FEBRUARY 4, 2018

SAFE DRIVER PREDICTION

PRESENTED BY: YASH MITTAL
PRESENTED TO: EDWISOR.COM

CONTENTS

1. Problem Statement :

Nothing ruins the thrill of buying a brand new car more quickly than seeing your new insurance bill. The sting's even more painful when you know you're a good driver. It doesn't seem fair that you have to pay so much if you've been cautious on the road for years.
Porto Seguro, one of Brazil's largest auto and homeowner insurance companies, completely agrees. Inaccuracies in car insurance company's claim predictions raise the cost of insurance for good drivers and reduce the price for bad ones.
In this competition, we're challenged to build a model that predicts the probability that a driver will initiate an auto insurance claim in the next year. While Porto Seguro has used machine learning for the past 20 years, they're looking to Kaggle's machine learning community to explore new, more powerful methods. A more accurate prediction will allow them to further tailor their prices, and hopefully make auto insurance coverage more accessible to more drivers.

In this competition, we will predict the probability that an auto insurance policy holder files a claim.

2. Data Used:

We are provided with 2 csv files one for training data and other for testing data. These contain following columns:

```
id                  int64
target              int64
ps_ind_01           uint8
ps_ind_02_cat       uint8
ps_ind_03           uint8
ps_ind_04_cat       uint8
ps_ind_05_cat       uint8
ps_ind_06_bin       uint8
ps_ind_07_bin       uint8
ps_ind_08_bin       uint8
ps_ind_09_bin       uint8
ps_ind_10_bin       uint8
ps_ind_11_bin       uint8
ps_ind_12_bin       uint8
ps_ind_13_bin       uint8
ps_ind_14           uint8
ps_ind_15           uint8
ps_ind_16_bin       uint8
ps_ind_17_bin       uint8
ps_ind_18_bin       uint8
ps_reg_01         float64
ps_reg_02         float64
ps_reg_03         float64
ps_car_01_cat       uint8
ps_car_02_cat       uint8
ps_car_03_cat       uint8
ps_car_04_cat       uint8
ps_car_05_cat       uint8
ps_car_06_cat       uint8
ps_car_07_cat       uint8
ps_car_08_cat       uint8
ps_car_09_cat       uint8
ps_car_10_cat       uint8
ps_car_11_cat       uint8
ps_car_11           uint8
ps_car_12         float64
ps_car_13         float64
ps_car_14         float64
ps_car_15         float64
```

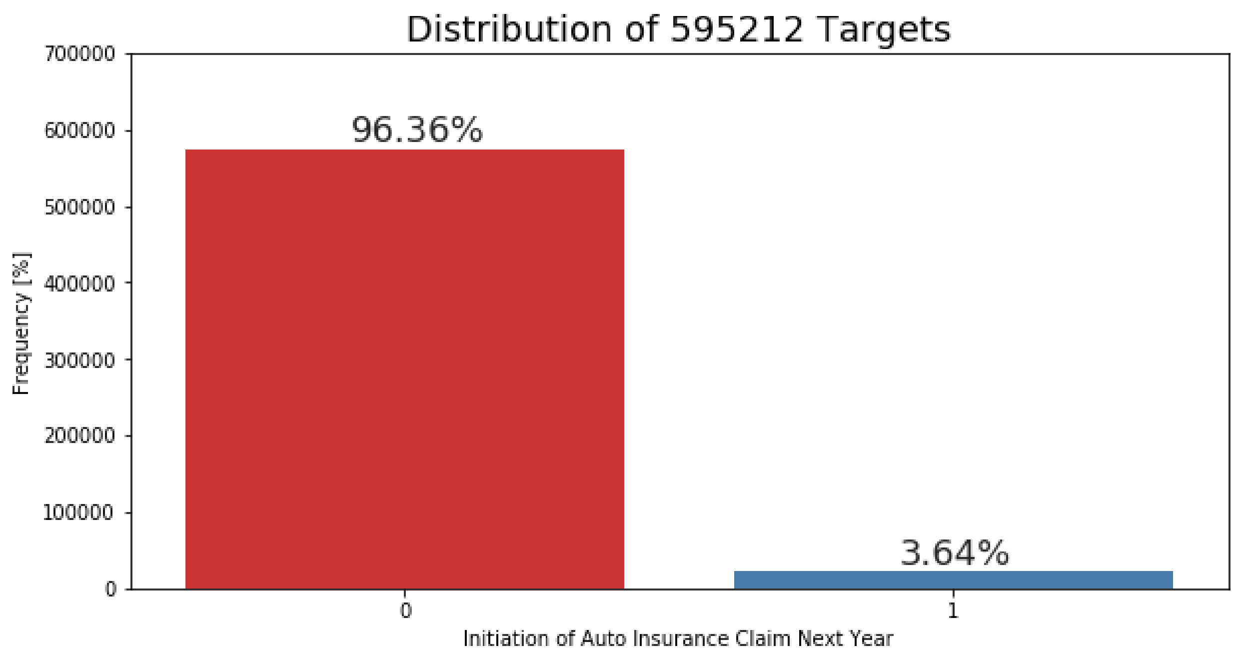
Test data doesn’t contain target variable.
In the train and test data, features that belong to similar groupings are tagged as such in the feature names (e.g., ind, reg, car, calc). In addition, feature names include the postfix bin to indicate binary features and cat to indicate categorical features. Features without these designations are either continuous or ordinal. Values of -1 indicate that the feature was missing from

the observation. The target columns signifies whether or not a claim was filed for that policy holder. Total 595212 rows are there in training data.

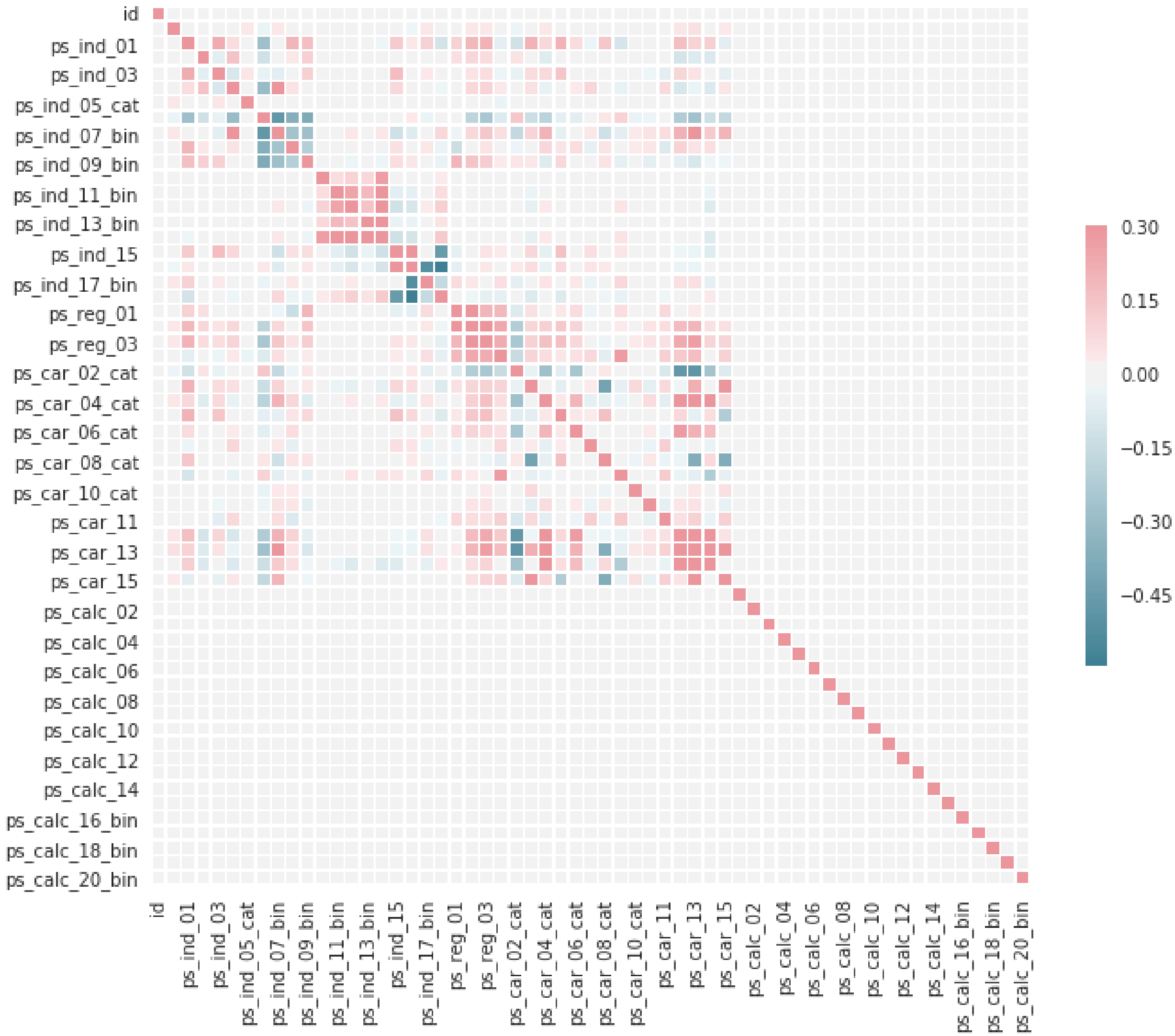
3. Exploration of Numerical variables:

3.1. Distribution of target variable:

Initially we will check our first variable that is target variable. We can see that it has most of the value as zero and only 3.64% values as 1 in our training data.



3.2. Checking correlation between variables:



We can see all ps_calc* variables are not related to others at all. We will remove them completely to have a better prediction.

3.3. Checking percent of null values:

Training data:

```
id 0.0
target 0.0
ps_ind_01 0.0
ps_ind_02_cat 0.0362895909357
ps_ind_03 0.0
ps_ind_04_cat 0.0139446113318
ps_ind_05_cat 0.975954785858
ps_ind_06_bin 0.0
ps_ind_07_bin 0.0
ps_ind_08_bin 0.0
ps_ind_09_bin 0.0
ps_ind_10_bin 0.0
ps_ind_11_bin 0.0
ps_ind_12_bin 0.0
ps_ind_13_bin 0.0
ps_ind_14 0.0
ps_ind_15 0.0
ps_ind_16_bin 0.0
ps_ind_17_bin 0.0
ps_ind_18_bin 0.0
ps_reg_01 0.0
ps_reg_02 0.0
ps_reg_03 18.1064897885
ps_car_01_cat 0.0179767881024
ps_car_02_cat 0.000840036827215
ps_car_03_cat 69.0898368984
ps_car_04_cat 0.0
ps_car_05_cat 44.7825312662
ps_car_06_cat 0.0
ps_car_07_cat 1.93023662157
ps_car_08_cat 0.0
ps_car_09_cat 0.095596190937
ps_car_10_cat 0.0
ps_car_11_cat 0.0
ps_car_11 0.000840036827215
ps_car_12 0.000168007365443
ps_car_13 0.0
ps_car_14 7.16047391518
ps_car_15 0.0
ps_calc_01 0.0
ps_calc_02 0.0
ps_calc_03 0.0
ps_calc_04 0.0
ps_calc_05 0.0
ps_calc_06 0.0
ps_calc_07 0.0
ps_calc_08 0.0
ps_calc_09 0.0
ps_calc_10 0.0
```

```
ps_calc_11 0.0
ps_calc_12 0.0
ps_calc_13 0.0
ps_calc_14 0.0
ps_calc_15_bin 0.0
ps_calc_16_bin 0.0
ps_calc_17_bin 0.0
ps_calc_18_bin 0.0
ps_calc_19_bin 0.0
ps_calc_20_bin 0.0
```

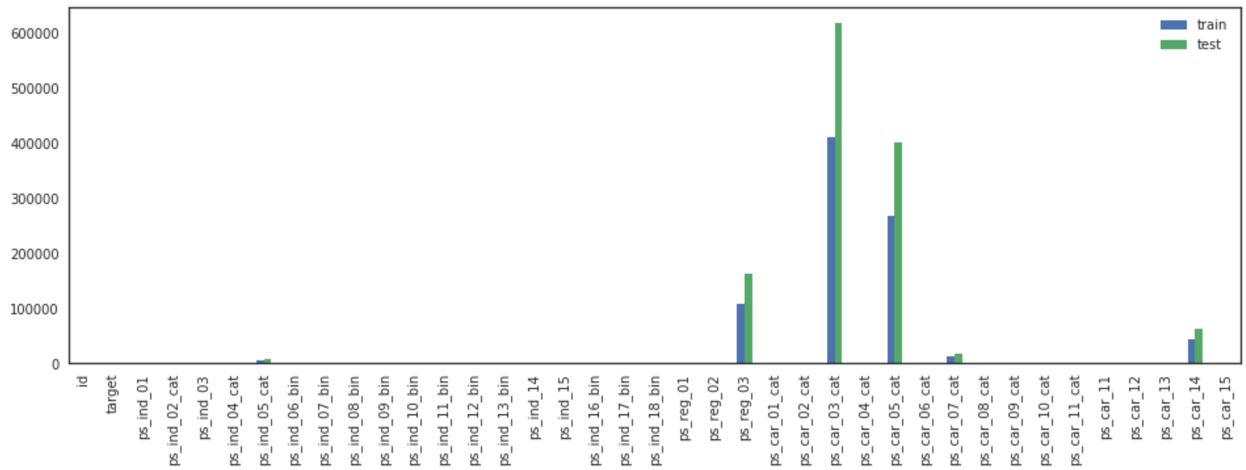
Testing data:

```
id 0.0
ps_ind_01 0.0
ps_ind_02_cat 0.0343855844877
ps_ind_03 0.0
ps_ind_04_cat 0.0162407483737
ps_ind_05_cat 0.975564954033
ps_ind_06_bin 0.0
ps_ind_07_bin 0.0
ps_ind_08_bin 0.0
ps_ind_09_bin 0.0
ps_ind_10_bin 0.0
ps_ind_11_bin 0.0
ps_ind_12_bin 0.0
ps_ind_13_bin 0.0
ps_ind_14 0.0
ps_ind_15 0.0
ps_ind_16_bin 0.0
ps_ind_17_bin 0.0
ps_ind_18_bin 0.0
ps_reg_01 0.0
ps_reg_02 0.0
ps_reg_03 18.1094424831
ps_car_01_cat 0.0179208257917
ps_car_02_cat 0.000560025805989
ps_car_03_cat 69.0972159997
ps_car_04_cat 0.0
ps_car_05_cat 44.842274332
ps_car_06_cat 0.0
ps_car_07_cat 1.94116144872
ps_car_08_cat 0.0
ps_car_09_cat 0.0982285263705
ps_car_10_cat 0.0
ps_car_11_cat 0.0
ps_car_11 0.000112005161198
ps_car_12 0.0
ps_car_13 0.0
ps_car_14 7.14648931023
ps_car_15 0.0
ps_calc_01 0.0
```

```
ps_calc_02 0.0
ps_calc_03 0.0
ps_calc_04 0.0
ps_calc_05 0.0
ps_calc_06 0.0
ps_calc_07 0.0
ps_calc_08 0.0
ps_calc_09 0.0
ps_calc_10 0.0
ps_calc_11 0.0
ps_calc_12 0.0
ps_calc_13 0.0
ps_calc_14 0.0
ps_calc_15_bin 0.0
ps_calc_16_bin 0.0
ps_calc_17_bin 0.0
ps_calc_18_bin 0.0
ps_calc_19_bin 0.0
ps_calc_20_bin 0.0
```

3.4 Checking for missing values

Visualization of missing data:



Treatment of missing values:

```
def missing_value(df):
    col = df.columns
    for i in col:
        if df[i].isnull().sum()>0:
            df[i].fillna(df[i].mode()[0],inplace=True)
```

We have replaced missing values with the mode of the variable by above code.

3.5. Checking for outliers:

We have 4 numerical variables 'ps_reg_03', 'ps_car_12', 'ps_car_13', 'ps_car_14'. We can do exploratory analysis on them:

```
train['ps_reg_03'].describe()
count    595212.000000
mean          0.846950
std           0.328237
min           0.061237
25%           0.633936
50%           0.720677
75%           1.000000
max           4.037945
Name: ps_reg_03, dtype: float64
```

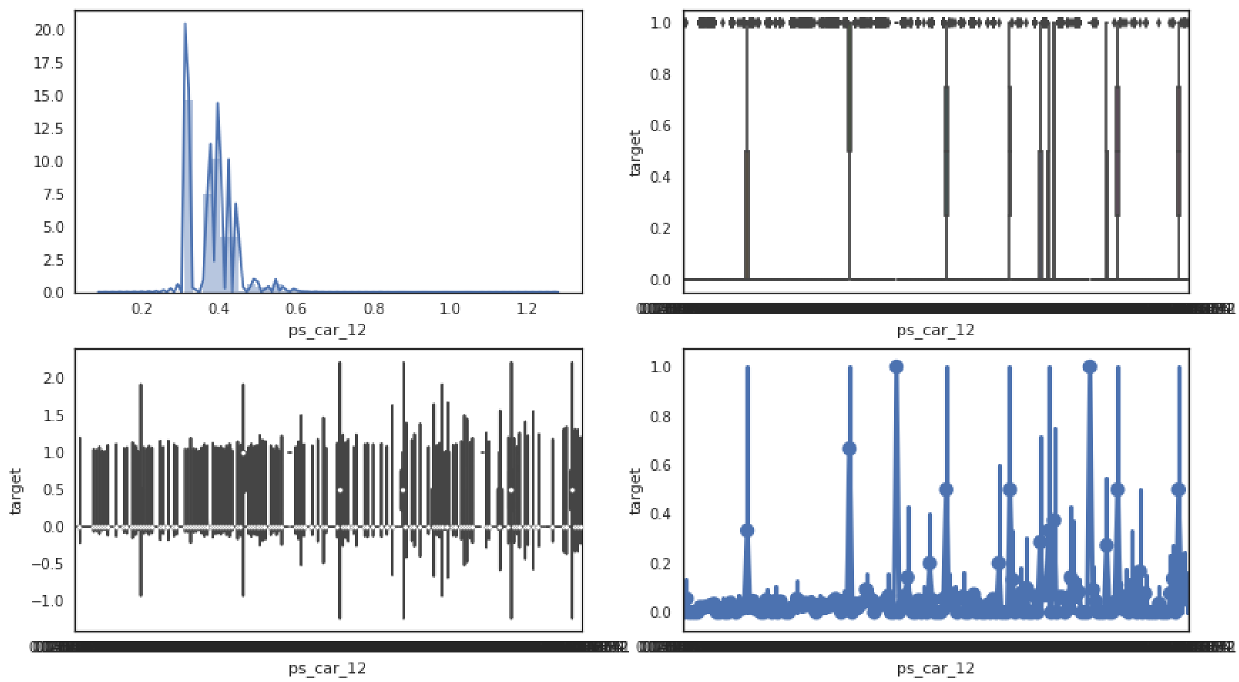
```
train['ps_car_12'].describe()

count    595212.000000
mean          0.379947
std           0.058300
min           0.100000
25%           0.316228
50%           0.374166
75%           0.400000
max           1.264911
Name: ps_car_12, dtype: float64
```

```
train['ps_car_13'].describe()

count    595212.000000
mean          0.813265
std           0.224588
min           0.250619
25%           0.670867
50%           0.765811
75%           0.906190
max           3.720626
Name: ps_car_13, dtype: float64
```

```
train['ps_car_14'].describe()

count    595212.000000
mean          0.373748
std           0.044078
```

```
min            0.109545
25%            0.353553
50%            0.368782
75%            0.396485
max            0.636396
Name: ps_car_14, dtype: float64
```

We tried making some plots of the variable to get some more insights of our variables. We can see in image upper left is distribution plot, upper right is box plot, lower left is violin plot and lower right is point plot.



There are outliers in numerical variables so we have to do the treatment of those variables. We treat it in the following way:

```
def outlier(df,columns):
    for i in columns:
        quartile_1,quartile_3 = np.percentile(df[i],[25,75])
        quartile_f,quartile_l = np.percentile(df[i],[1,99])
        IQR = quartile_3-quartile_1
        lower_bound = quartile_1 - (1.5*IQR)
        upper_bound = quartile_3 + (1.5*IQR)
        print(i,lower_bound,upper_bound,quartile_f,quartile_l)
        df[i].loc[df[i] < lower_bound] = quartile_f
```

```
        df[i].loc[df[i] > upper_bound] = quartile_1

outlier(train,num_col)
outlier(test,num_col)
```

We convert all variables except 'ps_reg_03', 'ps_car_12', 'ps_car_13', 'ps_car_14' into uint8. It would be easier to process them with uint8.

4. Exploration of Categorical Variables:

Our target variable is a categorical variable. We will convert it into category type and drop id column from both training and testing data.

```
X = train.drop(['target','id'],axis=1)
y = train['target'].astype('category')
x_test = test.drop('id',axis=1)
```

5. Building Predictive Models:

5.1. XGBoost Classifier:

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. The library is laser focused on computational speed and model performance, as such there are few frills. Nevertheless, it does offer a number of advanced features. We have defined one function for it as:

```
def runXGB(xtrain,xvalid,ytrain,yvalid,xtest,eta=0.1,num_rounds=100,max_de
pth=4):
    params = {
        'objective':'binary:logistic',
        'max_depth':max_depth,
        'learning_rate':eta,
        'eval_metric':'auc',
        'min_child_weight':6,
        'subsample':0.8,
        'colsample_bytree':0.8,
        'seed':45,
        'reg_lambda':1.3,
        'reg_alpha':8,
        'gamma':10,
        'scale_pos_weight':1.6
        #'n_thread':-1
    }

    dtrain = xgb.DMatrix(xtrain,label=ytrain)
    dvalid = xgb.DMatrix(xvalid,label=yvalid)
    dtest = xgb.DMatrix(xtest)
    watchlist = [(dtrain,'train'),(dvalid,'test')]

    model = xgb.train(params,dtrain,num_rounds,watchlist,early_stopping_ro
unds=50,verbose_eval=50)
    pred = model.predict(dvalid,ntree_limit=model.best_ntree_limit)
    pred_test = model.predict(dtest,ntree_limit=model.best_ntree_limit)
    return pred_test,model
```

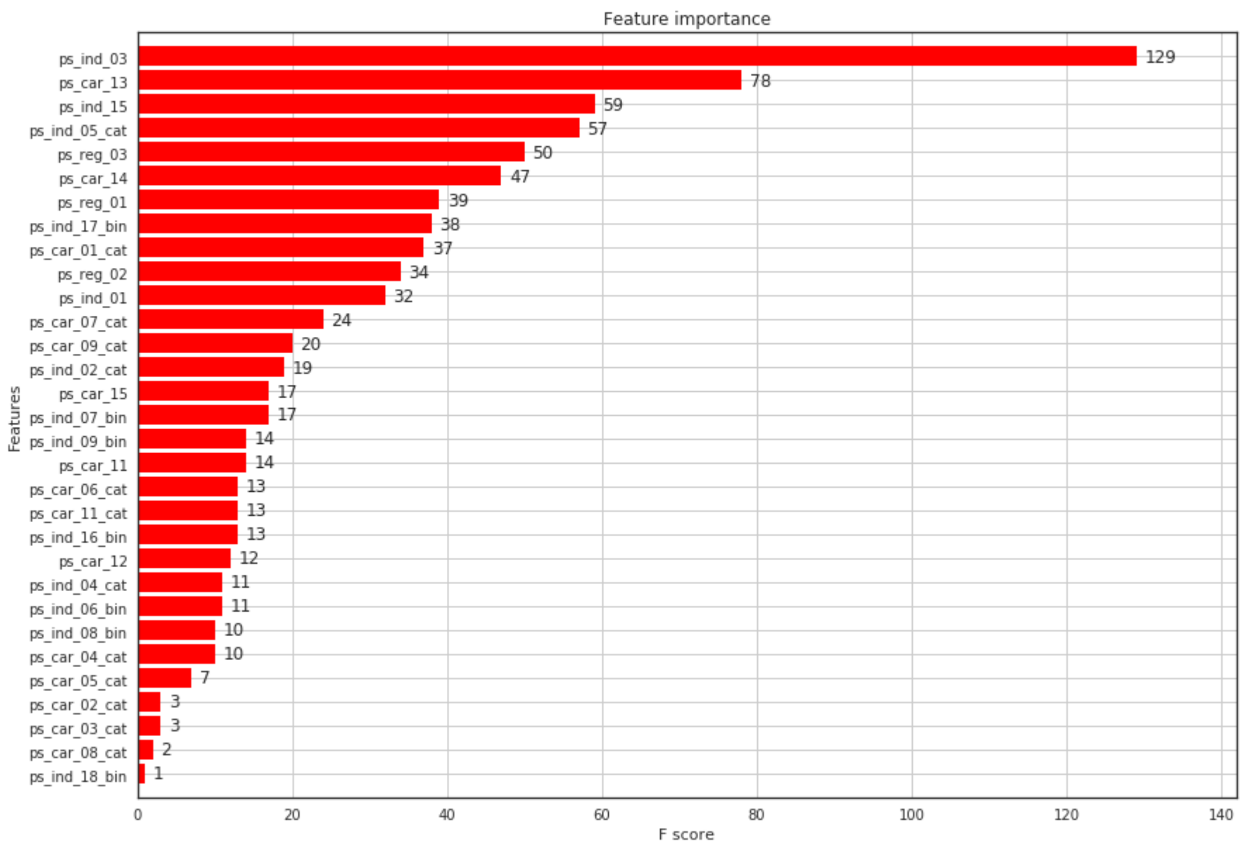
We have also done a k-cross validation on our model:

```
kf = StratifiedKFold(n_splits=cv,random_state=45)
pred_test_full =0
cv_score = []
i=1
for train_index,test_index in kf.split(X,y):
    print('{} of KFold {}'.format(i,kf.n_splits))
    xtr,xvl = X.loc[train_index],X.loc[test_index]
    ytr,yvl = y[train_index],y[test_index]

    pred_test,xg_model = runXGB(xtr,xvl,ytr,yvl,x_test,num_rounds=100,eta=
0.1)
```

```
    pred_test_full += pred_test
    cv_score.append(xg_model.best_score)
    i+=1
```

We got mean cv score as 0.6367985.

Feature importance graph in xgboost algorithm:



5.2 Logistic Regression:

Logistic regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes).

In logistic regression, the dependent variable is binary or dichotomous, i.e. it only contains data coded as 1 (TRUE, success, pregnant, etc.) or 0 (FALSE, failure, non-pregnant, etc.).

By applying grid search cv we get value of c in logistic regression as 0.1

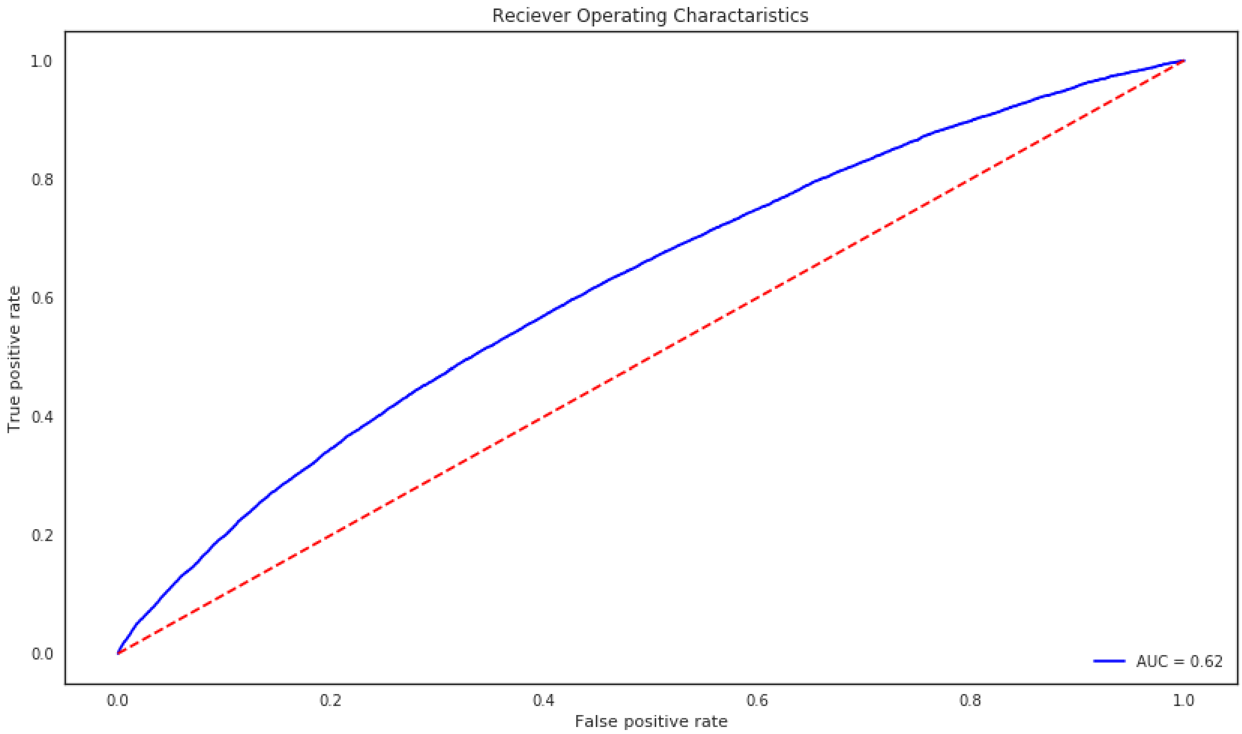
```
logreg = LogisticRegression(class_weight='balanced')
param = {'C':[0.001,0.003,0.005,0.01,0.03,0.05,0.1,0.3,0.5,1]}
clf = GridSearchCV(logreg,param,scoring='roc_auc',refit=True,cv=3)
clf.fit(X,y)
print('Best roc_auc: {:.4}, with best C: {}'.format(clf.best_score_, clf.best
_params_['C']))
Best roc_auc: 0.6207, with best C: 0.1
```

We used cross validation to fit our logistic regression model got mean cv score for the model as 0.620810304967.

Confusion matrix of model:

```
[[179931 106828]
 [  4974   5873]]
```

ROC curve of model:



6. Conclusion:

In the 2 models we prepared, our 1^{st} model is quiet good as compared to our 2^{nd} model so XGBoost classifier is very good in predicting probabilities of claim as compared to logistic regression model. In XGBoost we get roc score more than 0.636 while in logistic regression it was near to 0.62. We used roc to compare our models perfection.

7. Recommendations:

We can kept the value of cv as 2 in cross validation score, if we increase the value of this to 5 we will definitely get better model but it will increase the time to train the classifier.