the remaining folds (2, 3, 4, and 5 in this case) for training. In the second iteration, the second fold could be used for validation, and so on. This process ensures that each fold is used for validation exactly once. After each iteration, the model's performance is evaluated on the validation set using an appropriate performance metric. Once all `K` iterations are complete, the performance metrics from each iteration are aggregated to give a single estimate of the model's performance. After the cross-validation process, the model is typically retrained on the entire dataset before making predictions on new, unseen data.

**Stratified K-Fold Cross-validation** is an enhancement over K-Fold Cross-validation and is particularly useful when the dataset has an imbalanced class distribution. In Stratified K-Fold Cross-validation, the division of data into `K` folds is done in such a way that each fold has roughly the same proportion of samples of each target class as the complete set. For example, if the dataset has 20% positive and 80% negative samples, each fold should maintain this ratio. This is crucial for imbalanced datasets, where one class is significantly less frequent than the others.

Like in K-Fold Cross-validation, the model is trained `K` times, each time on `K-1` folds and validated on the remaining fold. The performance metrics from each iteration are aggregated to estimate the model's performance. The model can then be retrained on the entire dataset for predictions on new data.

By preserving the original class distribution in each fold, Stratified K-Fold Cross-validation provides a more robust estimate of the model's performance on unseen data. Both K-Fold and Stratified K-Fold Cross-validation make efficient use of data by using each data point for both training and validation, and provide a measure of how sensitive the model is to the choice of the training dataset. This can help in detecting overfitting, where a model learns the training data too well and performs poorly on unseen data.

We will use library caret to create stratified folds for 10-fold cross validation repeated 2 times (i.e., create 20 random stratified samples). To assess the effectiveness of our model, The error estimation is averaged across all trials to get a measure of the model effectiveness.

```
# Use caret to create stratified folds for 10-fold cross validation repeated
# 2 times (i.e., create 20 random stratified samples)
set.seed(48743)
cv.folds <- createMultiFolds(hotel_train$Is_Response, k = 10, times = 2)
# basically this will create 20 random stratified samples

cv.cntrl <- trainControl(method = "repeatedcv", number = 10,
                         repeats = 2, index = cv.folds)
```

Our data frame is non-trivial in size. As such, CV runs will take quite a long time to run. To cut down on total execution time, use the doSNOW package to allow for multi-core training in parallel. As our data is non-trivial in size at this point, use a single decision tree algorithm as our first model. We will check Random Forest later.

Library caret has train function that allows us to fit different machine learning models such as random forest or XGboost etc. Right now we are asking it to train a single decision-tree by the argument *rpart*. In the train function, tuneLength=7 parameter tells the algorithm to try several seven different configurations for rpart and find out which one of those seven configurations works the best. This is known as hyperparameter tuning.

```
library(doSNOW)

# Time the code execution
start.time <- Sys.time()

# Create a cluster to work on 4 logical cores.
cl <- makeCluster(3, type = "SOCK") # Simplistically, we are asking computer to run
# 3 instances of Rstudio for processing. I have total of 4 cores in my computer
registerDoSNOW(cl) # register the instance

rpart.cv.1 <- train(Label ~ ., data = hotel_train_df, method = "rpart",
                    trControl = cv.cntrl, tuneLength = 7)


# Processing is done, stop cluster.
stopCluster(cl)

# Total time of execution
total.time <- Sys.time() - start.time
total.time

# Check out our results.
rpart.cv.1
```

```
Time difference of 13.81791 mins
CART

 3501 samples
10651 predictors
    2 classes: 'happy', 'not happy'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 2 times)
Summary of sample sizes: 3151, 3151, 3151, 3151, 3151, 3151, ...
Resampling results across tuning parameters:

  cp           Accuracy   Kappa
  0.004668534  0.7891998  0.46763551
  0.005042017  0.7863423  0.45473284
  0.005602241  0.7847709  0.44618334
  0.006535948  0.7854851  0.44690791
  0.009337068  0.7716382  0.37779379
  0.013071895  0.7550663  0.30509996
  0.052598817  0.6995165  0.02998128

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was cp = 0.004668534.
```

The best accuracy is 78.9%, which is ok but not great. And look at the humongous amount of time it took to run the model.

**Estimating a Random Forest Model**

Let's use the train() function from caret for the rf method (random forest). For computational ease let's estimate the forest with 100 trees (instead of the default is 500 trees). If we use the original DTM without compressing, it will take forever on my computer (I know the model was still running after 20 hours when I finally stopped it). What we will do is use a LSA truncated dataset (To estimate the random forest model, I took the original csv file (taken without sampling) and then truncated it with LSA as opposed to the sampled file that I used for decision tree. Beware that it took over 6 and half hours to run). We will use the R package irlba which works much better with larger datasets as opposed to the LSA package that we had used previously. Essentially, we will use the truncated document matrix which represents the documents with higher level concepts and reduced dimensions.

```r
# Perform SVD. Specifically, reduce dimensionality down to 300 columns
# for our latent semantic analysis (LSA).
library(irlba)
train.lsa <- irlba(t(hotel_train_dfm), nv = 300, maxit = 600)
```

```
# Take a look at the new feature data up close.
View(train.lsa$v)
```

Let's prepare the LSA truncated training dataset for running random forest algorithm.

```
train_svd <- data.frame(Label = hotel_train$Is_Response, train.lsa$v)
# Time the code execution
start.time <- Sys.time()

# Create a cluster to work on 4 logical cores.
cl <- makeCluster(3, type = "SOCK")
registerDoSNOW(cl) # register the instance

rf1 <- train(Label ~ ., data = train_svd,
                     method = "rf",
                     ntree = 100,
                     trControl = cv.cntrl, tuneLength = 7,
                     importance = TRUE)

# Processing is done, stop cluster.
stopCluster(cl)

# Total time of execution
total.time <- Sys.time() - start.time
total.time
```

Let's look at the output

```
rf1
```

```
Random Forest

27253 samples
  300 predictor
    2 classes: 'happy', 'not happy'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 2 times)
Summary of sample sizes: 24527, 24528, 24529, 24527, 24527, 24529, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
    2   0.7148754  0.1711615
   51   0.8297065  0.5827657
  101   0.8302387  0.5870821
  151   0.8302754  0.5879077
  200   0.8301474  0.5883451
  250   0.8287709  0.5848035
  300   0.8270285  0.5815168

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 151.
```

You can look at the structure of the output:

```
str(rf1, max.level = 1)
```

```
 $ method      : chr "rf"
 $ modelInfo   :List of 15
 $ modelType   : chr "Classification"
 $ results     :'data.frame':   7 obs. of  5 variables:
 $ pred        : NULL
 $ bestTune    :'data.frame':   1 obs. of  1 variable:
 $ call        : language train.formula(form = Label ~ ., data = train_svd, method = "rf", ntree
= 100,      trControl = cv.cntrl, tuneLeng| __truncated__
 $ dots        :List of 2
 $ metric      : chr "Accuracy"
 $ control     :List of 27
 $ finalModel  :List of 23
  ..- attr(*, "class")= chr "randomForest"
 $ preProcess  : NULL
 $ trainingData:'data.frame':   27253 obs. of  301 variables:
 $ resample    :'data.frame':   20 obs. of  3 variables:
 $ resampledCM :'data.frame':   140 obs. of  6 variables:
 $ perfNames   : chr [1:2] "Accuracy" "Kappa"
 $ maximize    : logi TRUE
 $ yLimits     : NULL
 $ times       :List of 3
 $ levels      : chr [1:2] "happy" "not happy"
  ..- attr(*, "ordered")= logi FALSE
 $ terms       :Classes 'terms', 'formula'  language Label ~ X1 + X2 + X3 + X4 + X5 + X6 + X7 +
X8 + X9 + X10 + X11 + X12 +      X13 + X14 + X15 + X16 + X17 + X18 + X| __truncated__ ...
  .. ..- attr(*, "variables")= language list(Label, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10,
X11, X12, X13, X14,      X15, X16, X17, X18, X19, X20, X21, | __truncated__ ...
  .. ..- attr(*, "factors")= int [1:301, 1:300] 0 1 0 0 0 0 0 0 0 0 ...
  .. .. ..- attr(*, "dimnames")=List of 2
  .. ..- attr(*, "term.labels")= chr [1:300] "X1" "X2" "X3" "X4" ...
  .. ..- attr(*, "order")= int [1:300] 1 1 1 1 1 1 1 1 1 1 ...
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(Label, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11,
X12, X13, X14,      X15, X16, X17, X18, X19, X20, X21, | __truncated__ ...
  .. ..- attr(*, "dataClasses")= Named chr [1:301] "factor" "numeric" "numeric" "numeric" ...
  .. .. ..- attr(*, "names")= chr [1:301] "Label" "X1" "X2" "X3" ...
 $ coefnames   : chr [1:300] "X1" "X2" "X3" "X4" ...
 $ xlevels     : Named list()
 - attr(*, "class")= chr [1:2] "train" "train.formula"
```

The model output is always stored finalModel element of the list. To access it we can simply call rf$finalModel.

```
rf1$finalModel
```

```
Call:
 randomForest(x = x, y = y, ntree = 100, mtry = param$mtry, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 151

        OOB estimate of  error rate: 17.34%
Confusion matrix:
          happy not happy class.error
happy     16949      1616  0.08704552
not happy  3109      5579  0.35784991
```

```
# Let's drill-down on the results.
confusionMatrix(train_svd$Label, rf1$finalModel$predicted)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  happy not happy
  happy     16949      1616
  not happy  3109      5579

               Accuracy : 0.8266
                 95% CI : (0.8221, 0.8311)
    No Information Rate : 0.736
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5817

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.8450
            Specificity : 0.7754
         Pos Pred Value : 0.9130
         Neg Pred Value : 0.6422
             Prevalence : 0.7360
         Detection Rate : 0.6219
   Detection Prevalence : 0.6812
      Balanced Accuracy : 0.8102

       'Positive' Class : happy
```

This tells us some important things:

a) We used 100 trees

b) The out-of-bag (OOB) error rate is 17.34%. Here we get an OOB estimate of the error rate of 17.34%. This means for test observations, the model misclassifies whether the reviewer was happy or not happy 17.34% of the time.

c) The confusion matrix compares the predictions to the actual known outcomes. The error rate for happy reviewers is much smaller than the unhappy people.

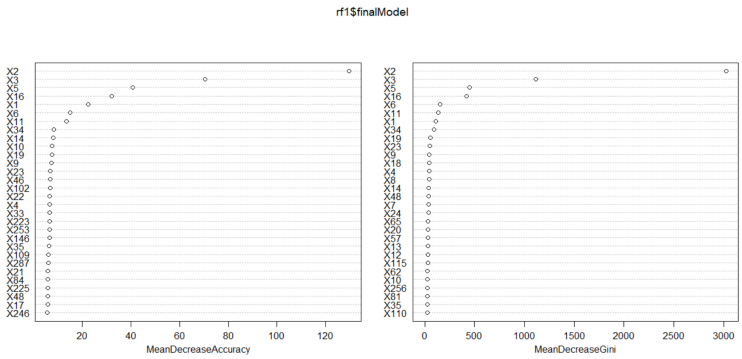
We improved the prediction accuracy a little from the single decision tree. A couple of things to note:

1) May be the smaller sample helped the prediction accuracy of the decision tree by removing noise.

2) We randomly took 300 dimensions we computing the LSA, taking different dimensions may alter the results.

3) We add some other features to the model to see if it improves the predictions. For instance, we can add in a feature we engineered previously (reference to class 1) the length of the review, to see if it improves things. I am not running this but you can try at home. You can also try adding n-grams.

```r
library(randomForest)
varImpPlot(rf1$finalModel)
```




**How important are the different features?**

As the output suggests, feature X2 had by far the most impact on the model and can be considered as the most important. Now what was X2? We really cannot tell as we had used the LSA truncated matrix where the dimensions are some mathematical entity that cannot be interpreted (and that's why the term 'blackbox models' is so prevalent for some machine learning techniques)

**Testing the RF Model** We've built what appears to be a moderately effective predictive model. Now let's verify the predictive ability of the model in the test holdout data. Let's first preprocess the test data.

```
hotel_test_token <- tokens(hotel_test$Description, what = "word",
                           remove_numbers = TRUE, remove_punct = TRUE,
                           remove_symbols = TRUE) %>%
                           tokens_tolower()

hotel_test_dfm<-hotel_test_token %>%
  tokens_remove(stopwords(source = "smart")) %>%
  tokens_wordstem() %>%
  dfm() %>%
    dfm_tfidf() %>%
    dfm_trim( min_termfreq = 10, min_docfreq = 2)
```

We need to ensure that our test data has the exact same format as our training data, we can use quanteda function dfm_match for that.

```
hotel_test_dfm <- dfm_match(hotel_test_dfm, featnames(hotel_train_dfm))
hotel_test_matrix <- as.matrix(hotel_test_dfm)
hotel_test_dfm
```

As with TF-IDF, we will need to project new data (e.g., the test data) into the SVD semantic space. The following code illustrates how to do this. This is based of "Learning Analytics in R with SNA, LSA, and MPIA by Fridolin Wild", I will be happy to share the book chapter if you are interested.

```
sigma.inverse <- 1 / train.lsa$d # taking the tranpose of the singular matrix is # same as calculating
u.transpose <- t(train.lsa$u)#transpose of the term matrix
```

```
test_svd <- t(sigma.inverse * u.transpose %*% t(hotel_test_dfm))
test_svd<-as.matrix(test_svd)
```

Lastly, we can now build the test data frame to feed into our trained machine learning model for predictions. Let's add Label.

```
test_svd <- data.frame(Label = hotel_test$Is_Response, test_svd)
```

Now we can make predictions on the test data set using our trained random forest.

```
preds <- predict(rf1, test_svd)
confusionMatrix(preds, test_svd$Label)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  happy not happy
  happy      7342      1324
  not happy   614      2399

               Accuracy : 0.8341
                 95% CI : (0.8272, 0.8408)
    No Information Rate : 0.6812
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5975

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9228
            Specificity : 0.6444
         Pos Pred Value : 0.8472
         Neg Pred Value : 0.7962
             Prevalence : 0.6812
         Detection Rate : 0.6286
   Detection Prevalence : 0.7420
      Balanced Accuracy : 0.7836

       'Positive' Class : happy
```

Overfitting is the idea that the model will perform much better on the training data as opposed to on the test dataset. So we do not find evidence of overfitting in the random forest model here.