Churn Reduction

Vidyut Rao

24th September 2018

Table of Contents

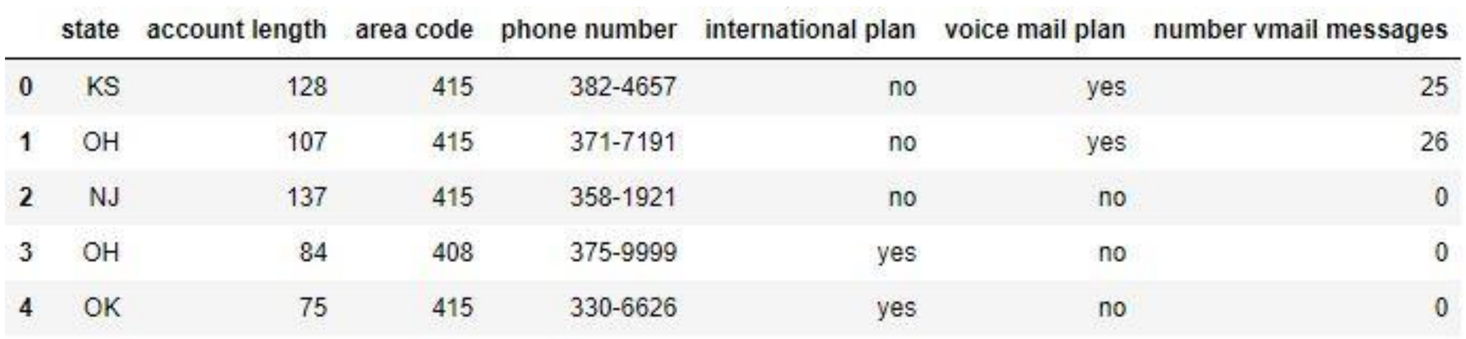
Problem Statement

Churn (loss of customers to competition) is a problem for companies because it is more expensive to acquire a new customer than to keep your existing one from leaving. This project is targeted at enabling churn reduction using analytics concepts.

Data

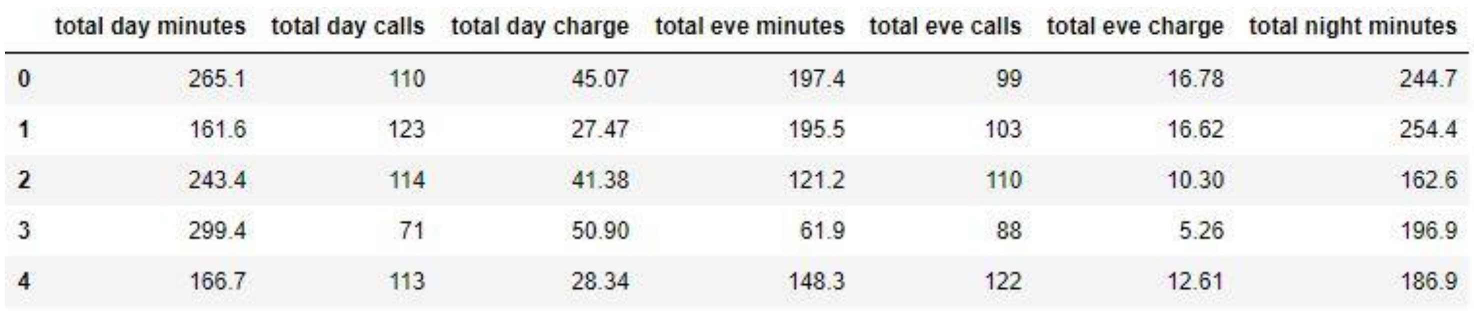
Before we proceed, let us familiarize ourselves with the given data as to what the predictors are and what the target variable is.

Predictors (1-7)



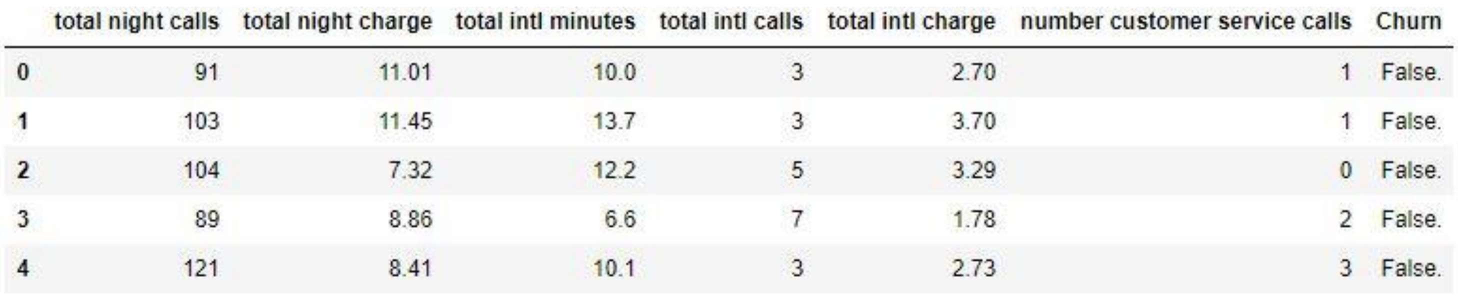
	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages
0	KS	128	415	382-4657	no	yes	25
1	OH	107	415	371-7191	no	yes	26
2	NJ	137	415	358-1921	no	no	0
3	OH	84	408	375-9999	yes	no	0
4	OK	75	415	330-6626	yes	no	0

Predictors (8-14)



	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	total eve charge	total night minutes
0	265.1	110	45.07	197.4	99	16.78	244.7
1	161.6	123	27.47	195.5	103	16.62	254.4
2	243.4	114	41.38	121.2	110	10.30	162.6
3	299.4	71	50.90	61.9	88	5.26	196.9
4	166.7	113	28.34	148.3	122	12.61	186.9

Predictors (14-20) and the target variable – *Churn*



	total night calls	total night charge	total intl minutes	total intl calls	total intl charge	number customer service calls	Churn
0	91	11.01	10.0	3	2.70	1	False.
1	103	11.45	13.7	3	3.70	1	False.
2	104	7.32	12.2	5	3.29	0	False.
3	89	8.86	6.6	7	1.78	2	False.
4	121	8.41	10.1	3	2.73	3	False.

Our dataset contains a mixed bag of binary, nominal and continuous variables. Also, this data has no missing values.

With the given data, we need to build a model that will predict whether a customer will churn or not. We will also use multiple methods to find out which predictors greatly influence the target variable, which variable are irrelevant to the modelling process and must be dropped and in turn how to manipulate the predictors accurately predict customer behavior.

Pre Processing and Exploratory Data Analysis

As the first step in any data science project, we need to explore the data for initial insights (and inconsistencies). The steps involved here include finding and imputing/eliminating missing values, analyzing outliers, analyzing the features for their strength in classifying the target variable (and eliminating multicollinearity) and scaling the features.

The primary objective of this task is to clean and smoothen the data as much as possible as better the data fed into the model, greater is the model's predictive power.

1. Preliminary Univariate Analysis

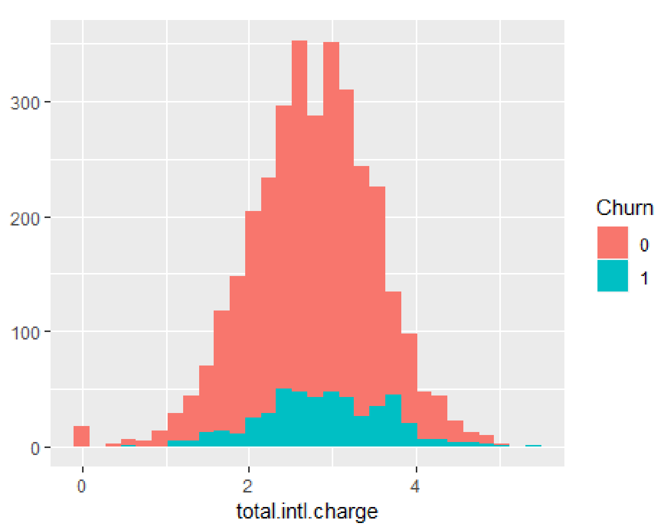
Most columns are already in their correct data type, the only exception being *Area Code*. We will need to convert it. Also, all categorical variables will have to be encoded into numeric levels.

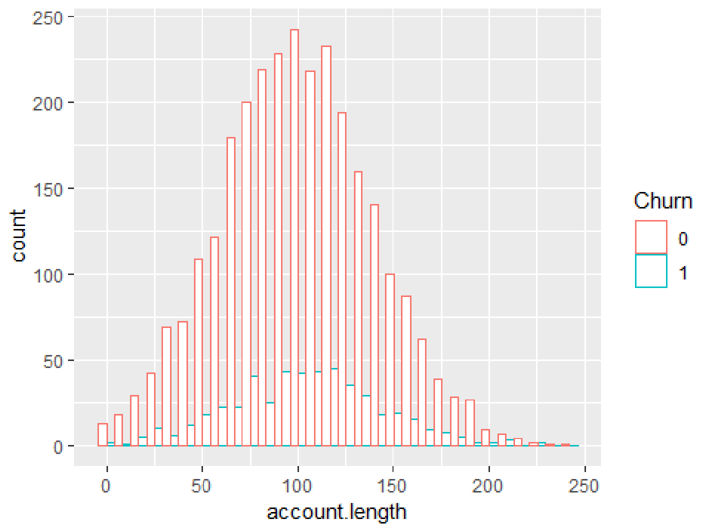
A final step will be to drop *Phone Number* as it is just a tag/ID with irrelevant information for prediction.

```
#Univariate Pre-Processing
df$area.code = as.factor(df$area.code)
df = subset(df, select = -c(phone.number))

#Encoding String Factors
for(i in 1:ncol(df)){
  if(class(df[,i]) == 'factor'){
    df[,i] = factor(df[,i],
labels=(1:length(levels(factor(df[,i])))))
  }
}
```

This data has no missing values. But we will perform traditional missing value treatment after outlier analysis.

All the numeric data is normally distributed with little skews either to the left or the right. This indicates the presence of Outliers, which we will analyze in greater detail and eliminate in the next section.



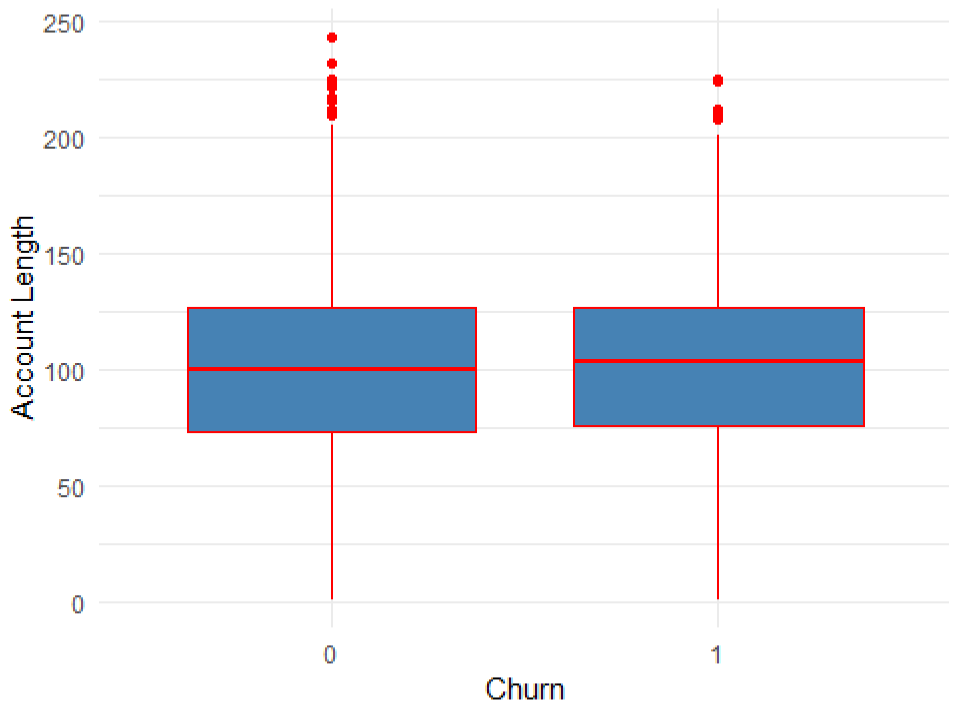


Distribution of *Total International Charge* and *Account Length*. As we can see the first graph is slightly skewed to the left while the second is slightly skewed to the right.

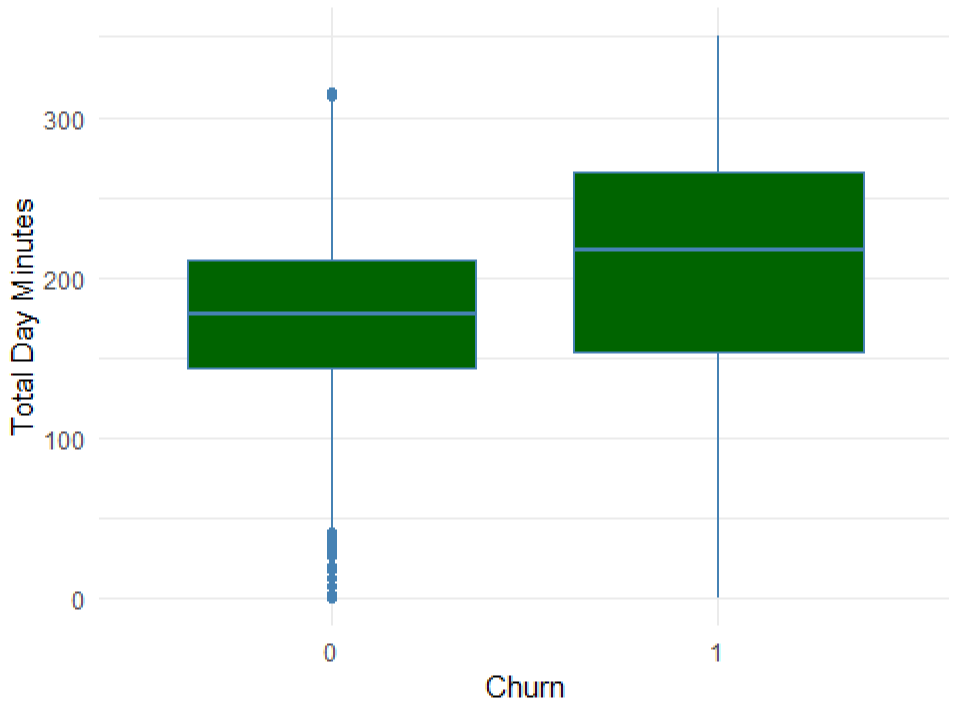
2. Outlier Analysis

The data needs to be inspected for extreme values that would skew and distort the data, resulting in noise that would yield a poor model. The best way to detect and remove outliers is by employing *Tukey's Boxplot Method*.

First, we will validate the presence of outliers using Boxplots.



Boxplot of *Account Length*. The red dots that lie outside the fences are outliers.



Boxplot of *Total Day Minutes*. The blue dots that lie outside the fences are outliers.

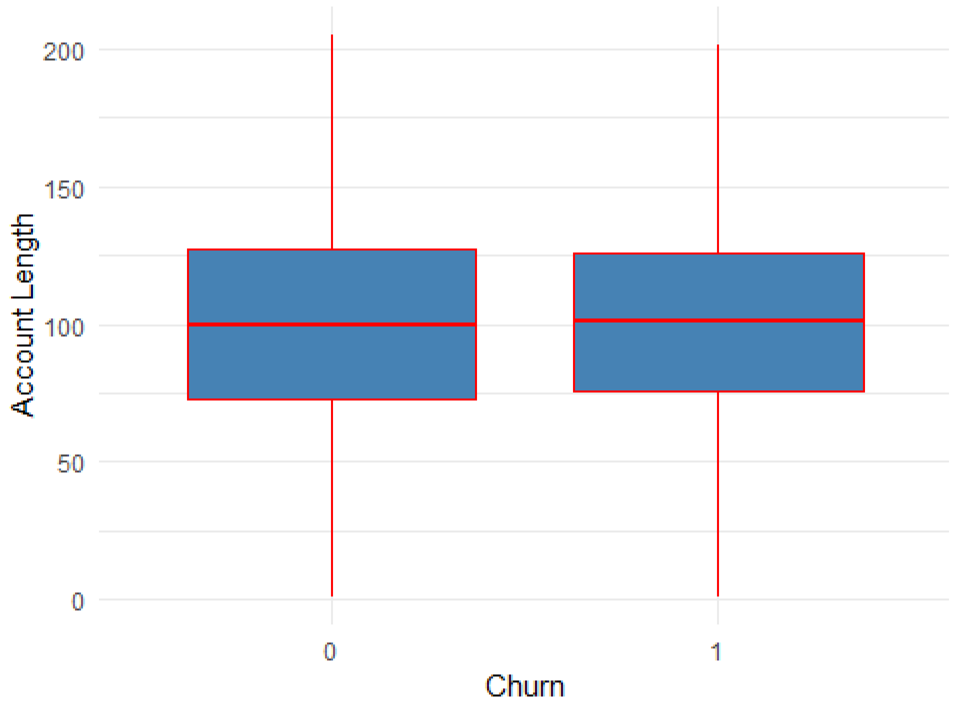
Outliers are detected using *Tukey's Boxplot Method*, the logic for which is showcased in the Python code below. In R, a the same logic is employed using the *boxplot.stats()$out*.

```
#Replacing Outliers with NA

for i in num_nm:
     print(i)
     q75, q25 = np.percentile(df.loc[:,i], [75 ,25])
     iqr = q75 - q25
     min = q25 - (iqr*1.5)
     max = q75 + (iqr*1.5)
     print(min)
     print(max)
     df.loc[df.loc[:,i] < min,i] = np.nan
     df.loc[df.loc[:,i] > max,i] = np.nan
```

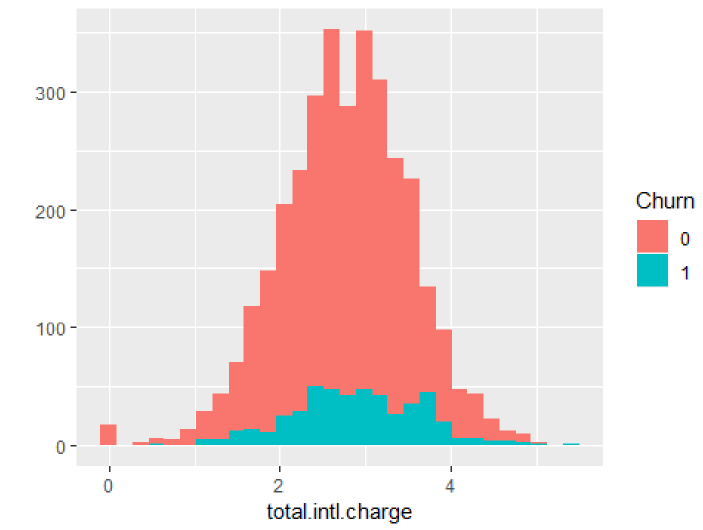
After this, missing values are generated which will have to imputed in one of three ways – Mean, Median and KNN imputation. To select the most accurate method, we will replace a known value with NA/NaN and impute it in each of the three ways to find which imputes the missing value with the most accurate value. I have repeated this across various numeric predictors to settle on the most accurate method.

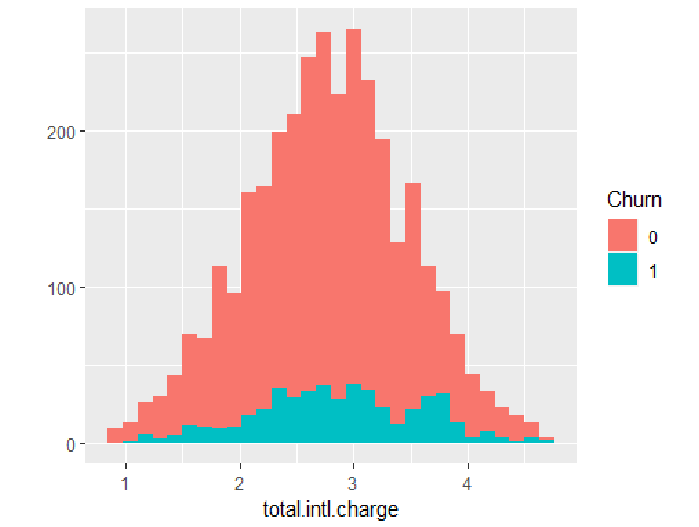
In R, I found that the Median method quite consistently provided the closest value. In Python, it was a tough call between Median and KNN, but I chose KNN imputation based on multiple trials.

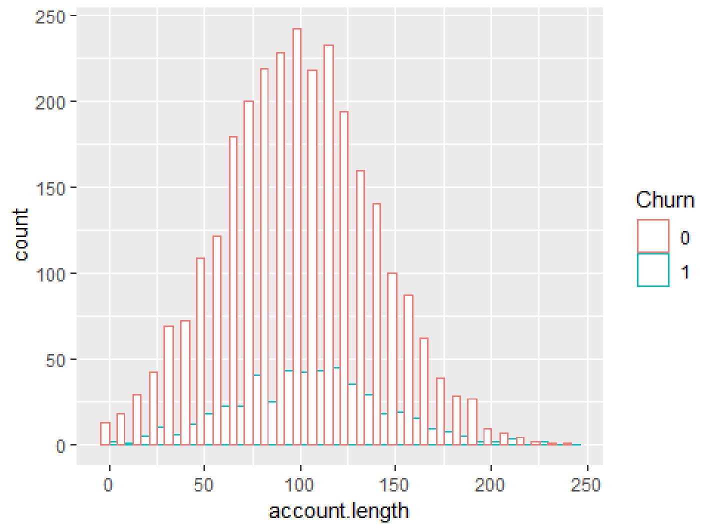
```
#Testing Best Method

#Please uncomment and comment as necessary to test
df_test['total day minutes'].loc[70]
#Original = 241.8
#With Mean = 179.92
#With Median = 179.4
#With KNN = 207.02
df_test['total day minutes'].loc[70] = np.nan
df_test['total day minutes'] = df_test['total day
minutes'].fillna(df_test['total day minutes'].mean())
df_test['total day minutes'] = df_test['total day
minutes'].fillna(df_test['total day minutes'].median())
df_test = pd.DataFrame(KNN(k = 3).complete(df_test), columns =
df_test.columns)
```

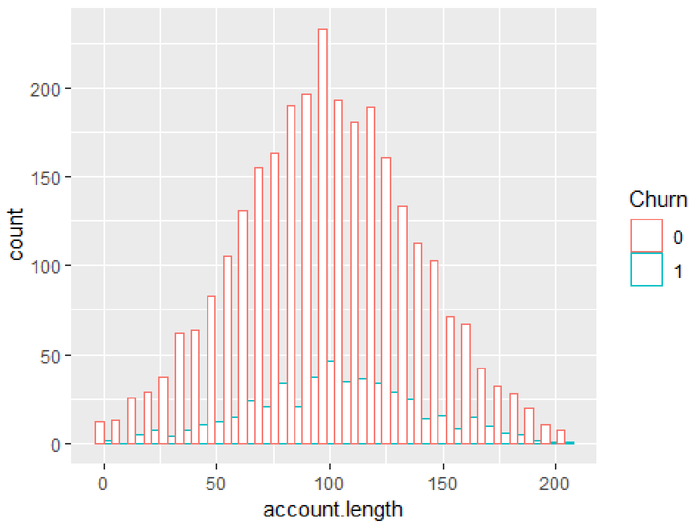


Boxplot after Outlier Treatment.









The distributions of *Total International Charge* and *Account Length* before and after Outliers were removed. It can be observed that skew that existed previously is now absent.

3. Feature Selection

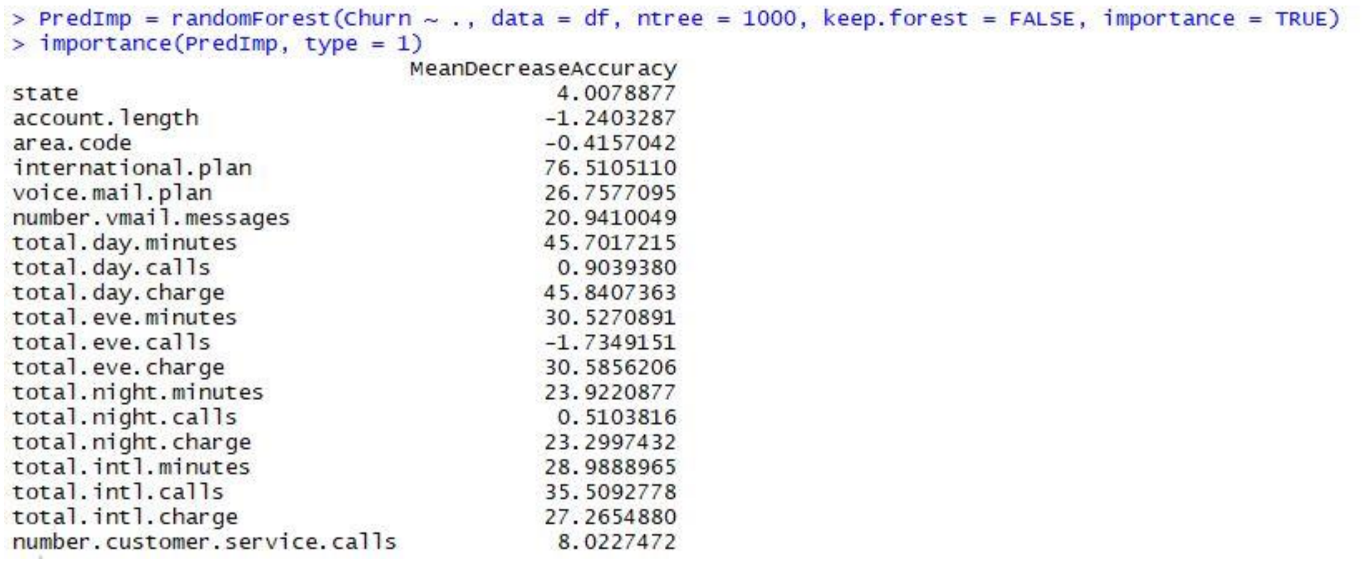
Feature selection is the process of choosing a subset of the most relevant and influential features for use in our model. The main objectives of feature selection are:

- Avoid over-fitting
- Eliminating the *curse of dimensionality*
- Eliminating multicollinearity
- Simplify the model and reduce the training time.

I will use Random Forests to ascertain the variable importance and create a Correlation Plot (R) or a Correlation Heat Map (Python) to find highly correlated predictors.

For categorical variables, Chi-Square Test will ascertain the variables that *Churn* is dependent on.

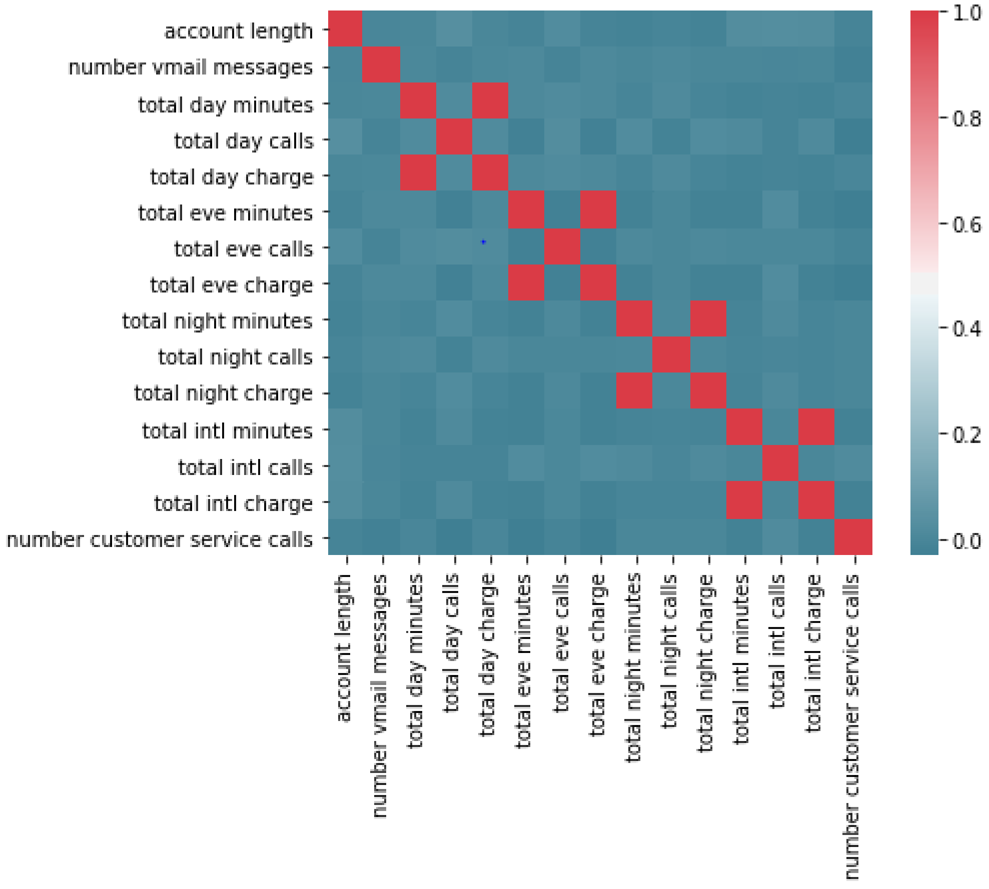
1. <u>Random Forest</u>



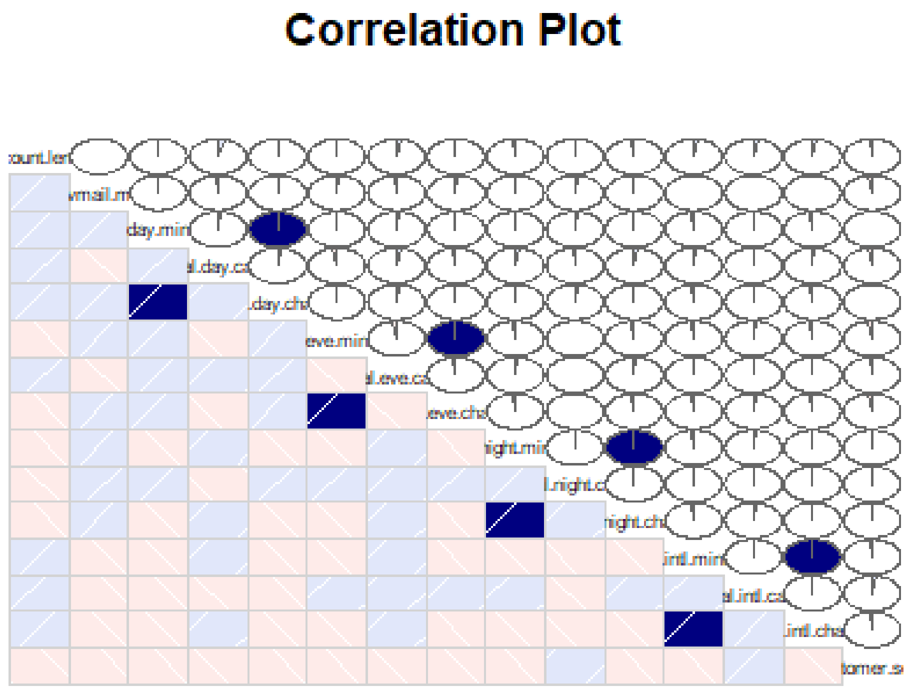
```
> PredImp = randomForest(Churn ~ ., data = df, ntree = 1000, keep.forest = FALSE, importance = TRUE)
> importance(PredImp, type = 1)
                              MeanDecreaseAccuracy
state                                    4.0078877
account.length                          -1.2403287
area.code                               -0.4157042
international.plan                      76.5105110
voice.mail.plan                         26.7577095
number.vmail.messages                   20.9410049
total.day.minutes                       45.7017215
total.day.calls                          0.9039380
total.day.charge                        45.8407363
total.eve.minutes                       30.5270891
total.eve.calls                         -1.7349151
total.eve.charge                        30.5856206
total.night.minutes                     23.9220877
total.night.calls                        0.5103816
total.night.charge                      23.2997432
total.intl.minutes                      28.9888965
total.intl.calls                        35.5092778
total.intl.charge                       27.2654880
number.customer.service.calls            8.0227472
```

As it can be seen, the variables *Account Length*, *Area Code*, *Total Day Calls*, *Total Night Calls* and *Total Eve Calls* have extremely low predictive power. We will proceed to drop them.

2. <u>Correlation</u>



Using this heat map created with the help of the s*eaborn* library in Python, we can see that all the *Charge* variables are very highly positively correlated with their corresponding *Total Minutes* variables. Hence we will drop all the *Charge* variables. The same analysis is done with the help of *corrgram* and *symnum* in R.



3. Chi- Square Test of Independence

To assess whether the target variable is dependent on the categorical variables, we can use the *Chi-Square Test of Independence*. This is a pairwise test that creates a contingency table for the target variable (which is categorical) against a categorical predictor. It then generates a Chi –Square value and a p-value. The p-value is used to reject or retain the NULL hypothesis that the target variable is independent of the predictor. A p-value < 0.05 rejects the NULL hypothesis and claims that the Alternate hypothesis is true – that the target is dependent on the predictor.

This test is done below.

```
> for(i in 1:4){
+   print(cat_name[i])
+   print(chisq.test(table(cat_dt$Churn,cat_dt[,i])))
+ }
[1] "state"

    Pearson's Chi-squared test

data:  table(cat_dt$Churn, cat_dt[, i])
X-squared = 83.044, df = 50, p-value = 0.002296

[1] "area.code"

    Pearson's Chi-squared test

data:  table(cat_dt$Churn, cat_dt[, i])
X-squared = 0.17754, df = 2, p-value = 0.9151

[1] "international.plan"

    Pearson's Chi-squared test with Yates' continuity correction

data:  table(cat_dt$Churn, cat_dt[, i])
X-squared = 222.57, df = 1, p-value < 2.2e-16

[1] "voice.mail.plan"

    Pearson's Chi-squared test with Yates' continuity correction

data:  table(cat_dt$Churn, cat_dt[, i])
X-squared = 34.132, df = 1, p-value = 5.151e-09

Warning message:
In chisq.test(table(cat_dt$Churn, cat_dt[, i])) :
  Chi-squared approximation may be incorrect
```

Here we can observe that the p-value is > 0.05 only for Churn vs Area Code, i.e., Churn is independent of Area Code.

4. Dimension Reduction

With the above knowledge, we can drop the variables as shown below:

```
df= subset(df, select = -c(total.night.calls, total.eve.calls, total.day.calls,
                              account.length, total.day.charge, total.eve.charge,
                              total.night.charge, total.intl.charge, area.code))
```

5. Feature Scaling

A last step in pre-processing the data is to normalize the numeric variable. The purpose of this is to ensure one variable doesn't greatly influence the model merely based on the broad range of its values. Using the logic given below, we can implement the same in R and Python

1. Normalization:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

```
#Normalization

numeric_index = sapply(df, is.numeric)
num_dt = df[,numeric_index]
num_nm = colnames(num_dt)

for(i in num_nm){
  print(i)
  df[,i] = (df[,i] - min(df[,i]))/(max(df[,i]) - min(df[,i]))
}
```

We can proceed with replicating the same steps for the Test data as well.

Modeling and Evaluation

Given that we need to classify *Churn* whose value is binary (Yes or No), our problem is one of ***Classification***. Classification, in the case of supervised machine learning, can be achieved through multiple methods: Decision Trees, Random Forest, Logistic Regression, KNN and Naïve Bayes. We will adopt each of these methods and evaluate the predictive power of each model using the validation techniques shown.

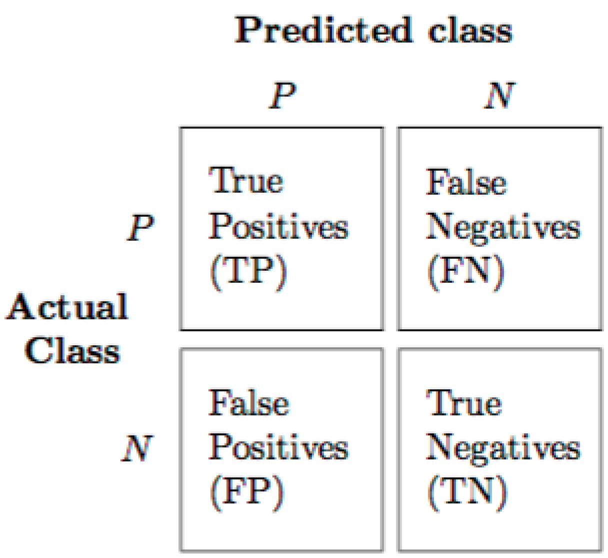
1. Splitting the data into train and test

This step is not required since the data is already given in separate ***Train.csv*** and ***Test.csv***. I would prefer getting the entire dataset , pre-process it and then split it using a stratified sampling method, so that the training data is more evenly distributed for the *Churn* = True and False (since we now have a training set which has far more Churn = False values). We will proceed regardless.

2. Evaluation

Before I build the classification models, we need to define evaluation metrics that will help validate the test results.

Confusion Matrix (Contingency Table)



		Predicted class	
		P	*N*
Actual Class	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

We create a contingency table of the generated predicted values (0 and 1) versus the actual values. The following metrics are most important in our case to evaluate the model :

1. Accuracy :

$$\mathrm{ACC} = \frac{\mathrm{TP}+\mathrm{TN}}{P+N} = \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$$

2. False Positive Rate:

False Positives or false alarms are called *Type I Errors*. These tell us the rate at which we predict that a Customer will move (Churn = Yes.) when they in fact didn't.

This is given by the expression:

$$\frac{FP}{FP + TN}$$

3. False Negative Rate:

False Negatives or misses are called *Type II Errors*. These tell us the rate at which we predict that a Customer will not move (Churn = False.) when they in fact did.

This is given by the expression:

$$\mathrm{FNR} = \frac{\mathrm{FN}}{P} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}}$$

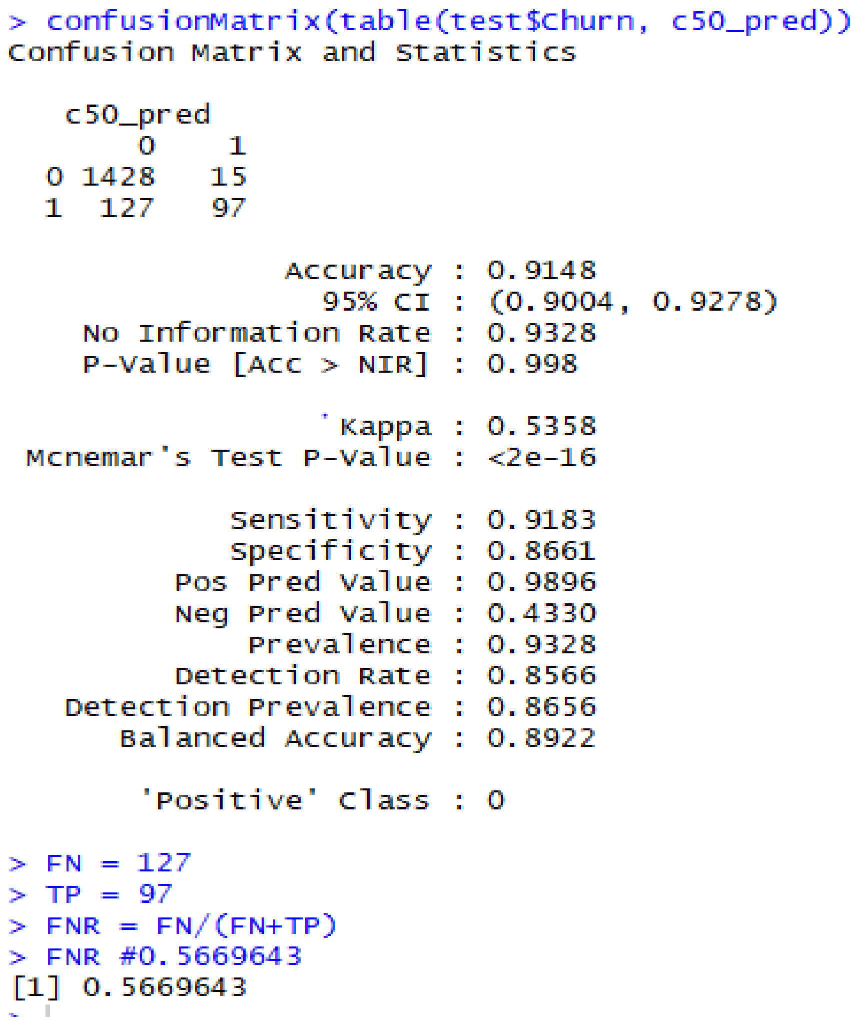
Our objective is to have a high Accuracy, low False Positive Rate and low False Negative Rate. We will see from the below models that first two hold good but we have a high FNR due to the fact that the number cases where Churn = False is much greater than when Churn = True. The aim of below models is to reduce FNR (even though FPR is arguably more important.)

3. Modeling

- Decision Trees

Decision Trees were built in R using the *C5.0* model from the *c50* library. This is a multiple split classifier that works based on Information Gain and uses rule based pruning.

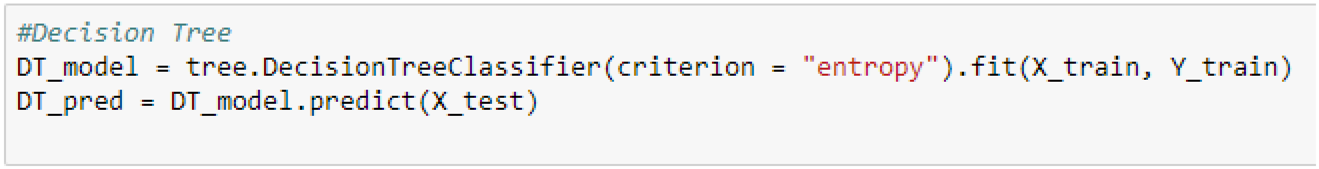
```
#Decision Tree Classifier
C50_model = C5.0(Churn ~., df, trials = 100, rules = TRUE)
summary(C50_model)
c50_pred = predict(C50_model, test[, -11], type = "class")
```



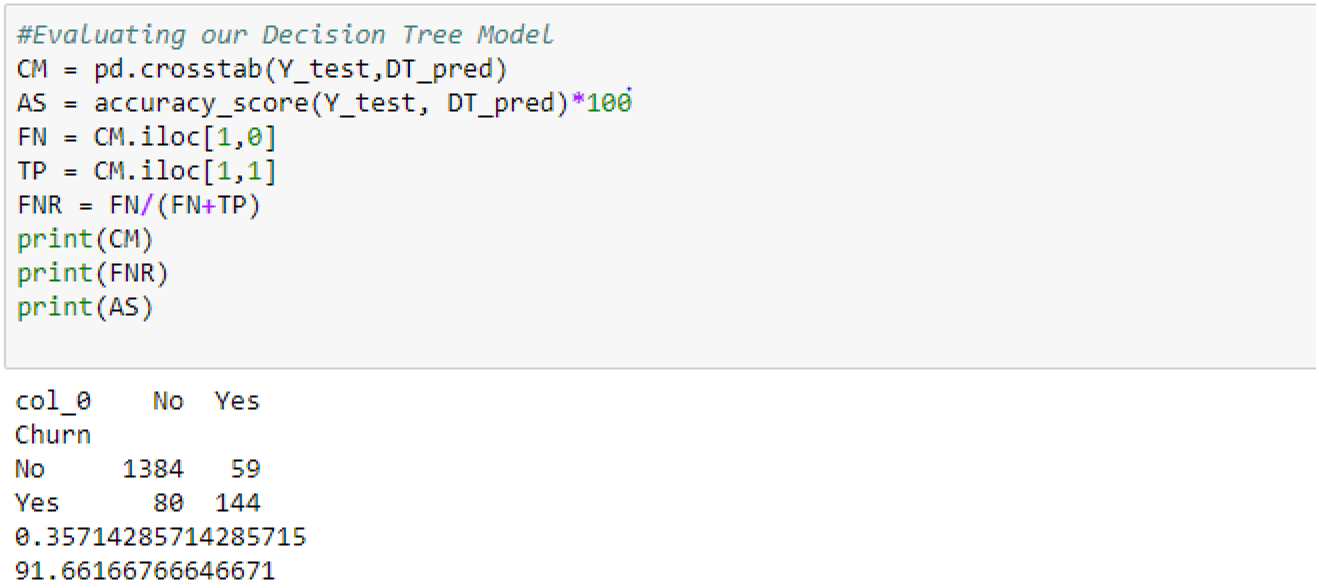
```
> confusionMatrix(table(test$Churn, c50_pred))
Confusion Matrix and Statistics

   c50_pred
       0    1
  0 1428   15
  1  127   97

               Accuracy : 0.9148
                 95% CI : (0.9004, 0.9278)
    No Information Rate : 0.9328
    P-Value [Acc > NIR] : 0.998

                  Kappa : 0.5358
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.9183
            Specificity : 0.8661
         Pos Pred Value : 0.9896
         Neg Pred Value : 0.4330
             Prevalence : 0.9328
         Detection Rate : 0.8566
   Detection Prevalence : 0.8656
      Balanced Accuracy : 0.8922

       'Positive' Class : 0

> FN = 127
> TP = 97
> FNR = FN/(FN+TP)
> FNR #0.5669643
[1] 0.5669643
```

Accuracy: 91.48
FNR: 0.566
FPR: 0.01

In Python, we used the *tree.DecisionTreeClassifier()* function from the *sklearn* library. It works on the same principle as the one done in R.



```
#Decision Tree
DT_model = tree.DecisionTreeClassifier(criterion = "entropy").fit(X_train, Y_train)
DT_pred = DT_model.predict(X_test)
```



```
#Evaluating our Decision Tree Model
CM = pd.crosstab(Y_test,DT_pred)
AS = accuracy_score(Y_test, DT_pred)*100
FN = CM.iloc[1,0]
TP = CM.iloc[1,1]
FNR = FN/(FN+TP)
print(CM)
print(FNR)
print(AS)
```

```
col_0     No  Yes
Churn
No      1384   59
Yes       80  144
0.35714285714285715
91.66166766646671
```

Accuracy: 91.66
FNR: 0.35714
FPR: 0.04

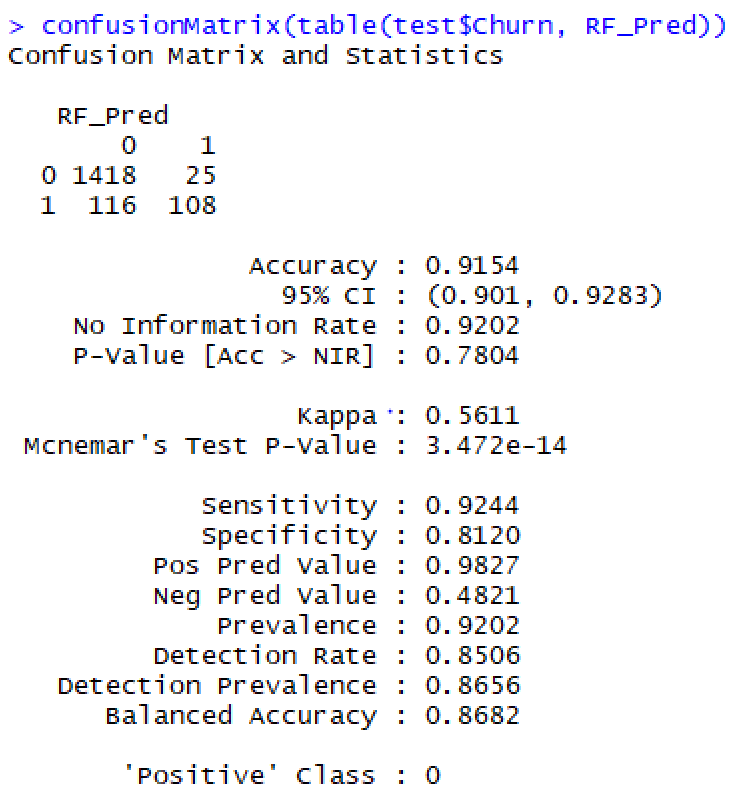
The decision tree models in both R and Python have performed reasonably well. The False Negative Rate in Python is the best of the models developed, as we'll see, at just 35%.

- Random Forests

Random Forests are an ensemble learning algorithm for both classification and regression which operate by constructing a multitude of decision trees and outputs the class that is the mode of the resulting classes. Each decision tree here randomly selects a subset of the total features present. Random Forests are better than a single decision tree as they do not overfit on the training data.

In R we use the *randomForest()* function present in the *randomForest* library.

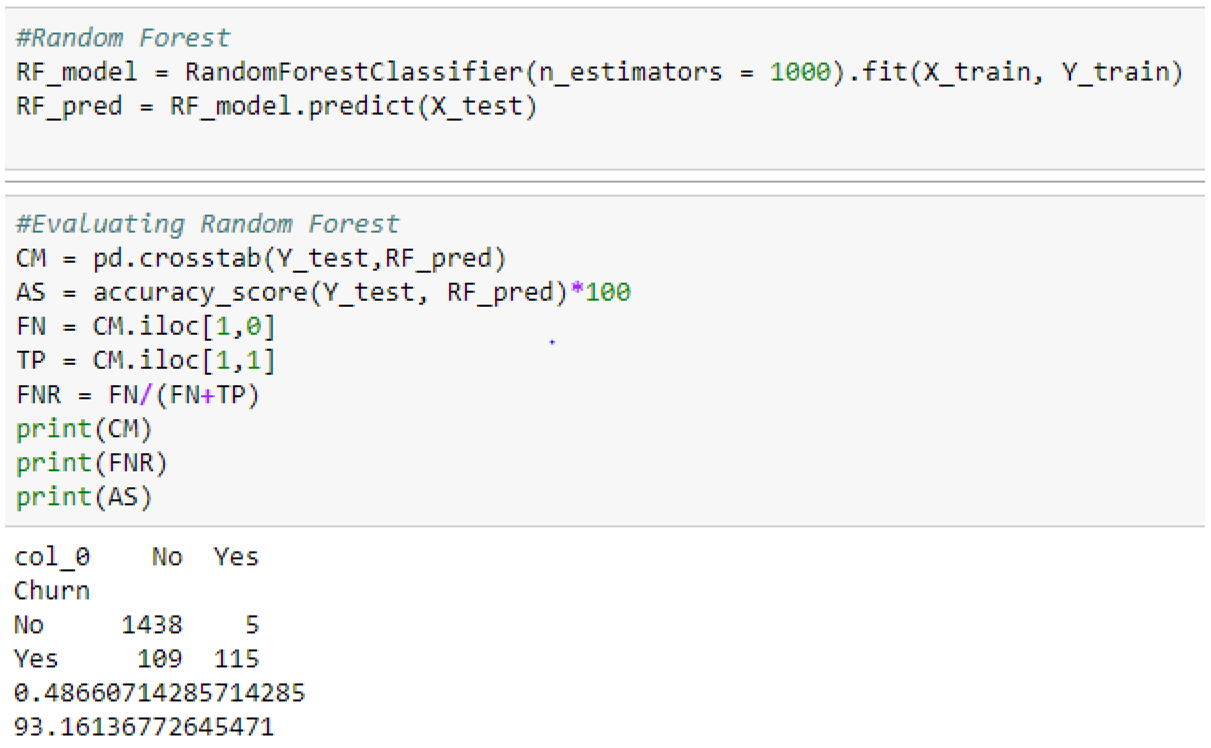
```
#Random Forest

RF_model = randomForest(Churn ~ ., df, importance = TRUE, ntree = 1000)
RF_Pred = predict(RF_model, test[,-11])
```



```
> confusionMatrix(table(test$Churn, RF_Pred))
Confusion Matrix and Statistics

   RF_Pred
       0    1
  0 1418   25
  1  116  108

               Accuracy : 0.9154
                 95% CI : (0.901, 0.9283)
    No Information Rate : 0.9202
    P-Value [Acc > NIR] : 0.7804

                  Kappa : 0.5611
 Mcnemar's Test P-Value : 3.472e-14

            Sensitivity : 0.9244
            Specificity : 0.8120
         Pos Pred Value : 0.9827
         Neg Pred Value : 0.4821
             Prevalence : 0.9202
         Detection Rate : 0.8506
   Detection Prevalence : 0.8656
      Balanced Accuracy : 0.8682

       'Positive' Class : 0
```

Accuracy: 91.54
FNR: 0.5089
FPR: 0.01

In Python we use the *RandomForestClassifier()* present *in sklearn.ensemble*. In both R and Python we have set the number of trees as 1000.



```
#Random Forest
RF_model = RandomForestClassifier(n_estimators = 1000).fit(X_train, Y_train)
RF_pred = RF_model.predict(X_test)
```

```
#Evaluating Random Forest
CM = pd.crosstab(Y_test,RF_pred)
AS = accuracy_score(Y_test, RF_pred)*100
FN = CM.iloc[1,0]
TP = CM.iloc[1,1]
FNR = FN/(FN+TP)
print(CM)
print(FNR)
print(AS)
```

```
col_0    No  Yes
Churn
No     1438    5
Yes     109  115
0.48660714285714285
93.16136772645471
```

Accuracy: 93.16
FNR: 0.4866
FPR: Almost 0

The Random Forest models obtained in R and Python also provided great results. The Random Forest model in Python produced the best Accuracy and lowest False Positive Rate of all, at 93.16% and almost 0 respectively.

- Logistic Regression

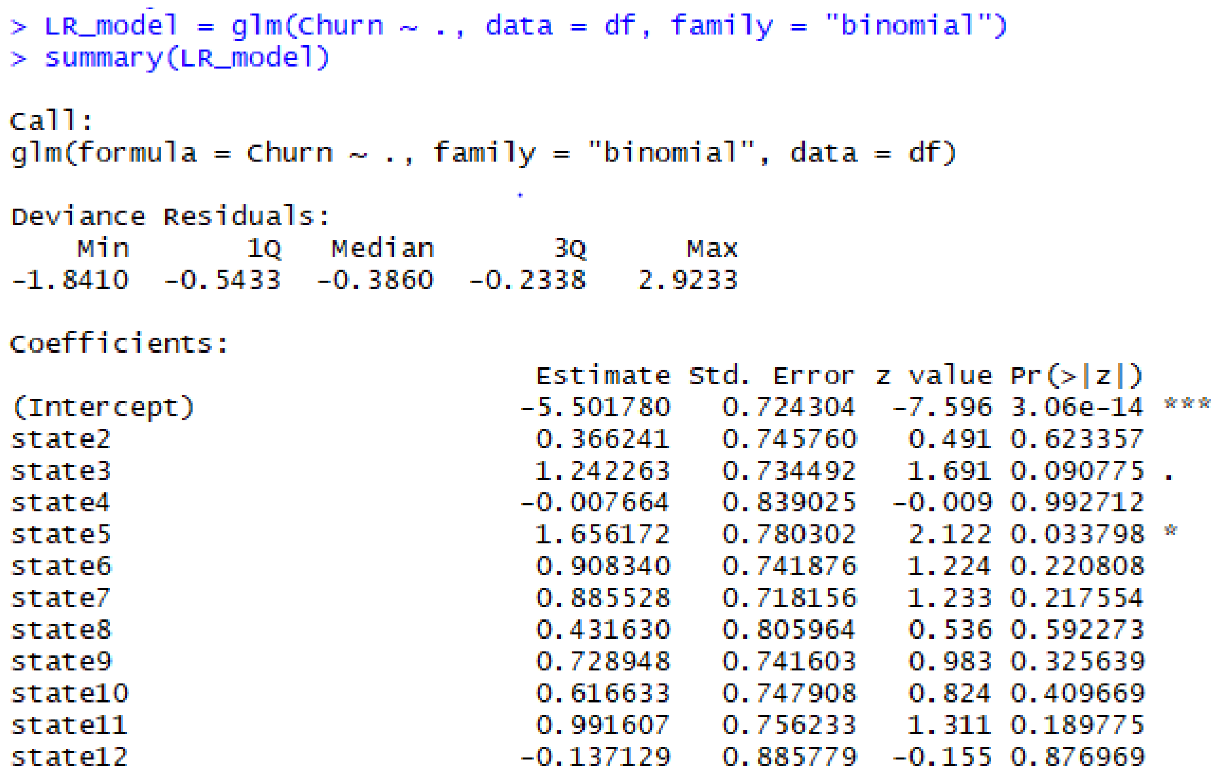
  Logistic regression is a classification model and the output of LR could be classification category (Male or Female) or probabilities. LR use logit function to calculate probabilities. Like linear regression, logistic regression builds coefficient equation that uses logit function to calculate probabilities.

Regression equation: Y = b0 + b1x1 + b2x2 + b3x3 + … bnxn = logit(p)

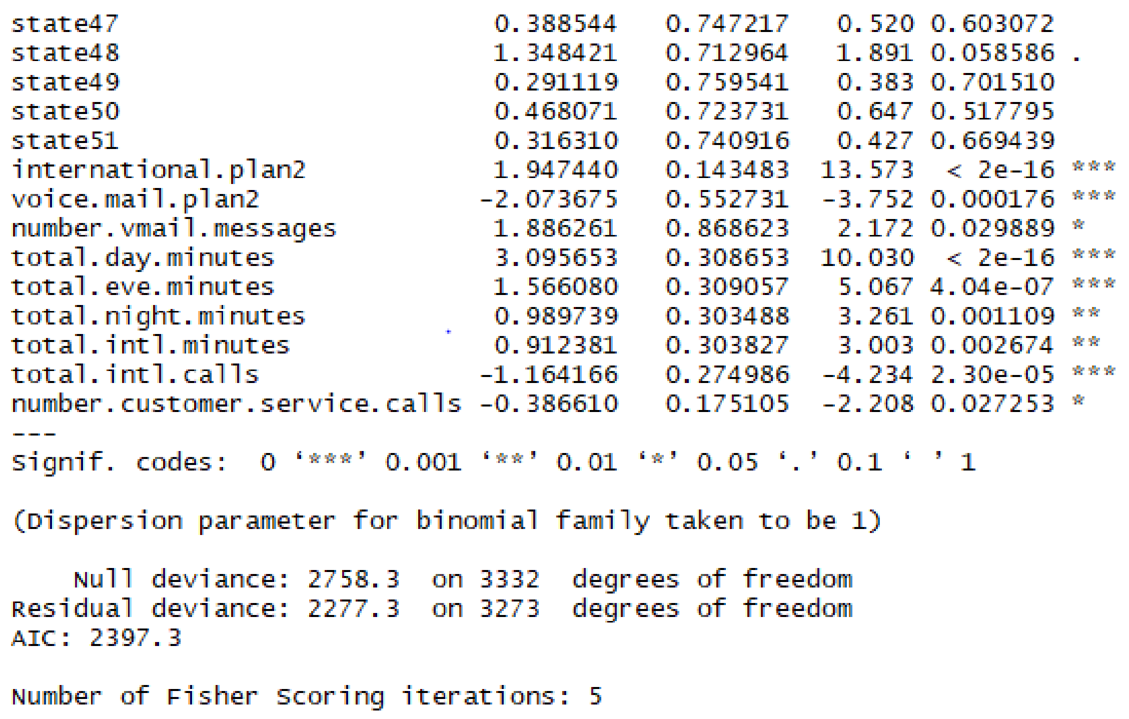
Logit function

$$\operatorname{logit} p = \ln \frac{p}{1-p} \quad \text{for } 0 < p < 1.$$

In R, we use the *glm()* function available in the *stats* library.

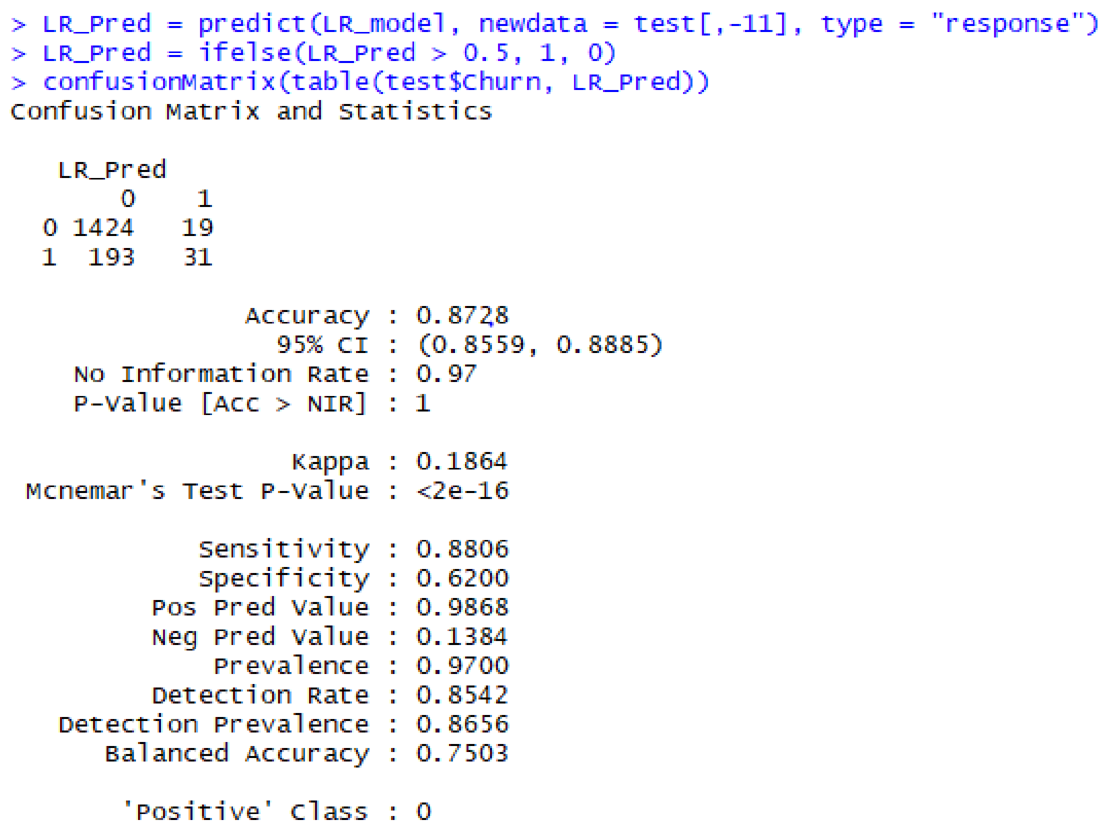


```
> LR_model = glm(Churn ~ ., data = df, family = "binomial")
> summary(LR_model)

Call:
glm(formula = Churn ~ ., family = "binomial", data = df)

Deviance Residuals: 
    Min       1Q   Median       3Q      Max  
-1.8410  -0.5433  -0.3860  -0.2338   2.9233  

Coefficients:
                               Estimate Std. Error z value Pr(>|z|)    
(Intercept)                   -5.501780   0.724304  -7.596 3.06e-14 ***
state2                         0.366241   0.745760   0.491 0.623357    
state3                         1.242263   0.734492   1.691 0.090775 .  
state4                        -0.007664   0.839025  -0.009 0.992712    
state5                         1.656172   0.780302   2.122 0.033798 *  
state6                         0.908340   0.741876   1.224 0.220808    
state7                         0.885528   0.718156   1.233 0.217554    
state8                         0.431630   0.805964   0.536 0.592273    
state9                         0.728948   0.741603   0.983 0.325639    
state10                        0.616633   0.747908   0.824 0.409669    
state11                        0.991607   0.756233   1.311 0.189775    
state12                       -0.137129   0.885779  -0.155 0.876969    
```



```
state47                          0.388544   0.747217   0.520 0.603072
state48                          1.348421   0.712964   1.891 0.058586 .
state49                          0.291119   0.759541   0.383 0.701510
state50                          0.468071   0.723731   0.647 0.517795
state51                          0.316310   0.740916   0.427 0.669439
international.plan2              1.947440   0.143483  13.573  < 2e-16 ***
voice.mail.plan2                -2.073675   0.552731  -3.752 0.000176 ***
number.vmail.messages            1.886261   0.868623   2.172 0.029889 *
total.day.minutes                3.095653   0.308653  10.030  < 2e-16 ***
total.eve.minutes                1.566080   0.309057   5.067 4.04e-07 ***
total.night.minutes              0.989739   0.303488   3.261 0.001109 **
total.intl.minutes               0.912381   0.303827   3.003 0.002674 **
total.intl.calls                -1.164166   0.274986  -4.234 2.30e-05 ***
number.customer.service.calls -0.386610   0.175105  -2.208 0.027253 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2758.3  on 3332  degrees of freedom
Residual deviance: 2277.3  on 3273  degrees of freedom
AIC: 2397.3

Number of Fisher Scoring iterations: 5
```

The logistic regression model in R automatically creates a set of dummy variables for each level in a categorical variable. The *Akaike Information Criterion* decides the relative quality of this model as it deals with the information lost by the model. A lower value is favored.



```
> LR_Pred = predict(LR_model, newdata = test[,-11], type = "response")
> LR_Pred = ifelse(LR_Pred > 0.5, 1, 0)
> confusionMatrix(table(test$Churn, LR_Pred))
Confusion Matrix and Statistics

   LR_Pred
       0    1
  0 1424   19
  1  193   31

               Accuracy : 0.8728
                 95% CI : (0.8559, 0.8885)
    No Information Rate : 0.97
    P-Value [Acc > NIR] : 1

                  Kappa : 0.1864
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8806
            Specificity : 0.6200
         Pos Pred Value : 0.9868
         Neg Pred Value : 0.1384
             Prevalence : 0.9700
         Detection Rate : 0.8542
   Detection Prevalence : 0.8656
      Balanced Accuracy : 0.7503

       'Positive' Class : 0
```

Accuracy: 87.28

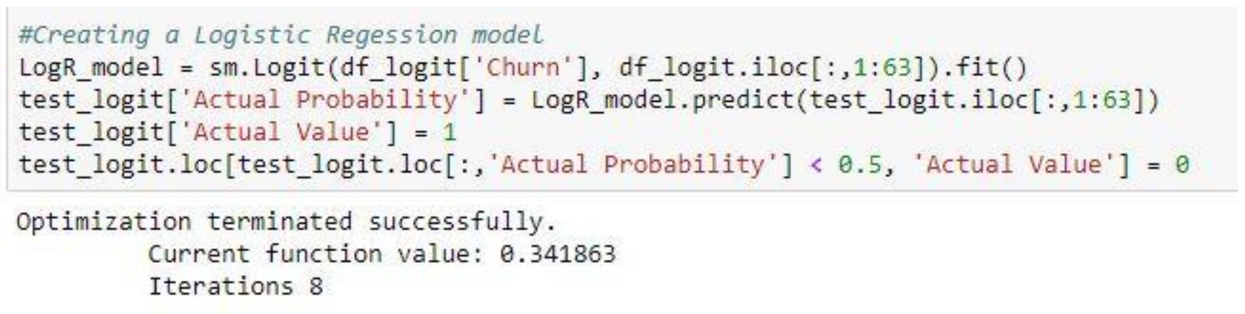
FNR: 0.861

FPR: 0.01

In Python, some manual data manipulation is required as we first need to create a set of dummy variables before implementing a model. This is done as shown below.

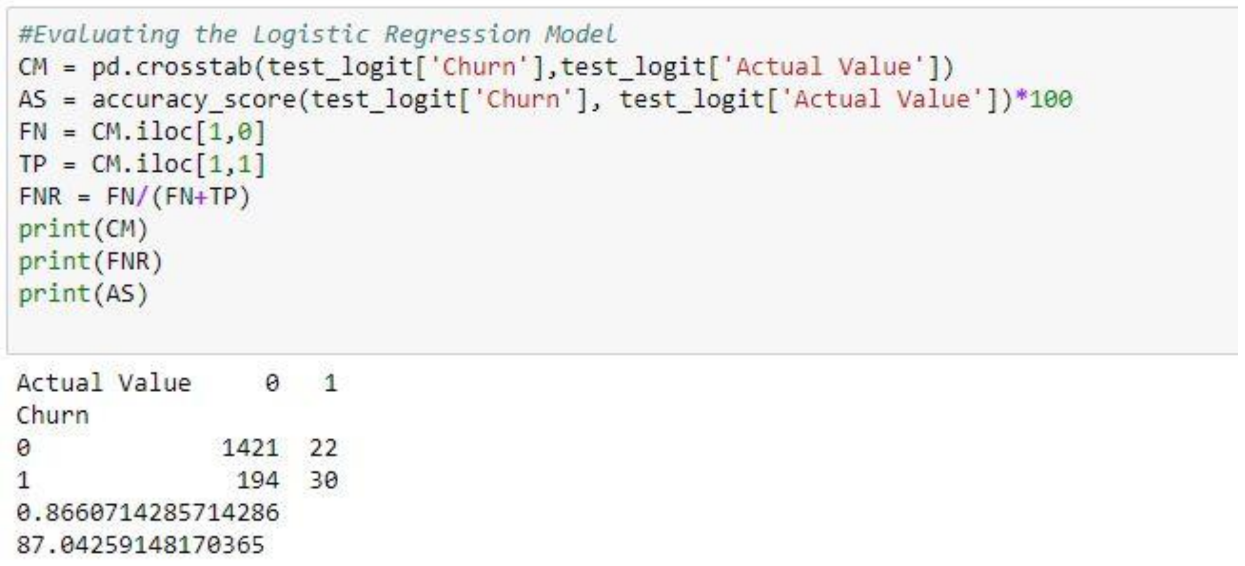
```
df_logit = pd.DataFrame(df['Churn'])
df_logit = df_logit.join(df[num_nm])

for i in cat_nm:
    if(i != 'Churn'):
        temp = pd.get_dummies(df[i], prefix = i)
        df_logit = df_logit.join(temp)
```

After this, a model was developed using the *sm.Logit()* function in Python.



```
#Creating a Logistic Regession model
LogR_model = sm.Logit(df_logit['Churn'], df_logit.iloc[:,1:63]).fit()
test_logit['Actual Probability'] = LogR_model.predict(test_logit.iloc[:,1:63])
test_logit['Actual Value'] = 1
test_logit.loc[test_logit.loc[:,'Actual Probability'] < 0.5, 'Actual Value'] = 0
```

```
Optimization terminated successfully.
         Current function value: 0.341863
         Iterations 8
```



```
#Evaluating the Logistic Regression Model
CM = pd.crosstab(test_logit['Churn'],test_logit['Actual Value'])
AS = accuracy_score(test_logit['Churn'], test_logit['Actual Value'])*100
FN = CM.iloc[1,0]
TP = CM.iloc[1,1]
FNR = FN/(FN+TP)
print(CM)
print(FNR)
print(AS)
```

```
Actual Value     0   1
Churn
0             1421  22
1              194  30
0.8660714285714286
87.04259148170365
```

Accuracy: 87.04

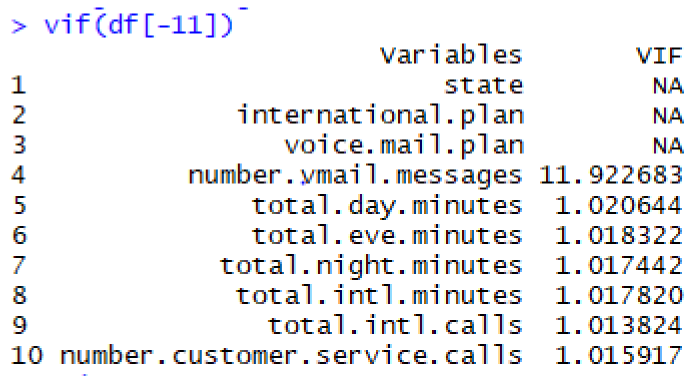
FNR: 0.866

FPR: 0.01

The logistic regression models developed in both R and Python are not nearly as good as the tree based models developed. Accuracy in both cases are about 87% which is fine, but the FNR rate is at 0.86 which is very high.

Variance Inflation Factor:

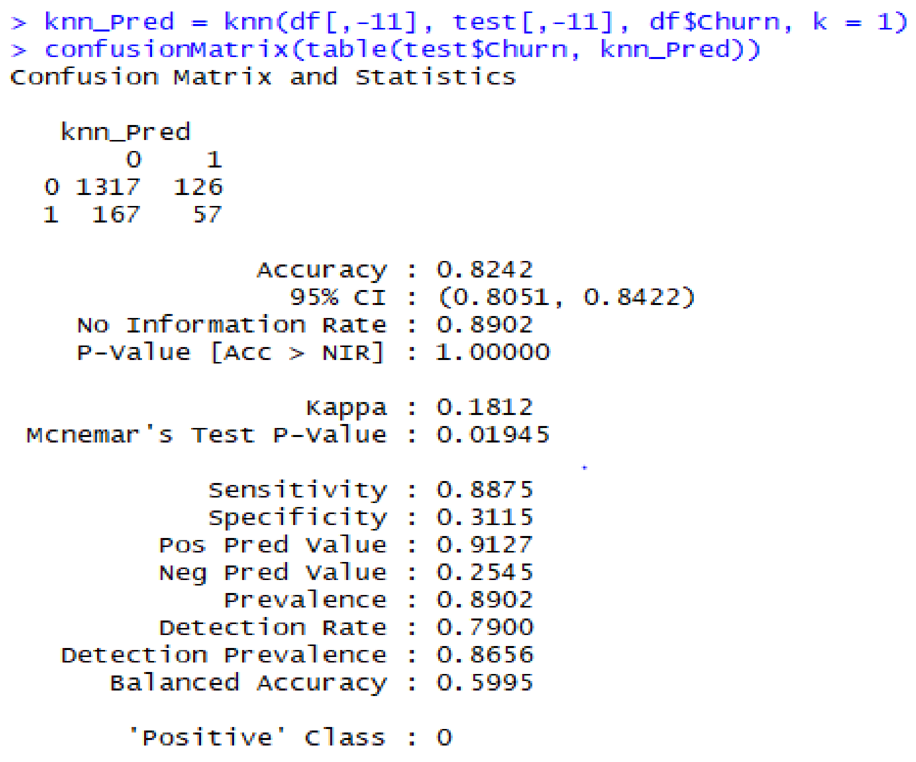
VIF is a quantity that gives us an idea of the multicollinearity that exists within our dataset. In R, we tested this using the *vif()* function in the *usdm* library as shown. *Number vmail messages* had very high VIF, but the performance of the models remained relatively unaffected on dropping this variable.



```
> vif(df[-11])
                    Variables       VIF
1                       state        NA
2          international.plan        NA
3             voice.mail.plan        NA
4         number.vmail.messages 11.922683
5           total.day.minutes  1.020644
6           total.eve.minutes  1.018322
7         total.night.minutes  1.017442
8          total.intl.minutes  1.017820
9            total.intl.calls  1.013824
10 number.customer.service.calls  1.015917
```

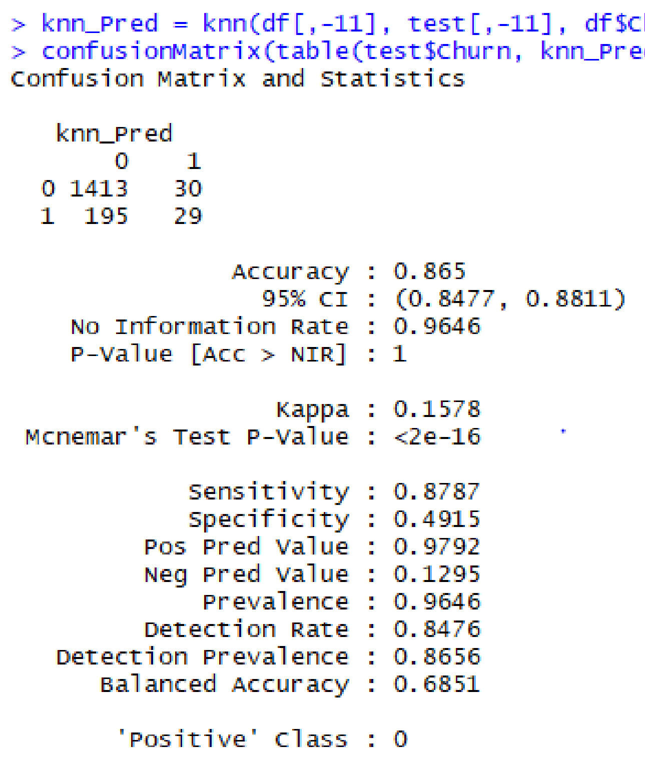
- KNN Imputation

So far, our models have either been rule based or weights based. K-NN is a lazy learner, it doesn't learn a discriminative function from the training data but "memorizes" the training dataset instead. In R, we used the *knn* function in the *DMwR* library.



```
> knn_Pred = knn(df[,-11], test[,-11], df$Churn, k = 1)
> confusionMatrix(table(test$Churn, knn_Pred))
Confusion Matrix and Statistics

   knn_Pred
       0    1
  0 1317  126
  1  167   57

               Accuracy : 0.8242
                 95% CI : (0.8051, 0.8422)
    No Information Rate : 0.8902
    P-Value [Acc > NIR] : 1.00000

                  Kappa : 0.1812
 Mcnemar's Test P-Value : 0.01945

            Sensitivity : 0.8875
            Specificity : 0.3115
         Pos Pred Value : 0.9127
         Neg Pred Value : 0.2545
             Prevalence : 0.8902
         Detection Rate : 0.7900
   Detection Prevalence : 0.8656
      Balanced Accuracy : 0.5995

       'Positive' Class : 0
```

Accuracy: 82.42
FNR: 0.7455
FPR: 0.087

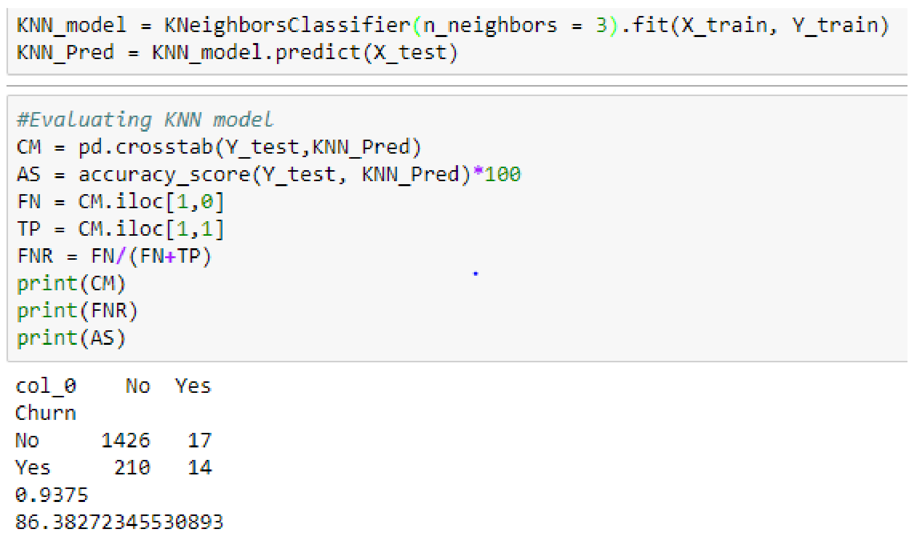


```
> knn_Pred = knn(df[,-11], test[,-11], df$Churn, k = 5)
> confusionMatrix(table(test$Churn, knn_Pred))
Confusion Matrix and Statistics

   knn_Pred
       0    1
  0 1413   30
  1  195   29

               Accuracy : 0.865
                 95% CI : (0.8477, 0.8811)
    No Information Rate : 0.9646
    P-Value [Acc > NIR] : 1

                  Kappa : 0.1578
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8787
            Specificity : 0.4915
         Pos Pred Value : 0.9792
         Neg Pred Value : 0.1295
             Prevalence : 0.9646
         Detection Rate : 0.8476
   Detection Prevalence : 0.8656
      Balanced Accuracy : 0.6851

       'Positive' Class : 0
```

Accuracy: 86.5
FNR: 0.87
FPR: 0.02

It can be seen that as we increase the number of neighbors, our accuracy and False Positive Rate improve, but our False Negative Rate gets poorer.

In Python, we used the *KNeighborsClassifier()* function from the *sklearn.neighbors* library to generate our knn machine learning model.



```
KNN_model = KNeighborsClassifier(n_neighbors = 3).fit(X_train, Y_train)
KNN_Pred = KNN_model.predict(X_test)
```

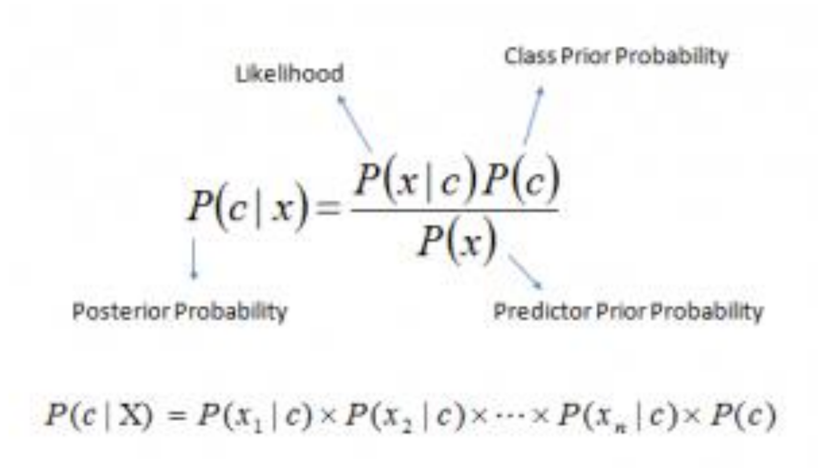
```
#Evaluating KNN model
CM = pd.crosstab(Y_test,KNN_Pred)
AS = accuracy_score(Y_test, KNN_Pred)*100
FN = CM.iloc[1,0]
TP = CM.iloc[1,1]
FNR = FN/(FN+TP)
print(CM)
print(FNR)
print(AS)
```

```
col_0    No  Yes
Churn
No     1426   17
Yes     210   14
0.9375
86.38272345530893
```

Accuracy: 86.38
FNR: 0.93
FPR: 0.01

- <u>Naïve Bayes</u>

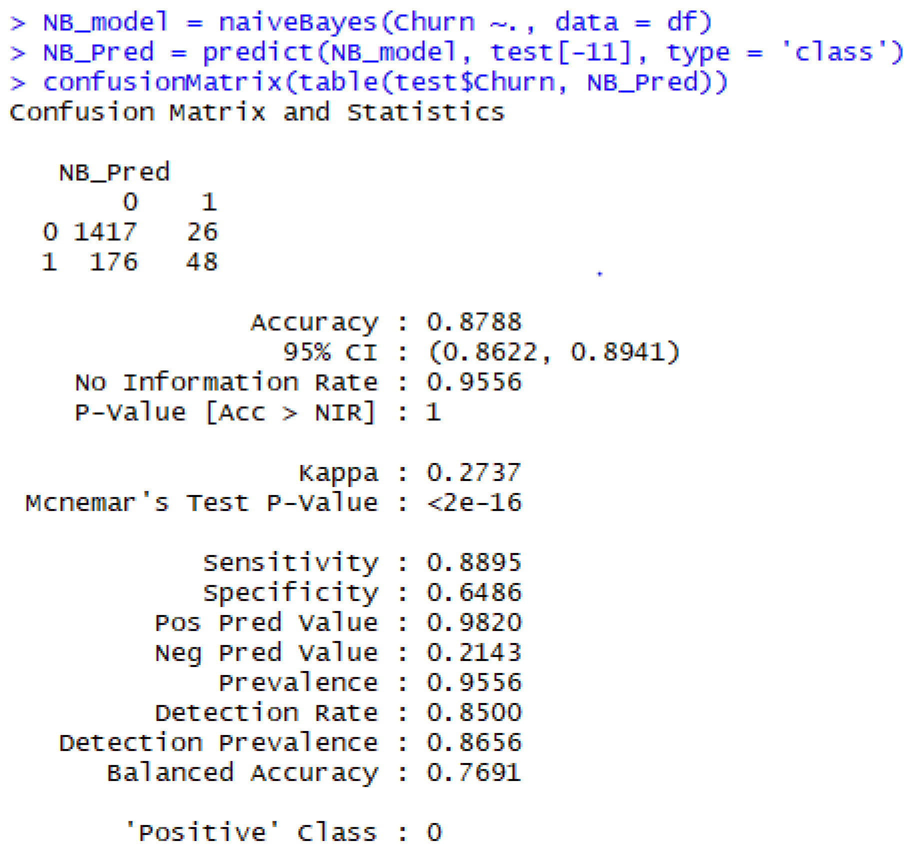
Naïve Bayes is a classification technique based on Bayes’ Theorem with an assumption of independence among predictors. A Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.



$$P(c \mid x) = \frac{P(x \mid c)P(c)}{P(x)}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

To implement this in R, we use the *naiveBayes()* function available in the *e1071* library.



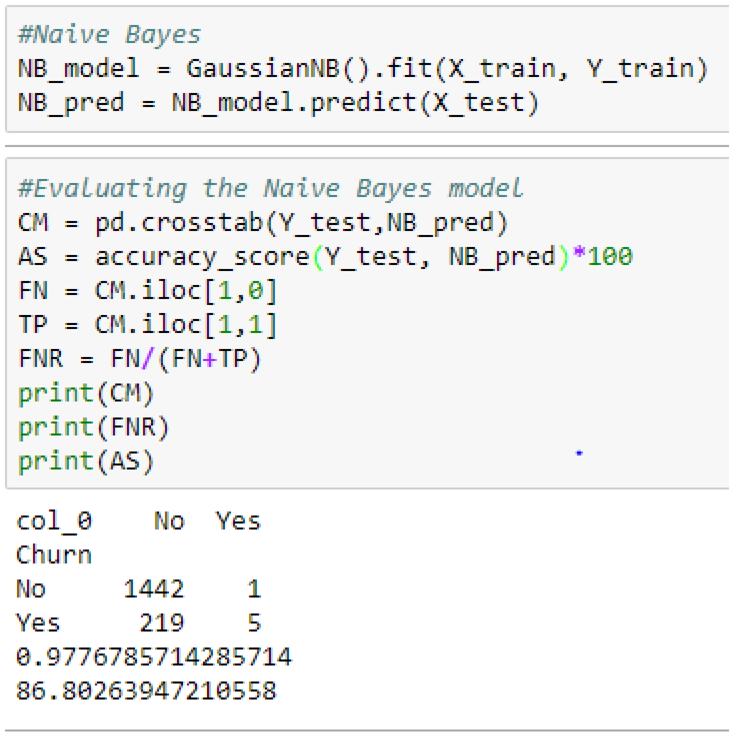
```
> NB_model = naiveBayes(Churn ~., data = df)
> NB_Pred = predict(NB_model, test[-11], type = 'class')
> confusionMatrix(table(test$Churn, NB_Pred))
Confusion Matrix and Statistics

   NB_Pred
       0    1
  0 1417   26
  1  176   48

               Accuracy : 0.8788
                 95% CI : (0.8622, 0.8941)
    No Information Rate : 0.9556
    P-Value [Acc > NIR] : 1

                  Kappa : 0.2737
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8895
            Specificity : 0.6486
         Pos Pred Value : 0.9820
         Neg Pred Value : 0.2143
             Prevalence : 0.9556
         Detection Rate : 0.8500
   Detection Prevalence : 0.8656
      Balanced Accuracy : 0.7691

       'Positive' Class : 0
```

Accuracy: 87.88

FNR: 0.785

FPR: 0.018

In Python, the *GaussianNB()* function available in the *sklearn.naive_bayes* library was used.



```
#Naive Bayes
NB_model = GaussianNB().fit(X_train, Y_train)
NB_pred = NB_model.predict(X_test)
```

```
#Evaluating the Naive Bayes model
CM = pd.crosstab(Y_test,NB_pred)
AS = accuracy_score(Y_test, NB_pred)*100
FN = CM.iloc[1,0]
TP = CM.iloc[1,1]
FNR = FN/(FN+TP)
print(CM)
print(FNR)
print(AS)
```

```
col_0    No  Yes
Churn
No     1442    1
Yes     219    5
0.9776785714285714
86.80263947210558
```

Accuracy: 86.80
FNR: 0.977
FPR: 0

Nothing about this model in R or Python stands out relative to the other models. The FNR is unacceptably high.

Closing Thoughts

The point of this project is to identify existing customers who are likely to churn out of the telecom service and to focus resources on retaining these customers. In this regard, it is important to have a good accuracy in the model, a good (i.e low) False Negative Rate and a good False Positive Rate. However the training set provided to us is heavily skewed in the False cases (where the customer doesn't churn). A dataset with more proportional True cases would enable us to have a lower FNR than what was obtained. With the models prepared, here are my suggestions:

1. In R
   I would suggest using the Random Forest Model as it has the highest accuracy and lowest False Negative Rate.
2. In Python
   Based on business user requirement, I would suggest either Decision Tree or Random Forest model since the former has better FNR, while the latter has better Accuracy and FPR.