Project Name -

CHURN REDUCTION

by

Harsh Agarwal

INTRODUCTION

Project Description:

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 problem statement is targeted at enabling churn reduction using analytics concepts.

Problem Statement:

The objective of this Case is to predict customer behaviour. We are providing you a public dataset that has customer usage pattern and if the customer has moved or not. We expect you to develop an algorithm to predict the churn score based on usage pattern.

Data:

Dataset provided consists of total of 5000 observations with 3333 observations in train_set and 1667 in test_set. Below is the sample of the dataset provided:

state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes
KS	128	415	382-4657	no	yes	25	265.1	110	45.07	197.4
OH	107	415	371-7191	no	yes	26	161.6	123	27.47	195.5
NJ	137	415	358-1921	no	no	0	243.4	114	41.38	121.2
OH	84	408	375-9999	yes	no	0	299.4	71	50.9	61.9
OK	75	415	330-6626	yes	no	0	166.7	113	28.34	148.3
AL	118	510	391-8027	yes	no	0	223.4	98	37.98	220.6
MA	121	510	355-9993	no	yes	24	218.2	88	37.09	348.5
MO	147	415	329-9001	yes	no	0	157	79	26.69	103.1
LA	117	408	335-4719	no	no	0	184.5	97	31.37	351.6
WV	141	415	330-8173	yes	yes	37	258.6	84	43.96	222
IN	65	415	329-6603	no	no	0	129.1	137	21.95	228.5
RI	74	415	344-9403	no	no	0	187.7	127	31.91	163.4
IA	168	408	363-1107	no	no	0	128.8	96	21.9	104.9
MT	95	510	394-8006	no	no	0	156.6	88	26.62	247.6
IA	62	415	366-9238	no	no	0	120.7	70	20.52	307.2

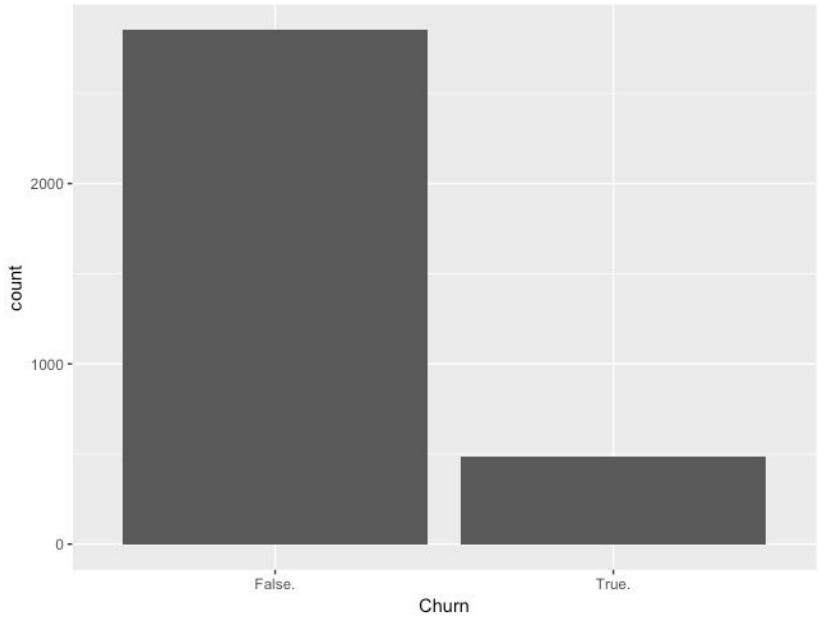
total eve calls	total eve charge	total night minutes	total night calls	total night charge	total intl minutes	total intl calls	total intl charge	number customer service calls	Churn
99	16.78	244.7	91	11.01	10	3	2.7	1	False.
103	16.62	254.4	103	11.45	13.7	3	3.7	1	False.
110	10.3	162.6	104	7.32	12.2	5	3.29	0	False.
88	5.26	196.9	89	8.86	6.6	7	1.78	2	False.
122	12.61	186.9	121	8.41	10.1	3	2.73	3	False.
101	18.75	203.9	118	9.18	6.3	6	1.7	0	False.
108	29.62	212.6	118	9.57	7.5	7	2.03	3	False.
94	8.76	211.8	96	9.53	7.1	6	1.92	0	False.
80	29.89	215.8	90	9.71	8.7	4	2.35	1	False.
111	18.87	326.4	97	14.69	11.2	5	3.02	0	False.
83	19.42	208.8	111	9.4	12.7	6	3.43	4	True.
148	13.89	196	94	8.82	9.1	5	2.46	0	False.
71	8.92	141.1	128	6.35	11.2	2	3.02	1	False.
75	21.05	192.3	115	8.65	12.3	5	3.32	3	False.

In the above data, we have a total of 20 predictor variables and **“Churn”** as the target variable.

Predictors

"state"	"account.length"
"area.code"	"phone.number"
"international.plan"	"voice.mail.plan"
"number.vmail.messages"	"total.day.minutes"
"total.day.calls"	"total.day.charge"
"total.eve.minutes"	"total.eve.calls"
"total.eve.charge"	"total.night.minutes"
"total.night.calls"	"total.night.charge"
"total.intl.minutes"	"total.intl.calls"
"total.intl.charge"	"number.customer.service.calls"

The target variable contains 2 classes: “True” and “False” but the dataset is highly imbalanced towards “False” label as is evident from the below barchart:



Data Exploration and Preparation

Data exploration and preprocessing can take upto 70% of the total project time and is a very important part as the quality of the input decides the quality of the output. We need to identify relationships between the predictor variables and the target variables, impute missing values, clean and standardize the data.

Firs

```
train_dataset= pd.read_csv('Train_data.csv')
test_dataset= pd.read_csv('Test_data.csv')
```

After combining the dataset, first thing is to trim all the cells for any leading and trailing white spaces which is achieved by the following code:

```
cat_var= [x for x in dataset.dtypes.index if dataset.dtypes[x]=='object']
train_dataset[cat_var]= train_dataset[cat_var].apply(lambda x: x.str.strip())
test_dataset[cat_var]= test_dataset[cat_var].apply(lambda x: x.str.strip())
```

Univariate Analysis:

Now, we need to explore all the variables one by one. We have a total of 20 predictors out of which 16 are continuous variables whereas 3 are categorical variables.

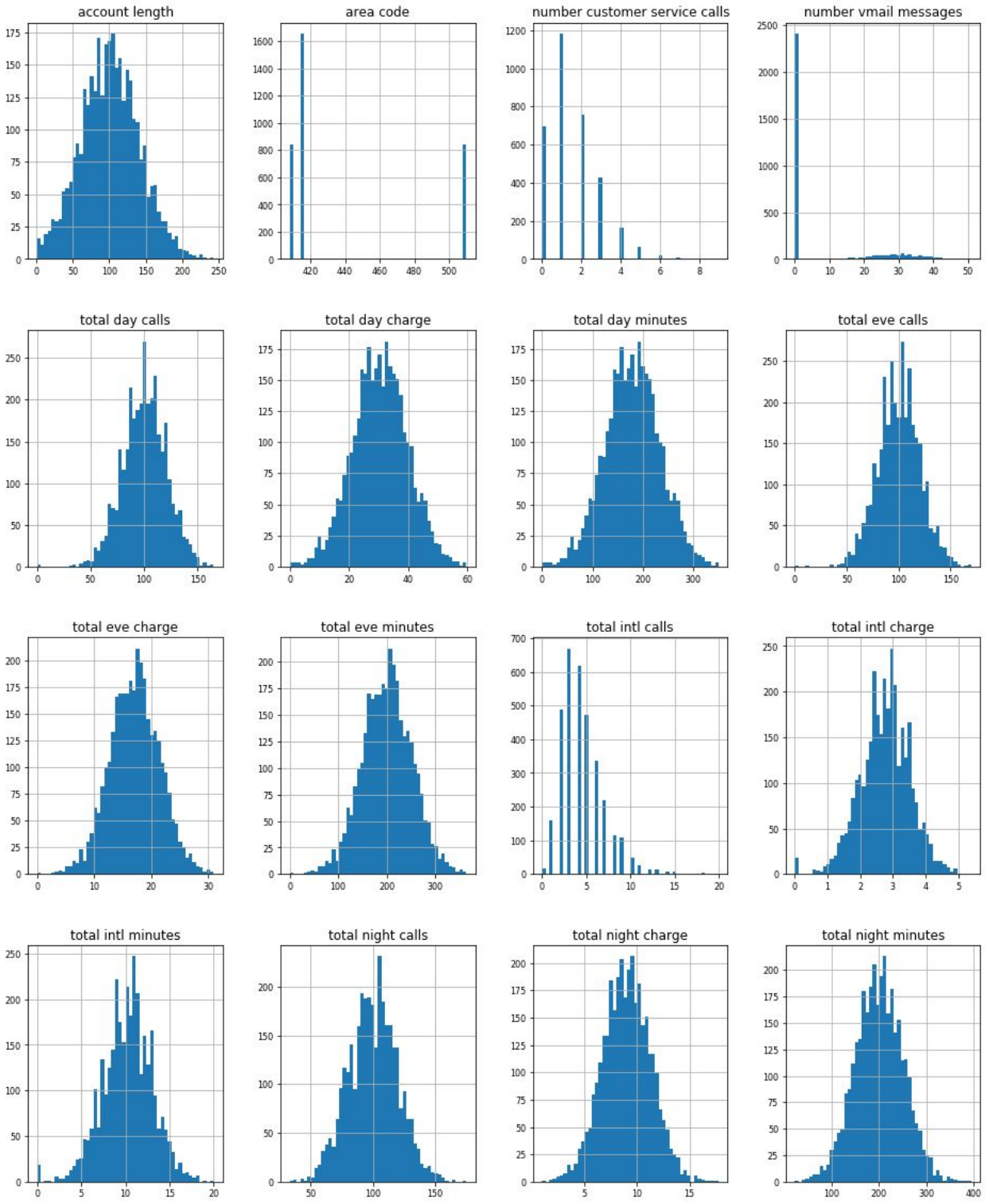
For the continuous variables, we use the below code to understand the central tendency and spread of the variable.

```
summary(train_dataset)
```

Index	account length	area code	nber vmail messa	total day minutes	total day calls	total day charge	total eve minutes
count	3333	3333	3333	3333	3333	3333	3333
mean	101.0648065	437.1824182	8.099009901	179.7750975	100.4356436	30.56230723	200.980348
std	39.82210593	42.37129049	13.68836537	54.4673892	20.06908421	9.259434554	50.71384443
min	1	408	0	0	0	0	0
25%	74	408	0	143.7	87	24.43	166.6
50%	101	415	0	179.4	101	30.5	201.4
75%	127	510	20	216.4	114	36.79	235.3
max	243	510	51	350.8	165	59.64	363.7

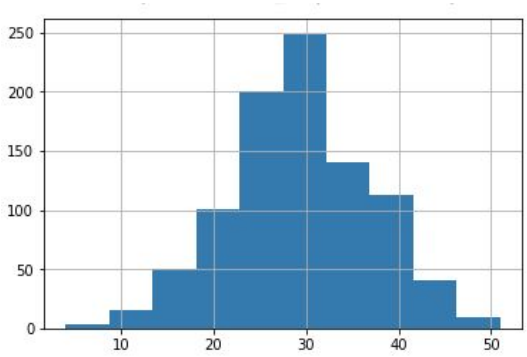
total eve calls	total eve charge	otal night minutes	total night calls	total night charge	total intl minutes	total intl calls	total intl charge	r customer servic
3333	3333	3333	3333	3333	3333	3333	3333	3333
100.1143114	17.08354035	200.8720372	100.1077108	9.039324932	10.23729373	4.479447945	2.764581458	1.562856286
19.92262529	4.310667643	50.57384701	19.56860935	2.275872838	2.791839548	2.461214271	0.7537726127	1.315491045
0	0	23.2	33	1.04	0	0	0	0
87	14.16	167	87	7.52	8.5	3	2.3	1
100	17.12	201.2	100	9.05	10.3	4	2.78	1
114	20	235.3	113	10.59	12.1	6	3.27	2
170	30.91	395	175	17.77	20	20	5.4	9

From the above table we can conclude that there are no missing values for any numeric features and there are some high range values which can possibly be because of outliers for eg: account_length which has a max value of 243 which is almost 4 standard deviations away from the mean. To get a more clear picture lets look at their distributions:



Most of the features above follow a normal distribution, except for international calls which is right skewed with peak at around 4, and number customer service calls which again is right skewed.

‘Number vmail messages’ has a lot of 0’s in the table, so lets remove the 0’s and then look at the distribution:



‘Number vmail messages’ also follows a normal distribution with a peak at around 30, if all the 0’s are removed.

‘Area code’ is randomly distributed and contains only 3 different values. We’ll analyze it further with respect to the target variable during bivariate analysis.

Next, we see the distribution of the categorical variables. Below table displays the unique values for each variable

train_dataset[cat_var].apply(lambda x: len(x.unique()))

state	51
phone number	3333
international plan	2
voice mail plan	2
Churn	2

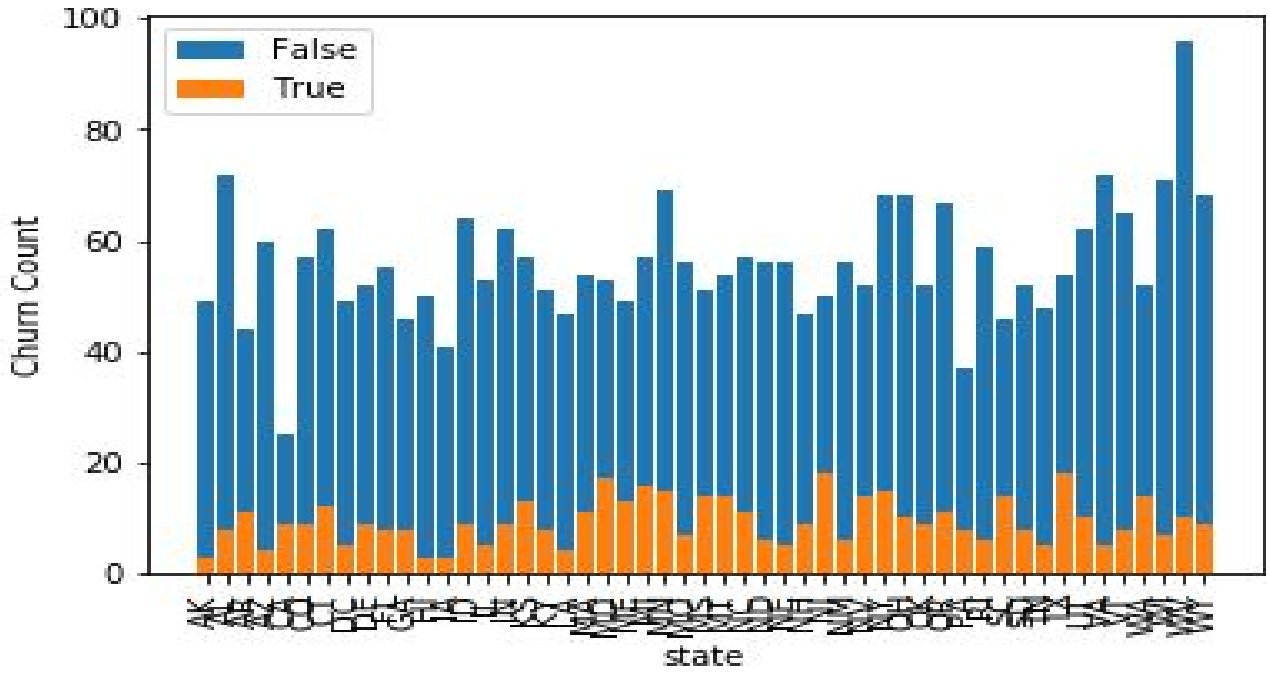
As expected international and voicemail plan have 2 unique values (yes and no), but the state variable has a total of 51 values which we will need to explore further.
Other important thing is that the variable phone number has a unique value for each observation, so we can remove this feature from our dataset as it is not adding any information to the model. We can drop this feature from our dataset.

Bivariate Analysis:

Bivariate analysis is used to find the relations between 2 variables, whether it be continuous or categorical.

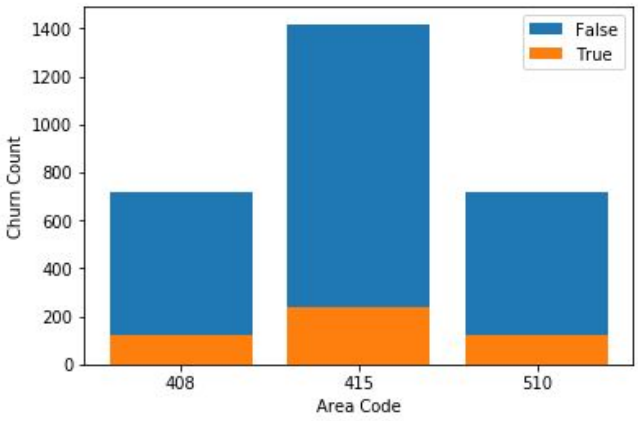
First, we find the relation between the categorical variables and the target variable.

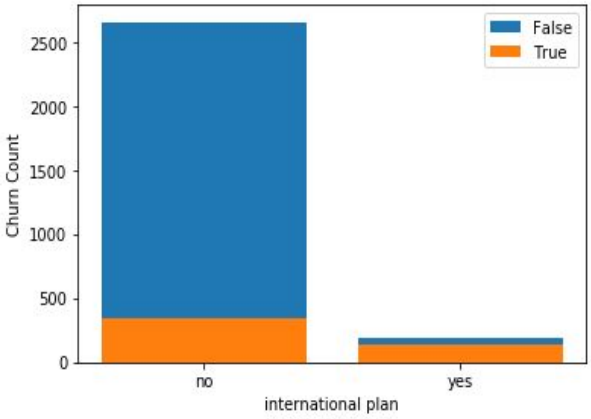
There are 51 unique values for the state variable, below is a bar graph depicting the frequency of each state with respect to the target variable.

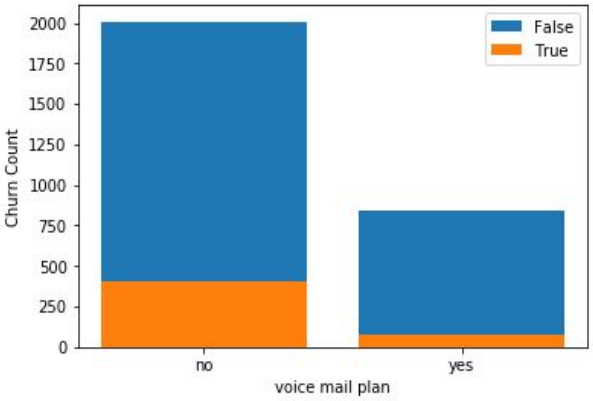


The figure clearly shows that some of the states have a higher churn rate as compared to others, so we are going to keep the feature.

Similarly, below are the graphs for international plan,voicemail plan, area code w.r.t. Churn.

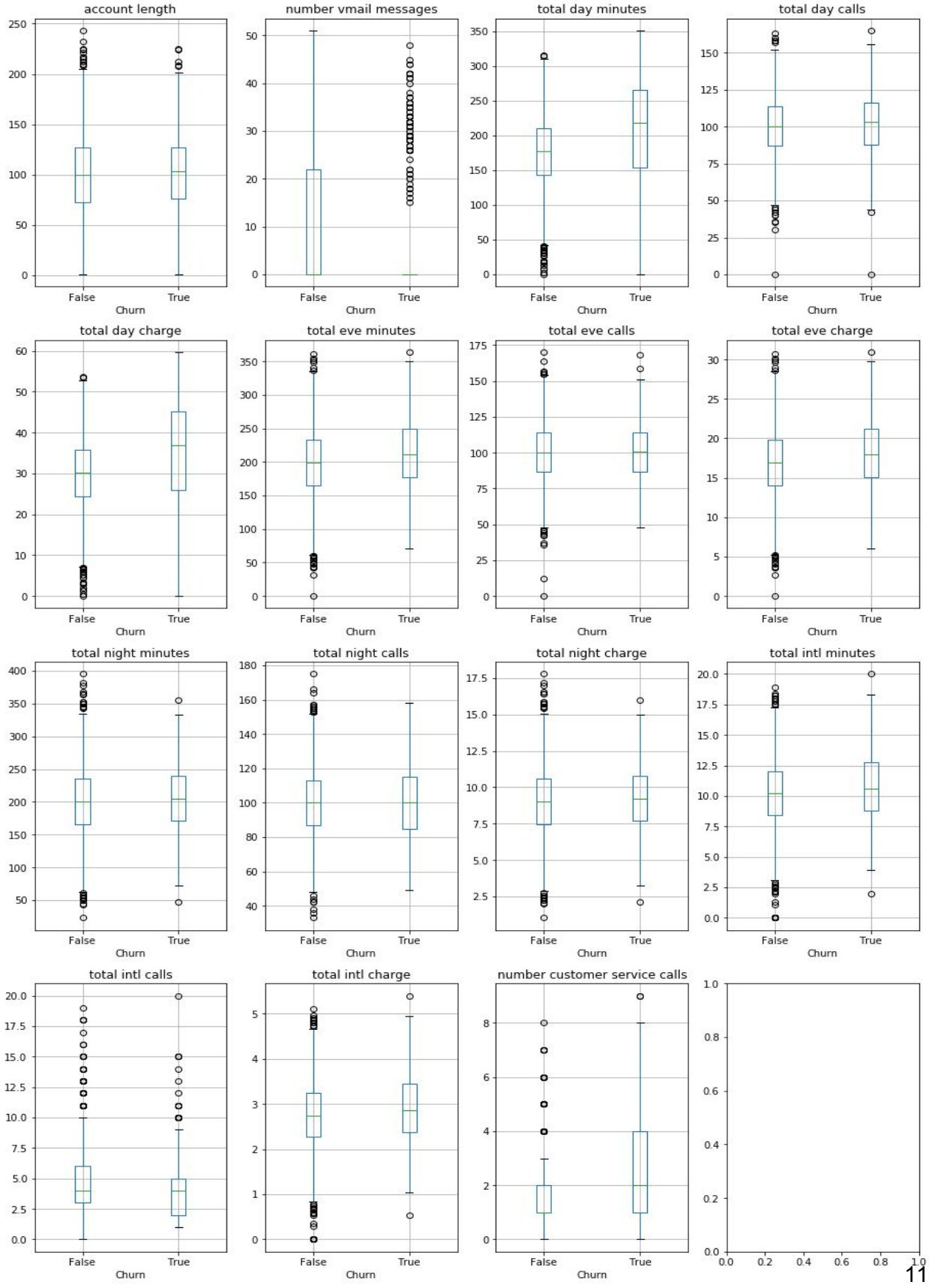






‘Area code’ can be converted to categorical variable as there are only 3 values and no such relationship with the increasing area codes.

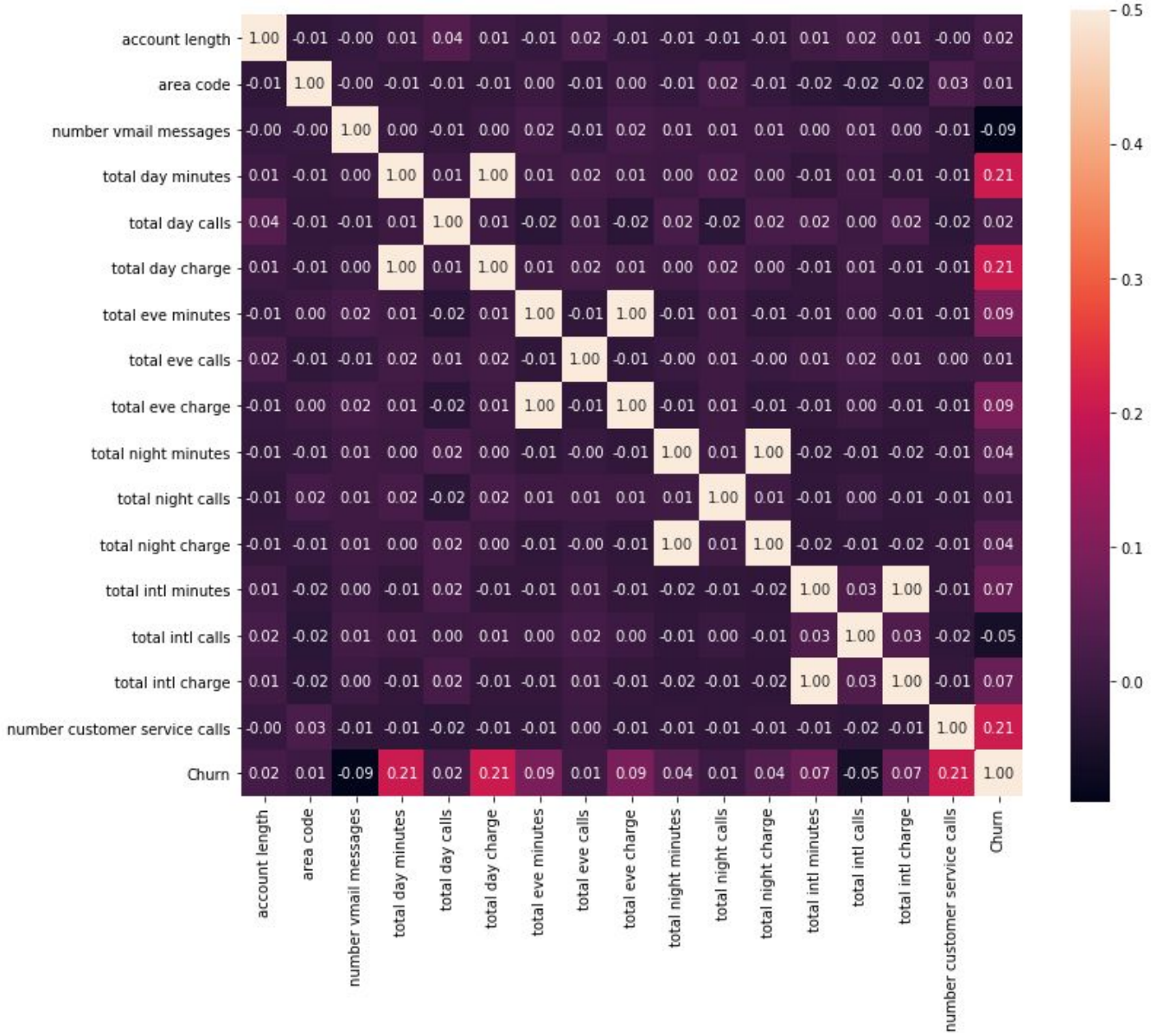
Next, we observe the relationship between the numeric variables and the target variables with the help of boxplots.


account length
number vmail messages
total day minutes
total day calls
total day charge
total eve minutes
total eve calls
total eve charge
total night minutes
total night calls
total night charge
total intl minutes
total intl calls
total intl charge
number customer service calls
False
True
Churn

There are a number of outliers visible in the boxplots above but as was the case with ‘number vmail messages’ which was visible in univariate analysis, most of the outliers are due to the number of 0’s present in each column. We can get a more clear picture by looking at the correlation heatmap which will help to understand the relationship between predictor variables and their correlation with the target variable as well.

Before that, first convert the target variable ‘Churn’ to numeric by mapping its values True and False to 0 and 1.

Below is the heatmap for the correlation matrix:



From the above figure, we can conclude that there is a very strong correlation(~1) between some of the predictors: (day/eve/night/intl charge-minutes) and also some of the predictors are showing a strong correlation with the target variable as compared to others. International plan, total day minutes, total day charge and number customer service calls show the most significant relation with Churn.

For the highly correlated independent variables, we can remove one of the variables or apply dimensionality reduction techniques.

Combining train and test dataset:

After performing the exploratory analysis, we now need to impute missing values, and label encode categorical variables. For that we can first combine both the datasets to reduce our efforts.

```
train_dataset['source']='train'
test_dataset['source']='test'
dataset= pd.concat([train_dataset,test_dataset],axis=0)
```

Missing Values:

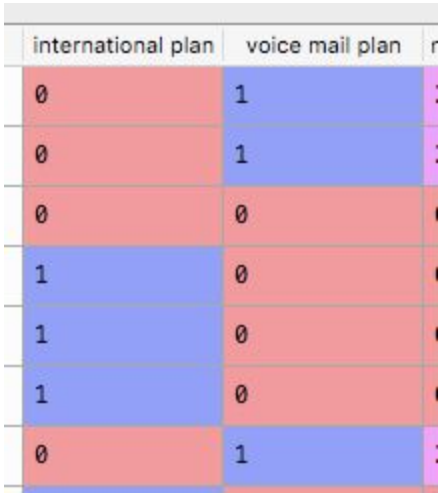
Next, we find the number of missing values in our dataset:

```
dataset.apply(lambda x: sum(x.isnull()))
```

The above line of code returns 0 for all the columns ie we have a complete dataset without any missing values.

Label Encoding:

In the dataset provided, we have a total of 3 categorical variables including the target variable. (State, Voice mail plan, International plan, Churn). We need to convert the categorical variables to numeric. Except for ‘state’ all other variables have binary labels and so ‘No’ is mapped to 0 and ‘Yes’ is mapped to 1.



international plan	voice mail plan	r
0	1	
0	1	
0	0	
1	0	
1	0	
1	0	
0	1	

One-Hot Encoding:

‘State’ feature is a nominal variable as there is no clear order between the values so label encoding won’t be as efficient,therefore, we need to create dummy variables which will be a better representation of the feature.

The transformation results in a total of 70 variables as 50 dummy variables are added for the 51 states.

state_AZ	state_CA	state_CO	state_CT	state_DC	state_DE	state_FL	state_GA	state_HI	state_IA	state_ID
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0

Similarly, ‘Area Code’ which is a numeric variable but contains only 3 values do not have any correlation with the target variable with increasing value ie it is a nominal variable just like state, so we’ll first convert it to categorical variable and then encode it.

area code_415	area code_510
1	0
1	0
1	0
0	0
1	0
0	1
0	1
1	0
0	0
1	0
1	0
1	0
0	0

Convert back to train and test dataset:

Finally, after removing the variable 'phone-number' from the dataset, we are left with 69 predictor variables and 5000 observations.
Now, we convert the dataset back to train and test dataset.
After that we separate the target variable from both test and train dataset to get:
xtrain(3333,69)
ytrain(3333,)
xtest(1667,69)
ytest(1667,)

Feature Scaling

In order to avoid the influence of large scale data on the weights, feature scaling is performed so that feature has equal opportunity to influence the weights and increase accuracy in predicting the target variable. Standardization is used to scale the data as most of the predictors are normally distributed.
Both the test and the train data are standardized, but the test data is standardized using the mean and variance of train data so as to prevent information from test dataset leaking into training dataset.

Dimensionality Reduction:

To remove highly correlated variables and reduce the dimensionality, we applied **PCA algorithm** and then checked the explained_variance_ratio, but there was no individual dimension with high value of explained variance and the performance did not improve so we will proceed further with all 69 features and treat them while tuning different hyperparameters for all models.

Modeling:

During the data exploration and preprocessing stage, the number of features increased to a total of 69 and the dimensionality reduction did not improve the performance, so we are going to use regularization and other hyperparameters to prevent overfitting.

Also the ratio of positive to negative class in the train set is around 1:7, so, in order to balance both the labels, clas_weights and cost sensitive learning will be used so as to increase the penalization on misclassification of positive class.

Evaluation Metrics:

Due to class imbalance, accuracy won't be a correct metric to evaluate the models, as it will tend to predict the negative class more which will lead to increased accuracy but a poor model.

Our main aim is to reduce false negatives ie we don't want to classify customers who are likely to churn as False as it will lead to losses, that's where Recall is important.

Recall is the ability of classifier to identify relevant instances and is given by true positives divided by a total of true positives and false negatives.

However, while recall is important we cannot improve it all the way to 1 at the cost of drasticallly reducing precision and so we need a balance between the two to get an optimal classifier. That's why we will use F1 Score which is a harmonic mean between recall and precision to evaluate the models.

Stratified K-Cross Validation and Grid Search:

Cross validation technique is used to evaluate the model by splitting the train dataset into K folds where each fold will be evaluated by training the model on the other folds and then a mean F1 score for all the folds will be calculated. As it is an imbalanced

class problem ,startification will ensure that each fold contains the same proportion of both the classes.
The test set will be kept aside and will not be used in the training and parameters tuning process.

GridSearch is a tool with a built in cross-validation which is used to get the best combination of hyperparameters. It tries different values of each hyperparameters which we pass to it, perform k cross validation on each combination and returns the mean score. The combination of parameters with the best f1 mean score are selected and the train data will be refitted with those parameters.

Custom Functions:

To reduce the redundancy in code while building and training different models, soe custom functions are defined in the code to perform repetitive tasks. These are

- Function **model_fit** will be used to build the perform cross validation and grid search on the train dataset and return the best parameters. It will also return the evaluation metrics (ROC_AUC, F1 Score, Recall, Precision) for both train and test datasets. A boolean argument is passed to this function, which if False sets the param_grid to be empty so as to only perform cross validation.

- Theres' another function to print the confusion matrix for the test set. It will take predicted target values and ytest as input and print the confusion matrix heatmap.

- Feature Importance function is used to plot a graph displaying the importance of each predictor in decreasing order. It can be used to select the first few features with the most importance.

- Test_Set_Report will generate the final report for the test set displaying the scores for all evaluation metrics and will call the confusion matrix and roc_auc curve functions. This function will be called on the test set after the model tuning is done.

LOGISTIC REGRESSION:

Logistic Regression algorithm was implemented and was used to train the model.
To tackle the imbalance classes, class_weight='balanced' parameters. It adjusts the weights using the below methodology: *The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes *np.bincount(y)).*

```
logreg= LogisticRegression(class_weight='balanced' , random_state=42)
logreg=model_fit(logreg, xtrain, ytrain, param_logreg, False)
```

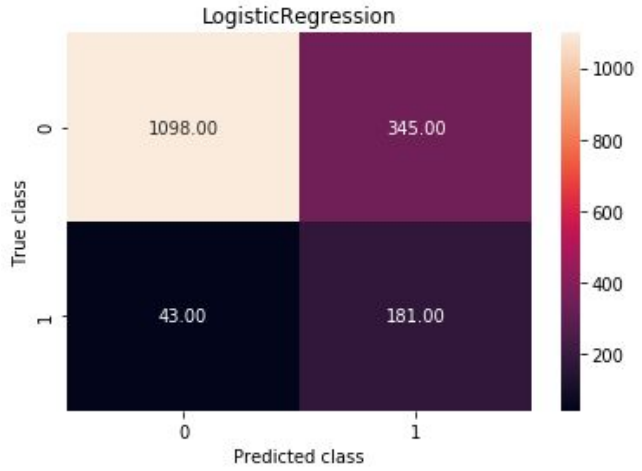
```
Train Fold Metrics:
ROC_AUC : 0.846331235567732
PRECISION : 0.37622287431453305
RECALL : 0.7883078282523999
F1-SCORE: 0.509338830231158

Test  Fold Metrics:
ROC_AUC : 0.8070408800918074
PRECISION : 0.34443055435031783
RECALL : 0.7206927522649171
F1-SCORE : 0.4656379882656029
```

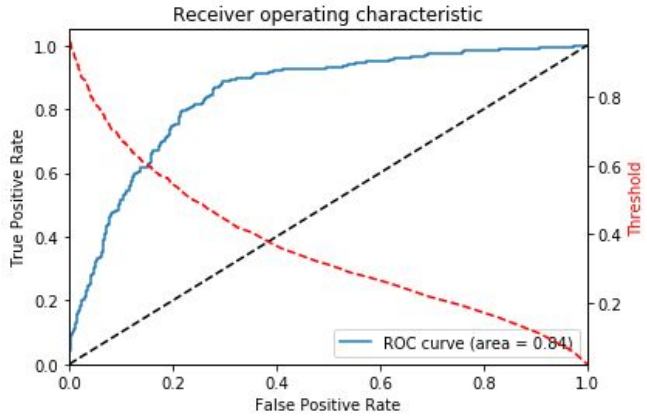
As we can see, though the ROC_AUC and RECALL are decent enough but the precision and consequently F1-score is around 0.46 for the test fold.
There is no such difference between train and test scores so we can rule out overfitting.
Even implementing the same with L1 regularization with a 'C' value of 0.05 (optimal value from gridsearch) did not improve the results.

On implementing the final model on the test_set_report, the results were unsatisfactory.

Test Set Report

Algorithm used: LogisticRegression
ROC_AUC: 0.7844752376002376
RECALL: 0.8080357142857143
PRECISION: 0.344106463878327
F1_SCORE: 0.48266666666666663
ACCURACY: 0.767246550689862





As was the case with test fold in cross validation, the test dataset also returned poor Precision and F1Score even for optimal logistic regression model.

DECISION TREES:

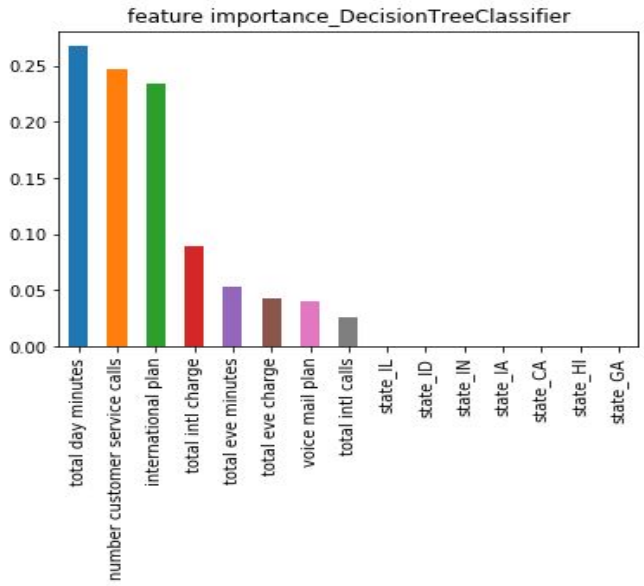
Next, we will implement decision trees. Again ‘class_weight’ was used to balance classes and hyperparameters: ‘max_depth’, ‘min_samples_split’ were used to prevent overfitting.

Below are the optimal parameters achieved using GridSearchCV to train the model.

```
dectree=
DecisionTreeClassifier(max_depth=4,min_samples_split=10,max_leaf_nodes=15,random_state
=42,class_weight='balanced')
dectree=model_fit(dectree, xtrain, ytrain,param_tree, False)
```

Evaluation metrics for best model:
Train Metrics:
ROC_AUC : 0.9149591871140679
PRECISION : 0.7079301670292303
RECALL : 0.841614116827998
F1-SCORE: 0.7685391476068617
Test Metrics:
ROC_AUC : 0.8875914407433688
PRECISION : 0.656837444918304
RECALL : 0.8033450252241718
F1-SCORE : 0.7225981446491372

We see an overall improvement in all the evaluation metrics from logistic regression. Precision has increased to 0.65 from 0.34 and subsequently F1 score increased to 0.72 from 0.46. Decision trees also has a very important function to calculate feature importance. Below is the graph representing the top15 features.



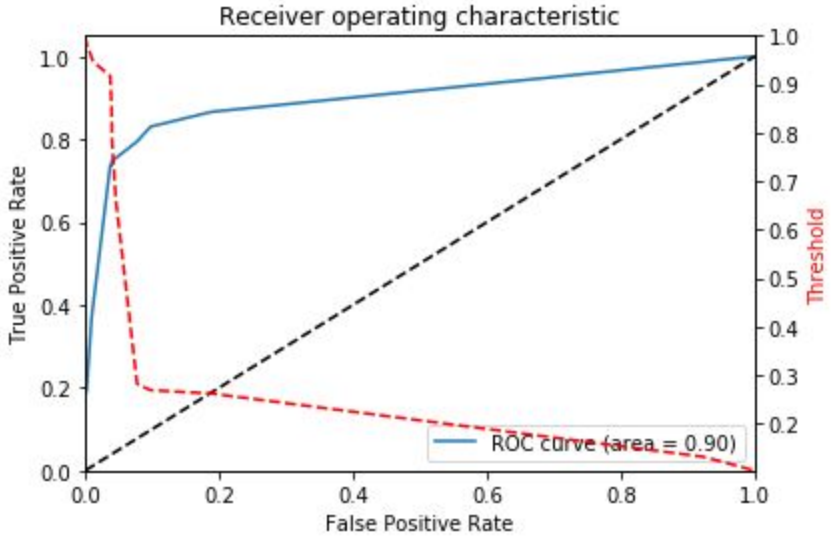
We implemented the decision tree algorithm again with the top 8 features and got the below results

Evaluation metrics for best model with top 15 features:
Train Metrics:
ROC_AUC : 0.9158506599822849
PRECISION : 0.7109534006340543
RECALL : 0.8395469333654656
F1-SCORE: 0.7695087492840107
Test Metrics:
ROC_AUC : 0.896987215847195
PRECISION : 0.6652442046425863
RECALL : 0.8012635670783572
F1-SCORE : 0.7268434750247628

Which did not show any improvement in the final results. So we proceed with all the variables with the optimal hyperparamters to check the final test report.

Test Set Report

Algorithm used: DecisionTreeClassifier
ROC_AUC: 0.8543631199881199
RECALL: 0.7544642857142857
PRECISION: 0.7191489361702128
F1_SCORE: 0.7363834422657951
ACCURACY: 0.9274145170965807



As we can see there is a substantial imporvement in the F1 Score for the test set as compared to logistic regression model.

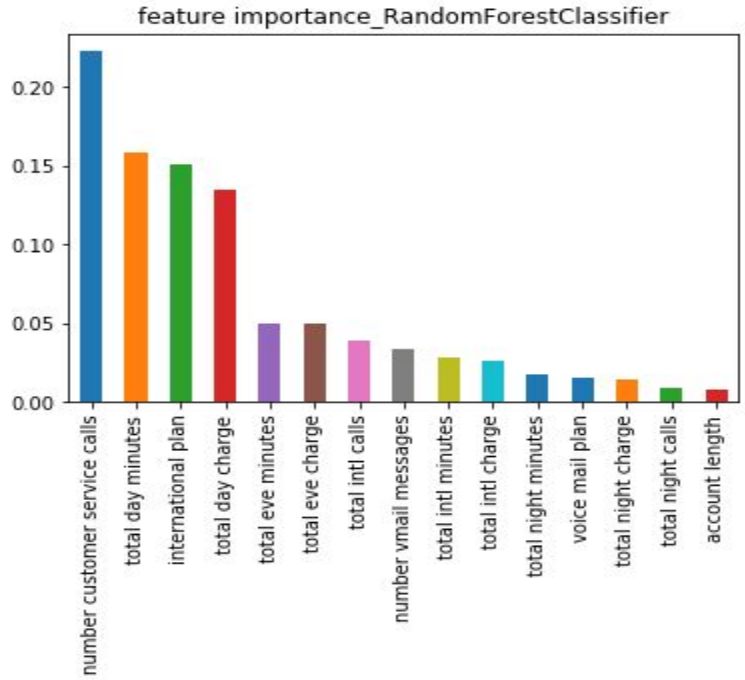
RANDOM FOREST CLASSIFIER:

Random Forests tend to perform better than decision trees, as instead of a single tree, multiple trees are built using a sample of features and data and the final prediction is decided by voting over all the trees.

After performing gridsearchCV, below hyperparameters were obtained which are then used to train the model.

Evaluation metrics for best model:
Train Metrics:
ROC_AUC : 0.9822982964668234
PRECISION : 0.8600533618767182
RECALL : 0.8649087574138786
F1-SCORE: 0.862451310479867
Test Metrics:
ROC_AUC : 0.9115231704261039
PRECISION : 0.8002935761863182
RECALL : 0.7971373812638995
F1-SCORE : 0.7983729104700269

The model further improved the Precision with Recall almost the same as compared to Decision Tree Algorithm. The F1 score improved to 0.798 from 0.72.
Below graph displays the feature importance for the top 15 variables.



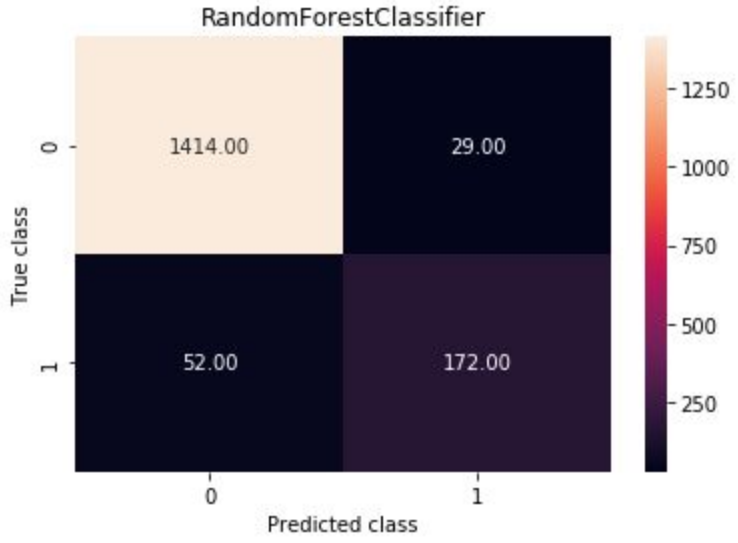
On implementing the random_forest model again with top 15 features, there was a slight improvement in recall and F1score,

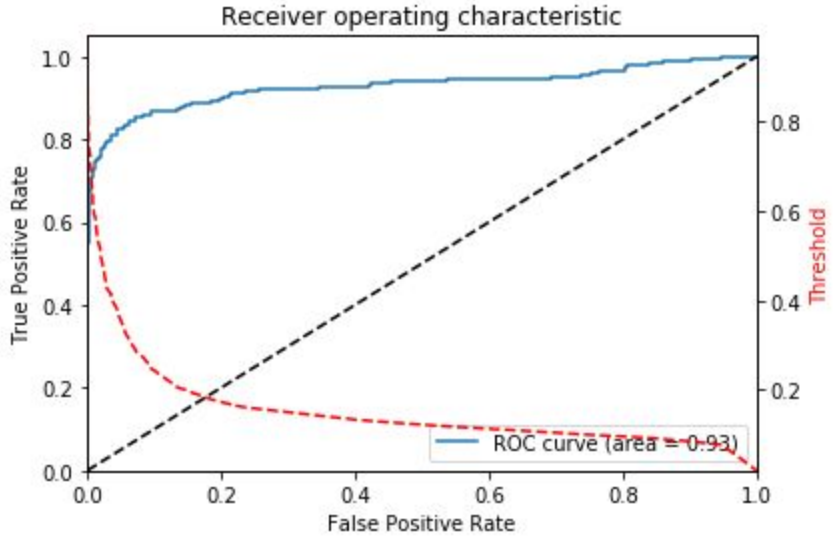
Evaluation metrics for best model with top 15 features:
Train Metrics:
ROC_AUC : 0.9808336827260475
PRECISION : 0.8497814238690579
RECALL : 0.8633583698169793
F1-SCORE: 0.8564851203745653
Test Metrics:
ROC_AUC : 0.9147024903249955
PRECISION : 0.7983705566790866
RECALL : 0.8116157620916731
F1-SCORE : 0.8045490916192107

So we will use only the top 15 featuers to evaluate the test set.

Test Set Report:

Algorithm used: RandomForestClassifier
ROC_AUC: 0.8738800613800614
RECALL: 0.7678571428571429
PRECISION: 0.8557213930348259
F1_SCORE: 0.8094117647058824
ACCURACY: 0.9514097180563887





There was an improvement in the F1 score from 0.736 to 0.809 as the Precision improved drastically with Recall almost remaining the same.

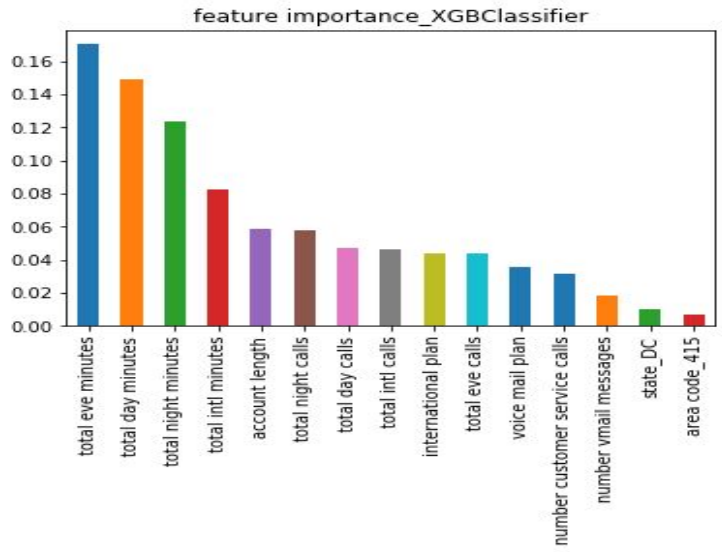
Gradient Boosting Algorithm - XGBOOST

Xtreme Gradient boosting follows the principle of gradient boosting and also performs regularization to help prevent overfitting.
It makes use of weak learners and is a sequential procedure where subsequent models add on the previous models predictions unlike random forest which is a parallel process.

'max_delta_step' and 'scale_pos_weight' are used to handle imbalance classes regularization parameters along with max_depth are used to prevent overfitting.

Evaluation metrics for best model:
Train Metrics:
ROC_AUC : 0.9990756959609264
PRECISION : 0.9070925817726708
RECALL : 0.9994818652849741
F1-SCORE: 0.9510234334458121
Test Metrics:
ROC_AUC : 0.910329096528752
PRECISION : 0.8168108086175747
RECALL : 0.826130873396618
F1-SCORE : 0.8209217545280686

XGBoost improves the F1 score to 0.821 which is better than that of Random Forests. Lets take a look at the best features.

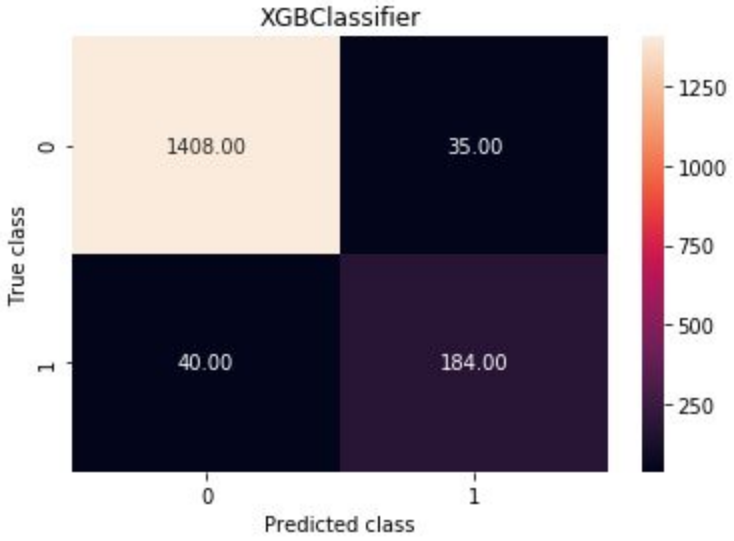


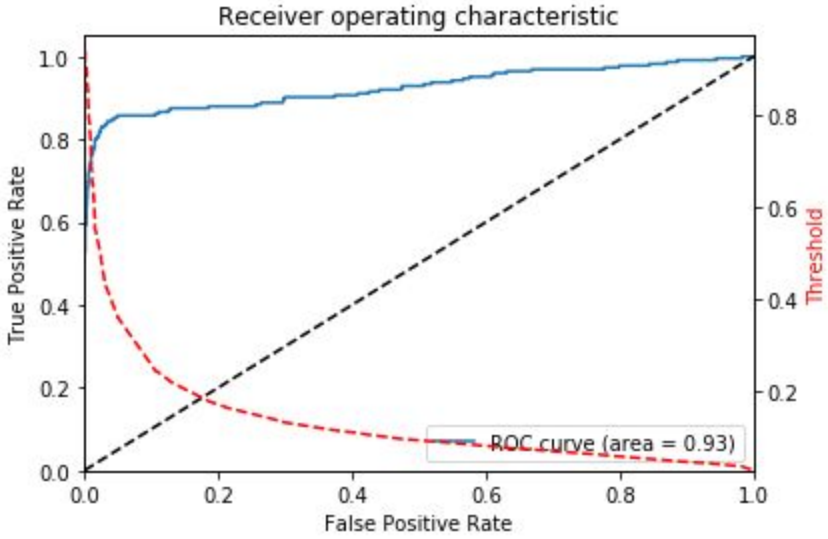
On implementing the model with the top 15 features, there was a marginal improvement, so with all the hyperparameters tuned to prevent overfitting all the features were used.

Evaluation metrics for best model with top 15 features:
Train Metrics:
ROC_AUC : 0.9990197253033374
PRECISION : 0.9077848888471077
RECALL : 0.9974106652742634
F1-SCORE: 0.9504620701190802
Test Metrics:
ROC_AUC : 0.9143121576129398
PRECISION : 0.8185832897247937
RECALL : 0.832356882595476
F1-SCORE : 0.8246009877187813

Test Set Report

Algorithm used: XGBClassifier
ROC_AUC: 0.8985867735867736
RECALL: 0.8214285714285714
PRECISION: 0.8401826484018264
F1_SCORE: 0.8306997742663657
ACCURACY: 0.9550089982003599





Above figures clearly shows that XGBoost model performs the best for Test dataset.

CONCLUSION:

Model Selection:

	F1-Score	Recall	Precision	ROC-AUC	Accuracy
Logistic Regression	0.483	0.808	0.344	0.785	0.767
Decision Trees	0.736	0.755	0.719	0.854	0.927
Random Forests	0.809	0.768	0.856	0.874	0.951
XGBoost	0.831	0.821	0.840	0.899	0.955

As we can observe from the above table, XGBoost classifier gives the best result for the test set. So we will select XGBoost as the final algorithm for Churn Rate Prediction.

The predicted values are saved in a csv file along with actual value and Phone_Numbers used as unique ID.

Phone_Number	Actual Churn	Predicted Churn
354-8815	0	0
381-7211	0	0
411-9481	0	0
418-9100	0	0
413-3643	0	0
375-6180	0	0
348-8073	0	0
359-9881	0	0
353-6954	0	0
403-4933	0	0
360-3811	0	0

APPENDIX A - Python Code

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt


# importing train and test data and storing them in dataframes
train_dataset= pd.read_csv('Train_data.csv')
test_dataset= pd.read_csv('Test_data.csv')
FinalResult_Python = test_dataset[:]


# Removing trailing and leading whitespaces and dots from the data
cat_var= [x for x in train_dataset.dtypes.index if train_dataset.dtypes[x]=='object']
train_dataset[cat_var]= train_dataset[cat_var].apply(lambda x: x.str.strip(' .'))
test_dataset[cat_var]= test_dataset[cat_var].apply(lambda x: x.str.strip(' .'))


# Descriptive statistics of all the numeric features
summary= train_dataset.describe()

# plotting histograms to observe the distributions for all numeric variables
features_numeric = train_dataset.select_dtypes(include = ['float64', 'int64'])
features_numeric.hist(figsize=(16, 20), bins=50, xlabelsize=8, ylabelsize=8);


# plotting bar charts for categorical variables with respect to their
# distribution in each target class
cat_pred = ['international plan','voice mail plan','area code','state']
for pred in cat_pred:
    crosstab_churn= pd.crosstab(index=train_dataset[pred],
columns=train_dataset['Churn'])
    crosstab_churn.reset_index(level=0, inplace= True)
    crosstab_churn[pred]=crosstab_churn[pred].astype('str')
```

```
    plot1=plt.bar(x=crosstab_churn[pred], height=(crosstab_churn['False']))
    plot2=plt.bar(x=crosstab_churn[pred], height=(crosstab_churn['True']))
    plt.xlabel(pred)
    plt.xticks(rotation=90)
    plt.ylabel('Churn Count')
    plt.legend((plot1[0],plot2[0]),('False','True'))
    plt.show()


# plotting boxplots for all numeric variables for each target class
features_numeric.drop(columns=['area code'], inplace=True)
fig, axes = plt.subplots(4,4, figsize=(15,25))
for i,pred in enumerate(features_numeric.columns):
    bxplt = train_dataset.boxplot(column=pred, by="Churn", ax=axes[int(i/4),i%4])
fig.delaxes(axes[3,3]) # remove empty subplot
fig.tight_layout()
plt.show()


# label encoding target class
train_dataset['Churn']= train_dataset['Churn'].map({'False':0, 'True':1})
test_dataset['Churn']= test_dataset['Churn'].map({'False':0, 'True':1})


# plotting correlation matrix heatmap for numeric predictors
corr= train_dataset.corr()
f, ax = plt.subplots(figsize=(12, 10))
sns.heatmap(corr, vmax=.5, square=True, annot=True, fmt='0.2f');


# combining dataset to ease preprocessing
train_dataset['source']='train'
test_dataset['source']='test'
dataset= pd.concat([train_dataset,test_dataset],axis=0)


# Finding out the number of missing values in each column
missing_values= dataset.apply(lambda x: sum(x.isnull()))
```

```
# label encoding the other binary predictors
map_dict= {'no':0, 'yes':1 ,}
predictors = ['voice mail plan', 'international plan']
for predictor in predictors:
    dataset.loc[:,predictor]=dataset[predictor].map(map_dict)


# creating dummy variabes for nominal variable 'state'
dataset=pd.get_dummies(dataset,columns=['state'],drop_first=True)

# converting and creating dummy variables for nominal variable 'area code'
dataset['area code']=dataset['area code'].astype(str)
dataset=pd.get_dummies(dataset,columns=['area code'],drop_first=True)


# dropping phone number from the dataset
dataset.drop(columns=['phone number'],inplace=True)


# separating the dataset after preprocessing
train_dataset= dataset[dataset['source']=='train']
test_dataset= dataset[dataset['source']=='test']


# separating the predictors and target variable for each train and test dataset
ytrain= train_dataset['Churn']
xtrain= train_dataset.drop(columns=['source','Churn'])
ytest= test_dataset['Churn']
xtest= test_dataset.drop(columns=['source','Churn'])


# storing all predictor names in a variable to be used later
predictors= [x for x in xtrain.columns]


# standardizing both the datasets using the mean and variance of train dataset
from sklearn.preprocessing import StandardScaler
```

```
sc= StandardScaler()
xtrain= sc.fit_transform(xtrain)
xtest= sc.transform(xtest)
xtrain=pd.DataFrame(data= xtrain, columns = predictors)
xtest=pd.DataFrame(data= xtest, columns = predictors)


'''
Applied PCA to reduce dimensions but the results did not improve,
so we continued with all the features
from sklearn.decomposition import PCA
pca= PCA(n_components=25, random_state=42)
xtrain_pca = pca.fit_transform(xtrain)
exp_var = pca.explained_variance_ratio_
xtest_pca = pca.transform(xtest)
'''


from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV


# importing warnings to prevent deprecated warnings dialogs from appearing in console
import warnings
warnings.filterwarnings(action='ignore', category=DeprecationWarning)


''' Generic function to cross validate and tune hyparameters. It will display the
 train and test scores for multiple evaluation metrics and returns best parameters '''
# algo - classifier object
# x_train - train dataset
```

```
# y_train - target variable corresponding to train dataset
# params - paramgrid containing different combinations to be tested using grid search
# useparams - boolean argument which on False sets param_grid to empty, so that only
cross
# validation is performed
def model_fit(algo, x_train, y_train, params, useparams):

    score_metrics= {'f1score':'f1','precision':'precision','recall':'recall','roc_auc':'roc_auc'}

    if(useparams==False):
        params={}

    gridsearch= GridSearchCV(algo, params, scoring=score_metrics, cv=5,
refit='f1score', verbose=1, return_train_score=True)
    gridsearch= gridsearch.fit(x_train, y_train)
    idx= np.argmax(gridsearch.cv_results_['mean_test_f1score'])


    print ("\nModel Report-",str(algo)[0:str(algo).find('(')])
    print("estimator:",gridsearch.best_estimator_)
    print()
    print("Best params:",gridsearch.best_params_)
    print()
    print("Evaluation metrics for best model:")
    print("Train Metrics:")
    print("ROC_AUC :",gridsearch.cv_results_['mean_train_roc_auc'][idx])
    print("PRECISION :",gridsearch.cv_results_['mean_train_precision'][idx])
    print("RECALL :",gridsearch.cv_results_['mean_train_recall'][idx])
    print("F1-SCORE:",gridsearch.cv_results_['mean_train_f1score'][idx])


    print("Test Metrics:")
    print("ROC_AUC :",gridsearch.cv_results_['mean_test_roc_auc'][idx])
    print("PRECISION :",gridsearch.cv_results_['mean_test_precision'][idx])
    print("RECALL :",gridsearch.cv_results_['mean_test_recall'][idx])
    print("F1-SCORE :",gridsearch.cv_results_['mean_test_f1score'][idx])


    return gridsearch.best_estimator_
```

```
''' Function to plot confusion matrix for test set '''
# algo - classifier object
# ytest - actual values for target variable corresponding to test dataset
# ypred - predicted values for target variable corresponding to test dataset
def plot_confusion_matrix(algo,ytest,ypred):

    conf_matrix=confusion_matrix(ytest, ypred)

    sns.heatmap(conf_matrix, fmt='0.2f',annot=True ,xticklabels = ["0", "1"] , yticklabels =
["0", "1"] )
    plt.ylabel('True class')
    plt.xlabel('Predicted class')
    plt.title(str(algo)[0:str(algo).find('(')])



''' Function to plot ROC_AUC curvealong with threshold for test set  '''
# ytrue - actual values for target variable corresponding to test dataset
# ypred - predicted probabilities for target variable corresponding to test dataset
def create_roc_curve(ytrue, ypred):


    fpr, tpr, thresholds = roc_curve(ytrue, ypred)
    auc_score= auc(fpr,tpr)
    plt.figure()
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % (auc_score))
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic')
    plt.legend(loc="lower right")

    # create the axis of thresholds (scores)
    ax2 = plt.gca().twinx()
```

```
    ax2.plot(fpr, thresholds, markeredgecolor='r',linestyle='dashed', color='r')
    ax2.set_ylabel('Threshold',color='r')
    ax2.set_ylim([thresholds[-1],thresholds[0]])
    ax2.set_xlim([fpr[0],fpr[-1]])
    plt.show()



''' generic function to print feature importance for the best 15 predictors '''
# algo - classifier object
# returns names of top 15 predictors
def print_feature_importance(algo):
    ftimp = pd.Series(algo.feature_importances_,
predictors).sort_values(ascending=False)
    ftimp=ftimp.nlargest(n=15)
    plt.figure()
    ftimp.plot(kind='bar', title='_'.join(['feature importance',str(algo)[0:str(algo).find('(')]]))
    return pd.Series(ftimp.index)



''' generic function to print Report for the test dataset for different models after the
 hyperparameters have been tuned '''
# algo - classifier object
# xtest - test dataset
# ytest - actual values for target variable corresponding to test dataset
# returns a Series containing predicted Churn values
def Test_Set_Report(algo,xtest,ytest):

    ypred= algo.predict(xtest)
    ypred_prob= algo.predict_proba(xtest)[:,1]

    print ("\nTest Set Report\n")
    print("Algorithm used:",str(algo)[0:str(algo).find('(')])
    print("ROC_AUC:",roc_auc_score(ytest, ypred))
    print("RECALL:",recall_score(ytest,ypred))
    print("PRECISION:",precision_score(ytest,ypred))
    print("F1_SCORE:",f1_score(ytest,ypred))
```

```
    print("ACCURACY:",accuracy_score(ytest,ypred))

    plot_confusion_matrix(algo, ytest, ypred)
    create_roc_curve(ytest, ypred_prob)

    return pd.Series(ypred)
```

```
''' Implementing logistic regression with L1 regularization'''

# using class_weight='balanced' to handle imbalanced classes
from sklearn.linear_model import LogisticRegression

logreg= LogisticRegression(class_weight='balanced', penalty='l1', C=0.05,
random_state=42)

param_logreg = {'C':[0.01,0.05,0.1,0.2,0.5,0.8,1,2,3]}
logreg=model_fit(logreg, xtrain, ytrain, param_logreg, False)

# train the model with best parameters achieved using gridsearch
logreg.fit(xtrain,ytrain)

# Evaluating the test set with best parameters and printing the final Report
ypred_logreg = Test_Set_Report(logreg,xtest, ytest)
```

```
''' Implementing Decision Trees '''

from sklearn.tree import DecisionTreeClassifier
```

```
dectree=
DecisionTreeClassifier(max_depth=4,min_samples_split=10,max_leaf_nodes=15,rando
m_state=42,class_weight='balanced')

param_tree= { 'max_leaf_nodes':[10,12,15], 'min_samples_split':[10,15,20,25]}
dectree=model_fit(dectree, xtrain, ytrain,param_tree, False)

dectree.fit(xtrain,ytrain)
bestpred_dectree = print_feature_importance(dectree)

ypred_dectree = Test_Set_Report(dectree,xtest, ytest)



''' Random Forest Algorithm '''

from sklearn.ensemble import RandomForestClassifier
ranfor=
RandomForestClassifier(n_estimators=45,max_depth=6,min_samples_split=20,max_le
af_nodes=50,random_state=42, max_features=0.5,class_weight='balanced')

param_ranfor=
{'min_samples_split':[20,30,40,50,60],'max_leaf_nodes':[20,30,40,50,60]}
ranfor=model_fit(ranfor, xtrain, ytrain,param_ranfor, False)

ranfor.fit(xtrain,ytrain)
bestpred_rf = print_feature_importance(ranfor)

ypred_ranfor = Test_Set_Report(ranfor,xtest, ytest)
```

```
''' Gradient Boosting - XGBoost '''

import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'


import xgboost as xgb
from xgboost.sklearn import XGBClassifier

xgbc= XGBClassifier(n_estimators=300,min_child_weight=1, learning_rate=0.01,
scale_pos_weight=9,gamma=3,reg_lambda=0.1,reg_alpha=2, max_depth=10,
max_delta_step=2, random_state=42)

param_xgb={'colsample_bytree':[0.7,0.8,0.9,1], 'subsample':[0.9,1]}
xgbc=model_fit(xgbc, xtrain, ytrain, param_xgb, False)

xgbc.fit(xtrain,ytrain)
bestpred_xgb = print_feature_importance(xgbc)

ypred_xgb = Test_Set_Report(xgbc,xtest, ytest)

# xgb.cv is used to get the actual number of n_estimators required based on the
learning rate,
# it uses early_stopping_rounds to get the optimal value
xdtrain = xgb.DMatrix(xtrain,label=ytrain)
cvresult_xgb=xgb.cv(xgbc.get_xgb_params(),
xdtrain,nfold=5,num_boost_round=5000,metrics='auc',early_stopping_rounds=50)


''' Storing the predicted results along with the actual prediction for each phone number in
a csv file'''
FinalResult_Python = pd.concat([FinalResult_Python,ypred_xgb], axis=1)
FinalResult_Python.rename(columns={0:'Predicted Churn'}, inplace=True)
FinalResult_Python['Predicted Churn']=FinalResult_Python['Predicted
Churn'].map({0:'False',1:'True'})
FinalResult_Python.to_csv("PredictedChurn_Python.csv", index=False)
```

APPENDIX B - R CODE

```
rm(list = ls())

# set working directory
setwd("Downloads/Data Science Problem sets/ChurnRate")

# importing test and train data into dataframes by and removing whitespaces
train_dataset = read.csv("Train_Data.csv", header = T, strip.white= T )
test_dataset= read.csv("Test_Data.csv", header = T, strip.white= T )
FinalResult_R <- cbind(NA, NA)
FinalResult_R <- cbind(test_dataset)


# renaming column names
names(train_dataset) = gsub("\\.", "_", names(train_dataset))
names(test_dataset) = gsub("\\.", "_", names(test_dataset))


# getting the descriptive statistics of the dataset
summary(train_dataset)

# label encoding target class
levels(train_dataset$Churn) = c(0:(length(levels(train_dataset$Churn))-1))
levels(test_dataset$Churn) = c(0:(length(levels(test_dataset$Churn))-1))

# correlation matrix of all numeric predictors and target variable
dataset_corr = data.frame(lapply(train_dataset, function(x) {
  if(is.integer(x)) as.numeric(x) else  x}))
dataset_corr$Churn = as.numeric(as.character(dataset_corr$Churn))
correlation_matrix = cor(dataset_corr[sapply(dataset_corr, is.numeric)], method =
'pearson')
```

```
# histograms to observe distribution for all numeric predictors
df_columns = data.frame("colType"=sapply(dataset_corr, function (x) class(x)))
df_columns$colName = rownames(df_columns)
df_columns = df_columns[-c(21),]
rownames(df_columns) = NULL


nCol = nrow(df_columns)
for(i in 1:nCol){
  if(df_columns[i,1]=="numeric"){
        hist(train_dataset[,df_columns[i,2]],
         breaks=50,
         main="HISTOGRAM",
         xlab=paste(df_columns[i,2],sep=""),
         ylab="Count",
         col="green",
         border="red" )
  }
}


# plotting boxplots for all numeric variables for each target class
df_columns = df_columns[-c(3),]
nCol = nrow(df_columns)
for(i in 1:nCol){
  if(df_columns[i,1]=="numeric"){
    boxplot(train_dataset[,df_columns[i,2]] ~ Churn,
            data = train_dataset,
            xlab='Churn',
            ylab=paste(df_columns[i,2],sep=""))
  }
}


# plotting bar charts for categorical variables with respect to their
# distribution in each target class
cat_Features = c('international_plan','voice_mail_plan','state','area_code')
for(feature in cat_Features){
```

```
crosstab_churn = table(train_dataset$Churn, train_dataset[[feature]])
barplot(crosstab_churn, main="Distribution per Target Class",
        xlab=feature, col=c("red","green"),
        ylab ='Churn',
        legend = rownames(crosstab_churn))
}

# adding a column 'Source' to distinguish between test and train data
train_dataset$Source = 'train'
test_dataset$Source = 'test'

# combining test and train dataset to preprocess both the dataset at once
dataset = rbind(train_dataset, test_dataset)

# finding the count of missing values
missing_values = data.frame(c(sapply(dataset, function(x)  sum(is.na(x)))))
colnames(missing_values) = 'Count'

# label encoding factor variables
map_value = c('voice_mail_plan', 'international_plan')
for (feature in map_value)
{
  levels(dataset[[feature]]) = c(0:(length(levels(dataset[[feature]]))-1))
  if(feature != 'Churn') dataset[[feature]] = as.numeric(as.character(dataset[[feature]]))
}

# creating dummy variables for both state and area code and removing the original
columns
for(unique_value in unique(dataset$state)){
  dataset[paste("state", unique_value, sep = ".")] = ifelse(dataset$state == unique_value,
1, 0)
}

dataset$area_code= as.factor(dataset$area_code)
for(unique_value in unique(dataset$area_code)){
```

```
  dataset[paste("area_code", unique_value, sep = ".")] = ifelse(dataset$area_code ==
unique_value, 1, 0)
}
dataset = within(dataset, rm(state,area_code))


# remove 'phone.number' column as it is unique for every entry and thus adding no
information to the model
# length(unique(dataset$phone.number)) = 5000
dataset = within(dataset, rm(phone_number))


# splitting the dataset after preprocessing back to train_dataset and test_dataset
train_dataset = dataset[dataset$Source=='train',]
test_dataset = dataset[dataset$Source=='test',]

# Removing temporary variable source
train_dataset = within(train_dataset, rm(Source))
test_dataset = within(test_dataset, rm(Source))

# Moving Churn column to the end of dataframe
train_dataset =
train_dataset[,c(colnames(train_dataset)[colnames(train_dataset)!='Churn'],'Churn')]
test_dataset =
test_dataset[,c(colnames(test_dataset)[colnames(test_dataset)!='Churn'],'Churn')]


# normalizing the train and test data using preprocess function of caret package
pp = preProcess(train_dataset[-72], method=c('center','scale'))
train_dataset = predict(pp,train_dataset)
test_dataset = predict(pp, test_dataset)


# creating training and testing tasks
traintask = makeClassifTask(data= train_dataset , target= 'Churn' , positive = '1')
testtask = makeClassifTask(data= test_dataset , target= 'Churn' , positive = '1')

# creating a weight vector to for each observation to handle imbalance classes
mask = train_dataset['Churn']=='1'
```

```
obs_weight = sapply(mask, function(x) if(x==T) 9 else 1)

# creating weighted training task
traintask_weighted = makeClassifTask(data= train_dataset , target= 'Churn' , positive =
'1', weight= obs_weight)


# generic function for tuning hyperparameters using gridsearch
model_tuning <- function(algoname, algo, params_tune, traintask ){
  set.seed(42)
  gridsearch = tuneParams(algo , resampling = makeResampleDesc("CV",iters = 5L,
stratify = T, predict = "both"),
                          task= traintask, par.set = params_tune,
                          measures = list(f1,tpr,ppv,auc,setAggregation(f1,
train.mean),setAggregation(tpr, train.mean),setAggregation(ppv,
train.mean),setAggregation(auc, train.mean)) ,
                          control = makeTuneControlGrid()
  )

  cat("\nGrid Search Report-",algoname,"\n")
  cat("Best params:\n")
  print(as.matrix(gridsearch$x))
  cat("Optimum threshold:", gridsearch$threshold)
  cat("\nEvaluation metrics for best model:Train Fold\n")
  cat("F1-Score :",gridsearch$y[5],"\n")
  cat("ROC_AUC :",gridsearch$y[8],"\n")
  cat("RECALL :",gridsearch$y[6],"\n")
  cat("PRECISION :",gridsearch$y[7],"\n")
  cat("\nEvaluation metrics for best model:Test Fold\n")
  cat("F1-Score :",gridsearch$y[1],"\n")
  cat("ROC_AUC :",gridsearch$y[4],"\n")
  cat("RECALL :",gridsearch$y[2],"\n")
  cat("ACCURACY :",gridsearch$y[3],"\n")
  return(gridsearch$x)
}


# generic function to display cross validation scores for train dataset
cross_validation <- function(algoname, algo, traintask){
```

```
  set.seed(42)
  r=resample(algo, traintask, makeResampleDesc("CV",iters=5, predict="both", stratify =
T),
             measures = list(f1,tpr,ppv,auc,setAggregation(f1, train.mean),
setAggregation(tpr, train.mean),setAggregation(ppv, train.mean),setAggregation(auc,
train.mean)))

  cat("\nResampling Report-",algoname,"\n")
  cat("\nEvaluation metrics:Train Fold \n")
  cat("F1-Score :",r$aggr[5],"\n")
  cat("ROC_AUC :",r$aggr[8],"\n")
  cat("RECALL :",r$aggr[6],"\n")
  cat("PRECISION :",r$aggr[7],"\n")
  cat("\nEvaluation metrics:Test Fold \n")
  cat("F1-Score :",r$aggr[1],"\n")
  cat("ROC_AUC :",r$aggr[4],"\n")
  cat("RECALL :",r$aggr[2],"\n")
  cat("PRECISION :",r$aggr[3],"\n")
}


# function to generate feature importance graph
featureImportance <- function(algoname, algo, traintask){
  ftimp = data.frame(t((getFeatureImportance(train(algo, traintask)))$res))
  setDT(ftimp, keep.rownames = TRUE)[]
  colnames(ftimp)= c('Feature','Importance')
  ftimp = ftimp[order(ftimp$Importance, decreasing = T),]
  ftimp = ftimp[1:15,]
  ftimp = ftimp[order(ftimp$Importance, decreasing = F),]
  level_order = ftimp$Feature
  gg = ggplot(ftimp, aes(x=factor(Feature, level =
level_order),y=Importance))+geom_col()+labs(title=algoname, x='Feature')+coord_flip()
  print(gg)
  return(ftimp$Feature)
}


# generic function to evaluate the test set using optimal parameters
Test_Set_Report <- function (algoname, algo, traintask, testtask){
```

```
  model_fit = train(algo, traintask)
  ypred = predict(model_fit , testtask)
  conf_matrix = table(test_dataset$Churn, ypred$data$response)
  recall_score = conf_matrix[2,2]/ (conf_matrix[2,1]+conf_matrix[2,2])
  precision_score = conf_matrix[2,2]/(conf_matrix[1,2]+conf_matrix[2,2])
  accuracy = sum(diag(conf_matrix))/ sum(conf_matrix)
  auc=performance(ypred,auc)
  f1= performance(ypred,f1)

  cat("\nTest Set Report-",algoname,"\n")
  cat("Confusion_Matrix:\n")
  print(conf_matrix)
  cat("\nEvaluation metrics:\n")
  cat("F1-Score :",f1,"\n")
  cat("ROC_AUC :",auc,"\n")
  cat("RECALL :",recall_score,"\n")
  cat("PRECISION :",precision_score,"\n")
  cat("ACCURACY :",accuracy,"\n")

  df = generateThreshVsPerfData(ypred, measures = list(fpr, tpr))
  gg= plotROCCurves(df)
  print(gg)

  return(data.frame(ypred$data$response))
}
```

```
#### LOGISTIC REGRESSION ####
logreg_model = makeLearner("classif.logreg", predict.type='prob' )

# cross validation report
cross_validation("Logistic Regression", logreg_model, traintask_weighted )

# evaluating the test set performance
ypred_logreg = Test_Set_Report("Logistic Regression", logreg_model,
traintask_weighted, testtask)
```

```
#### Regularized Logistic Regression with L1 Penalty ####
classweights= c(1,6)
names(classweights)=c('0','1')
ridge_model = makeLearner("classif.LiblineaRL1LogReg", predict.type = 'prob', par.vals
= list(cost=0.5, wi=classweights))

# cross validation report
cross_validation("Regularized Logistic Regression", ridge_model, traintask)

# evaluating the test set performance
ypred_ridge = Test_Set_Report("Regularized Logistic Regression", ridge_model,
traintask, testtask)
```

```
#### DECISION TREE CLASSIFIER ####

# creating a learner
dectree_model = makeLearner("classif.rpart" , predict.type = "prob", par.vals=
list(minbucket=50,maxdepth=4, minsplit=10), parms= list(prior=c(0.4,0.6)))

#Search for hyperparameters
params_dectree = makeParamSet(
  makeDiscreteParam("minsplit",values=c(20)),
  makeDiscreteParam("minbucket", values=c(50)),
  makeDiscreteParam("maxdepth", values=c(3)),
  makeDiscreteParam("parms", values= list(a=list(prior=c(0.4,0.6))))
)

# perform grid search to get optimal parameters
best_params_dectree = model_tuning("Decision Tree", dectree_model,
params_dectree, traintask)

# cross validation report using optimal parameters
dectree_model = setHyperPars(dectree_model , par.vals = best_params_dectree)
cross_validation("Decison Tree", dectree_model, traintask)

# plot feature importance for decision trees
```

```
bestpred_dectree= featureImportance("Decision Tree",dectree_model, traintask)
colnames(bestpred_dectree)= 'Best_Features'

# evaluating the test set performance
ypred_dectree = Test_Set_Report("Decision Tree", dectree_model, traintask, testtask)
```

```
#### RANDOM FOREST CLASSIFIER ####
ranfor_model = makeLearner("classif.randomForest" , predict.type = "prob", par.vals=
list(ntree= 45,mtry=35, classwt=c(0.30,0.70), nodesize= 8, maxnodes= 50,
importance=T))

#Search for hyperparameters
params_ranfor = makeParamSet(
  makeDiscreteParam("ntree",values=c(55,58,63,65)),
  makeDiscreteParam('mtry', values=c(5,6,8,10)),
  makeDiscreteParam("nodesize", values=c(4,5,6)),
  makeDiscreteParam("maxnodes", values=c(43,45,47))
)

# perform grid search to get optimal parameters
best_params_ranfor = model_tuning("Random Forest", ranfor_model, params_ranfor,
traintask)

# cross validation report using optimal parameters
ranfor_model = setHyperPars(ranfor_model , par.vals = best_params_ranfor)
cross_validation("Random Forest", ranfor_model, traintask)

# plot feature importance for random forest
bestpred_rf= featureImportance("Random Forest",ranfor_model, traintask)
colnames(bestpred_rf)= 'Best_Features'

# evaluating the test set performance
ypred_ranfor= Test_Set_Report("Random Forest", ranfor_model, traintask, testtask)
```

```
#### XGBOOST CLASSIFIER ####


#make learner with inital parameters
xgb_model <- makeLearner("classif.xgboost", predict.type = "prob")
xgb_model$par.vals <- list(
objective = "binary:logistic",
nrounds = 500,
eta= 0.01,
subsample= 1,
colsample_bytree= 1,
eval_metric="auc",
early_stopping_rounds=50,
verbose=1,
print_every_n = 25,
max_depth= 10,
min_child_weight = 1,
gamma = 3,
alpha = 2,
lambda = 0.1,
max_delta_step = 2
)


#Search for hyperparameters
params_xgb = makeParamSet(
  makeDiscreteParam("max_depth",values=c(5)),
  makeDiscreteParam("gamma",values=c(0)),
  makeDiscreteParam('min_child_weight', values=c(1)),
  makeDiscreteParam("subsample", values=c(1)),
  makeDiscreteParam("colsample_bytree", values=c(0.8)),
  makeDiscreteParam("max_delta_step", values=c(0.32)),
  makeDiscreteParam("alpha", values=c(0.1,0.5,1)),
  makeDiscreteParam("lambda", values=c(0.1,0.5,1))
  )

# perform grid search to get optimal parameters
```

```
best_params_xgb = model_tuning("XGBoost", xgb_model, params_xgb,
traintask_weighted)

# cross validation report using optimal parameters
xgb_model = setHyperPars(xgb_model , par.vals = best_params_xgb)
cross_validation("XGBoost", xgb_model, traintask_weighted)


# plot feature importance for random forest
bestpred_xgb = data.frame(featureImportance("XGBoost",xgb_model,
traintask_weighted))
colnames(bestpred_xgb)= 'Best_Features'

# evaluating the test set performance
ypred_xgb = Test_Set_Report("XGBoost", xgb_model, traintask_weighted, testtask)


### Storing the predicted results along with the actual prediction for each phone
#number in a csv file
FinalResult_R = cbind(FinalResult_R, ypred_xgb)
colnames(FinalResult_R)[22]= c( 'Predicted.Churn')
levels(FinalResult_R$`Predicted Churn`) = c('False', 'True')
write.csv(FinalResult_R, file="PredictedChurn_R.csv", row.names = F)
```