SANTANDER CUSTOMER TRANSACTION

MUKHESH NARRA

Contents

1.INTRODUCTION

1.1 Problem statement

Santander is an online bank in Spain which help people and businesses prosper. They are always looking for ways to help our customers understand their financial health and identify which products and services might help them achieve their monetary goals.

The Objective is to identify which customers will make a specific transaction in the future, irrespective of the amount of money transacted.

The data is anonymized, each row containing 200 numerical values identified just with a number and target variable with two classes suggesting it is a binary classification problem.

2. Methodology

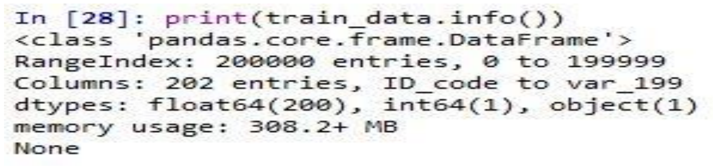
2.1 Pre-Processing

Any predictive modeling requires that we look at the data before we start modeling. However, in data mining terms looking at data refers to so much more than just looking. Looking at data refers to exploring the data, cleaning the data as well as visualizing the data through graphs and plots. This is often called as Exploratory Data Analysis.

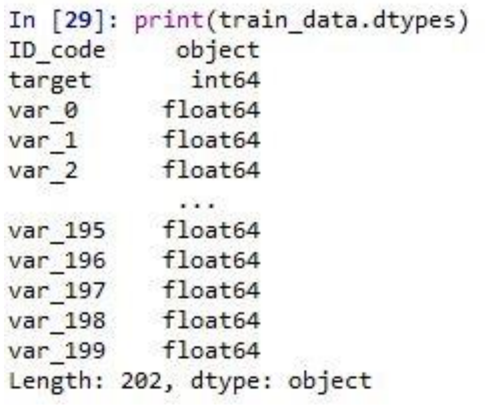
2.1.1 Exploratory Data Analysis

2.1.1.1 Data

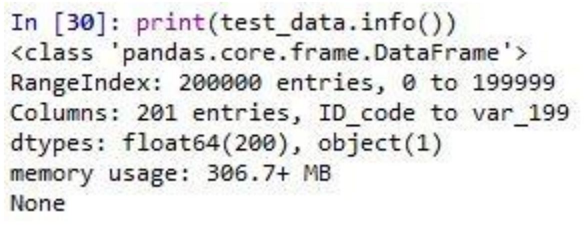
For both train and test dataset contains 200k entries with 202 and 201 columns respectively



```
In [28]: print(train_data.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Columns: 202 entries, ID_code to var_199
dtypes: float64(200), int64(1), object(1)
memory usage: 308.2+ MB
None
```



```
In [29]: print(train_data.dtypes)
ID_code     object
target       int64
var_0      float64
var_1      float64
var_2      float64
            ...
var_195    float64
var_196    float64
var_197    float64
var_198    float64
var_199    float64
Length: 202, dtype: object
```



```
In [30]: print(test_data.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Columns: 201 entries, ID_code to var_199
dtypes: float64(200), object(1)
memory usage: 306.7+ MB
None
```

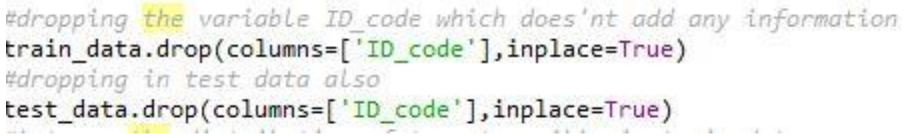
Train contains:

- ID_code (string)
- target
- 200 numerical variables, named from var_0 to var_199;

Test contains

- ID_code (string);
- 200 numerical variables, named from var_0 to var_199;

Where dropping the variable ID_code which doesn't add any information.

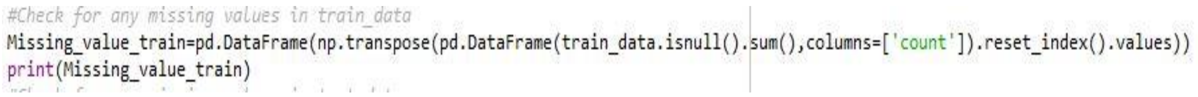


```
#dropping the variable ID_code which does'nt add any information
train_data.drop(columns=['ID_code'],inplace=True)
#dropping in test data also
test_data.drop(columns=['ID_code'],inplace=True)
```

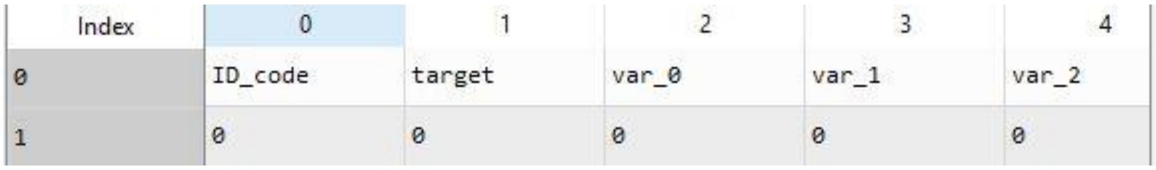
2.1.1.2 Missing Value Analysis

Missing value analysis is done to check is there any missing value present in given dataset. Missing values can be easily treated using various methods like mean, median method, knn imputation method to impute missing value.

Let us check missing values in train data

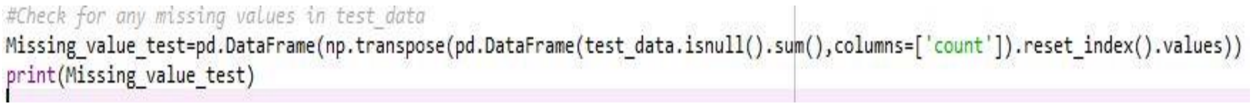


```
#Check for any missing values in train_data
Missing_value_train=pd.DataFrame(np.transpose(pd.DataFrame(train_data.isnull().sum(),columns=['count']).reset_index().values))
print(Missing_value_train)
```

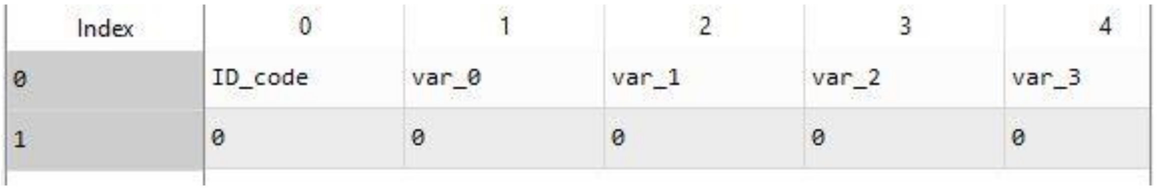


Index	0	1	2	3	4
0	ID_code	target	var_0	var_1	var_2
1	0	0	0	0	0

Let us check missing values in test data



```
#Check for any missing values in test_data
Missing_value_test=pd.DataFrame(np.transpose(pd.DataFrame(test_data.isnull().sum(),columns=['count']).reset_index().values))
print(Missing_value_test)
```

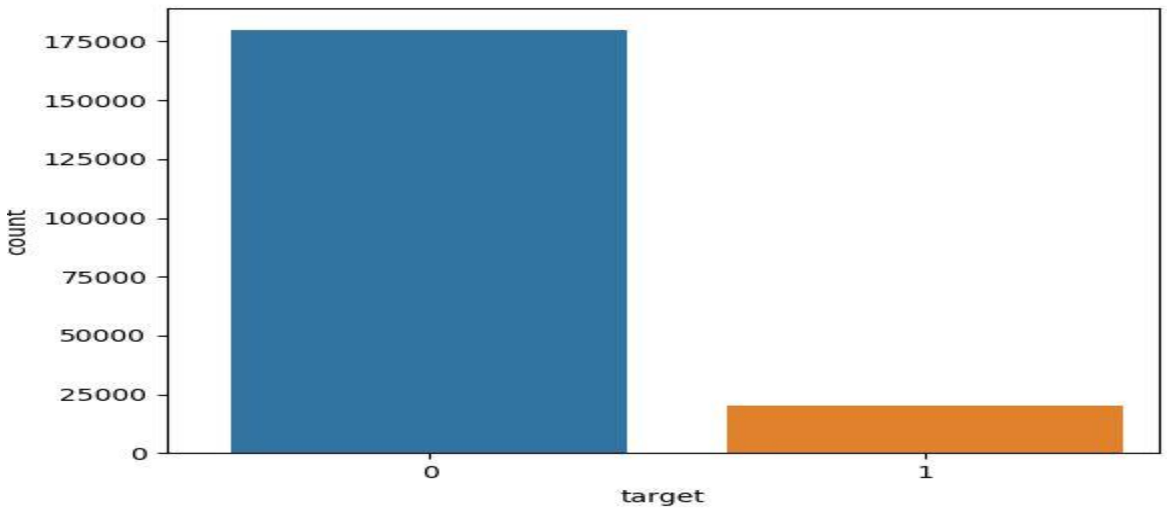


Index	0	1	2	3	4
0	ID_code	var_0	var_1	var_2	var_3
1	0	0	0	0	0

In both train and test datasets we didn't found any missing values.

2.1.1.3 Target Class Distribution

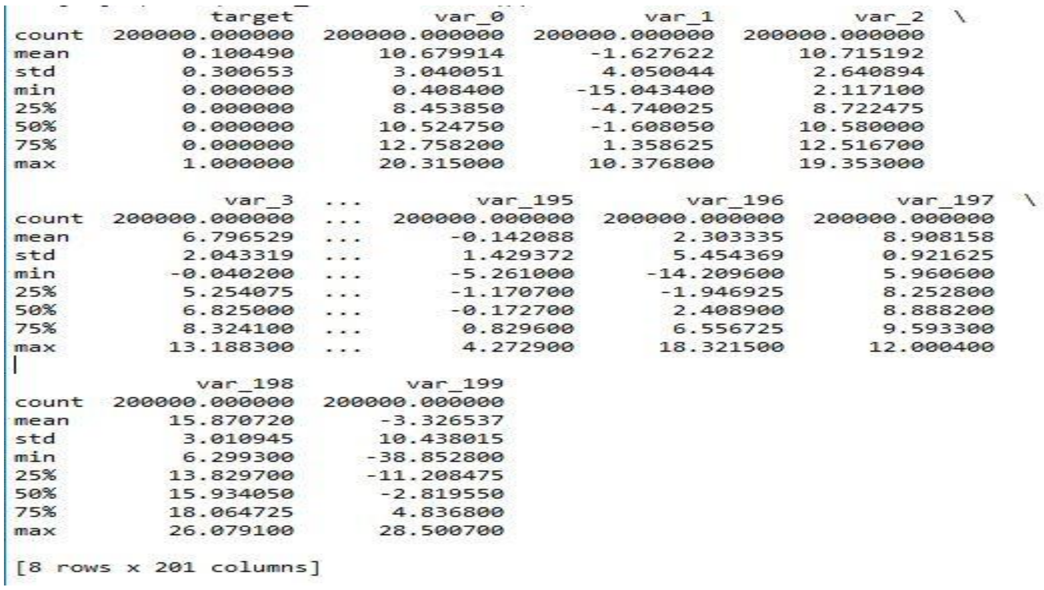
Let us check the target class distribution



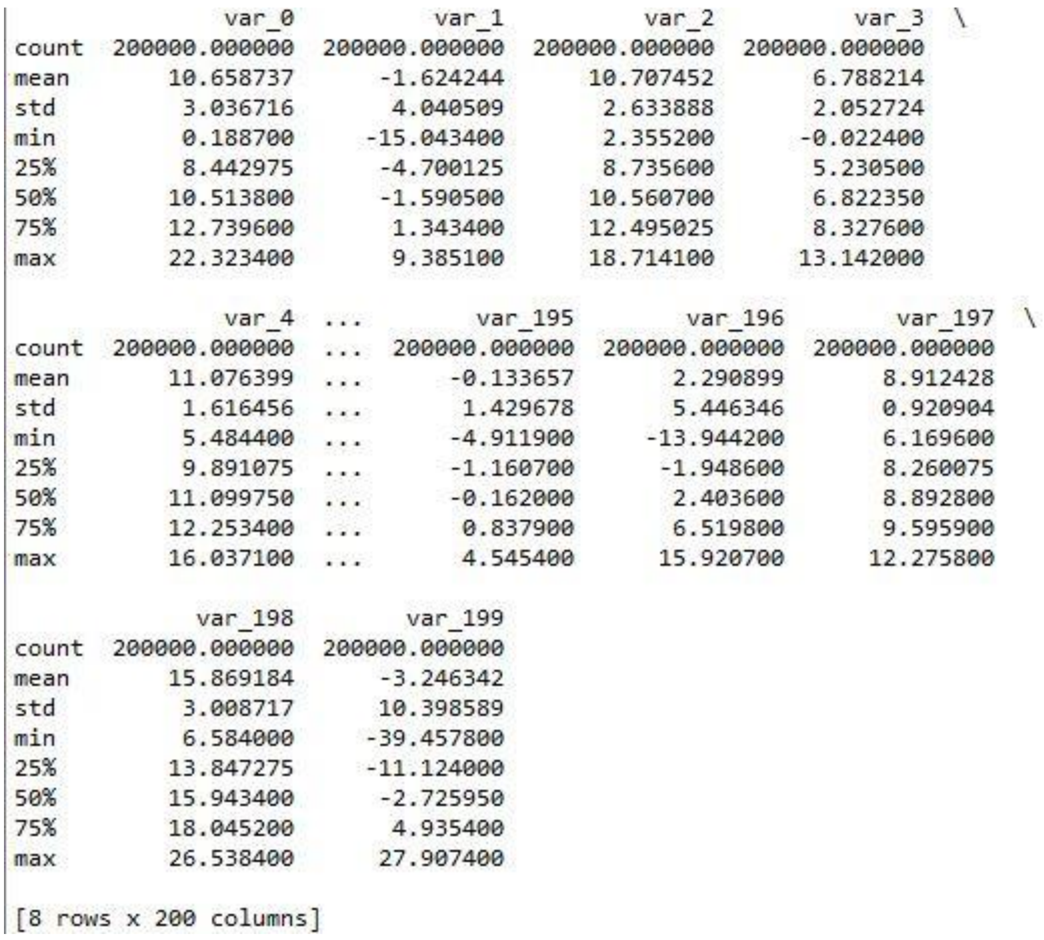
As observation made from above figure we can say that train data set is imbalanced because class 0 entries are more than class 1 entries.

2.1.1.4 Numerical value Analysis

Let us check numerical analysis on test and train data.



```
              target          var_0          var_1          var_2  \
count  200000.000000  200000.000000  200000.000000  200000.000000
mean        0.100490      10.679914      -1.627622      10.715192
std         0.300653       3.040051       4.050044       2.640894
min         0.000000       0.408400     -15.043400       2.117100
25%         0.000000       8.453850      -4.740025       8.722475
50%         0.000000      10.524750      -1.608050      10.580000
75%         0.000000      12.758200       1.358625      12.516700
max         1.000000      20.315000      10.376800      19.353000

               var_3  ...        var_195        var_196        var_197  \
count  200000.000000  ...  200000.000000  200000.000000  200000.000000
mean        6.796529  ...      -0.142088       2.303335       8.908158
std         2.043319  ...       1.429372       5.454369       0.921625
min        -0.040200  ...      -5.261000     -14.209600       5.960600
25%         5.254075  ...      -1.170700      -1.946925       8.252800
50%         6.825000  ...      -0.172700       2.408900       8.888200
75%         8.324100  ...       0.829600       6.556725       9.593300
max        13.188300  ...       4.272900      18.321500      12.000400

             var_198        var_199
count  200000.000000  200000.000000
mean       15.870720      -3.326537
std         3.010945      10.438015
min         6.299300     -38.852800
25%        13.829700     -11.208475
50%        15.934050      -2.819550
75%        18.064725       4.836800
max        26.079100      28.500700

[8 rows x 201 columns]
```



```
               var_0          var_1          var_2          var_3  \
count  200000.000000  200000.000000  200000.000000  200000.000000
mean       10.658737      -1.624244      10.707452       6.788214
std         3.036716       4.040509       2.633888       2.052724
min         0.188700     -15.043400       2.355200      -0.022400
25%         8.442975      -4.700125       8.735600       5.230500
50%        10.513800      -1.590500      10.560700       6.822350
75%        12.739600       1.343400      12.495025       8.327600
max        22.323400       9.385100      18.714100      13.142000

               var_4  ...        var_195        var_196        var_197  \
count  200000.000000  ...  200000.000000  200000.000000  200000.000000
mean       11.076399  ...      -0.133657       2.290899       8.912428
std         1.616456  ...       1.429678       5.446346       0.920904
min         5.484400  ...      -4.911900     -13.944200       6.169600
25%         9.891075  ...      -1.160700      -1.948600       8.260075
50%        11.099750  ...      -0.162000       2.403600       8.892800
75%        12.253400  ...       0.837900       6.519800       9.595900
max        16.037100  ...       4.545400      15.920700      12.275800

             var_198        var_199
count  200000.000000  200000.000000
mean       15.869184      -3.246342
std         3.008717      10.398589
min         6.584000     -39.457800
25%        13.847275     -11.124000
50%        15.943400      -2.725950
75%        18.045200       4.935400
max        26.538400      27.907400

[8 rows x 200 columns]
```

Some few observations are made:

- standard deviation is relatively large for both train and test variable data.
- min, max, mean, sdt values for train and test data looks quite closely.
- mean values are distributed over a large range.

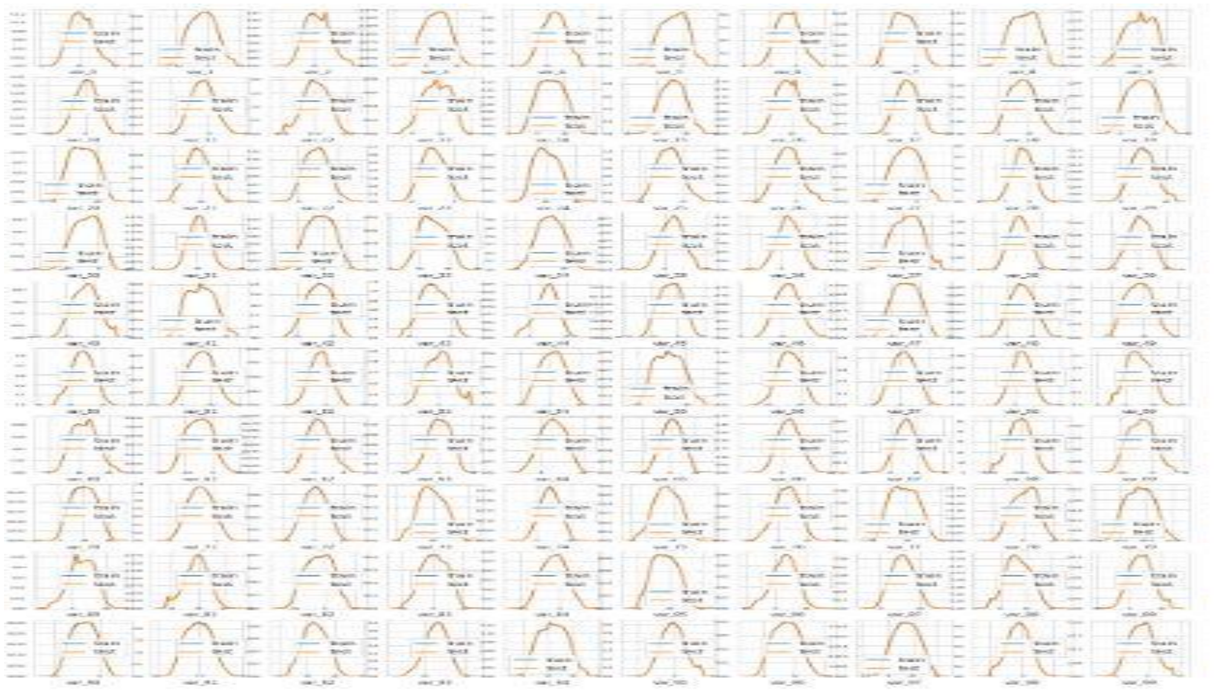
2.1.1.4.1 Density Plot of features

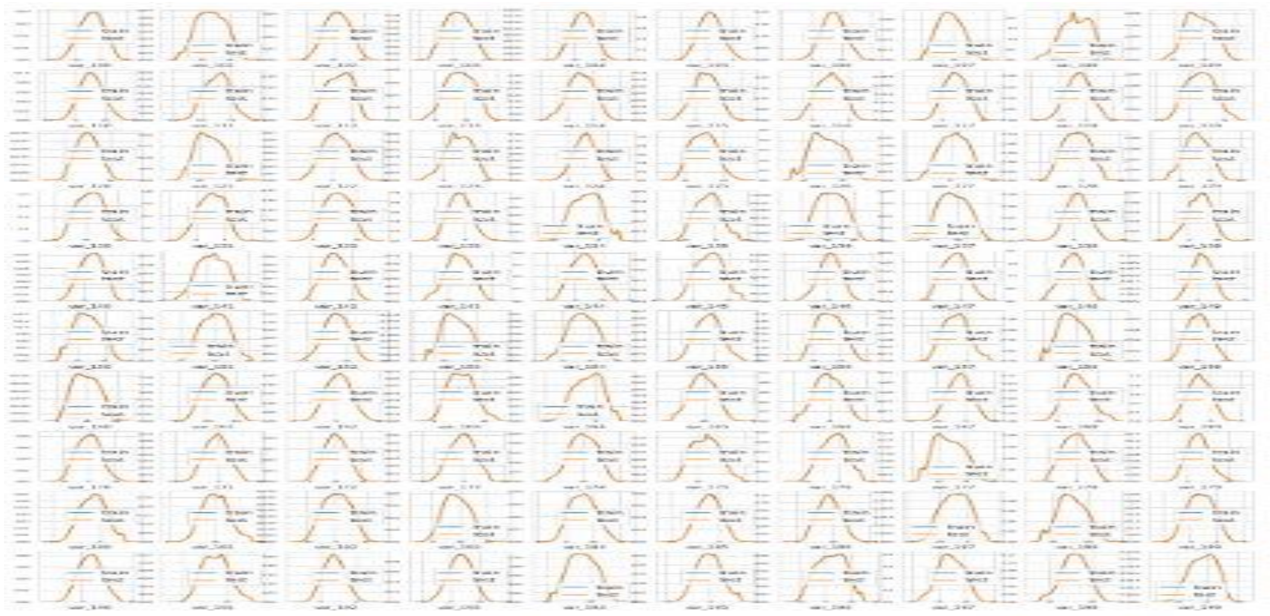
Let’s represent the density plot of variables in train dataset with the distribution for values with target value 0 and 1.

We can observe that there is a considerable number of features with significant different distribution for the two target values.
For example var_0, var_1, var_2, var_5, var_9, var_13, var_106, var_109, var_139 and many others.

We will take this into consideration in the future for the selection of the features for our prediction model.

Let look the same distribution of features for both train and test datasets parallelly.

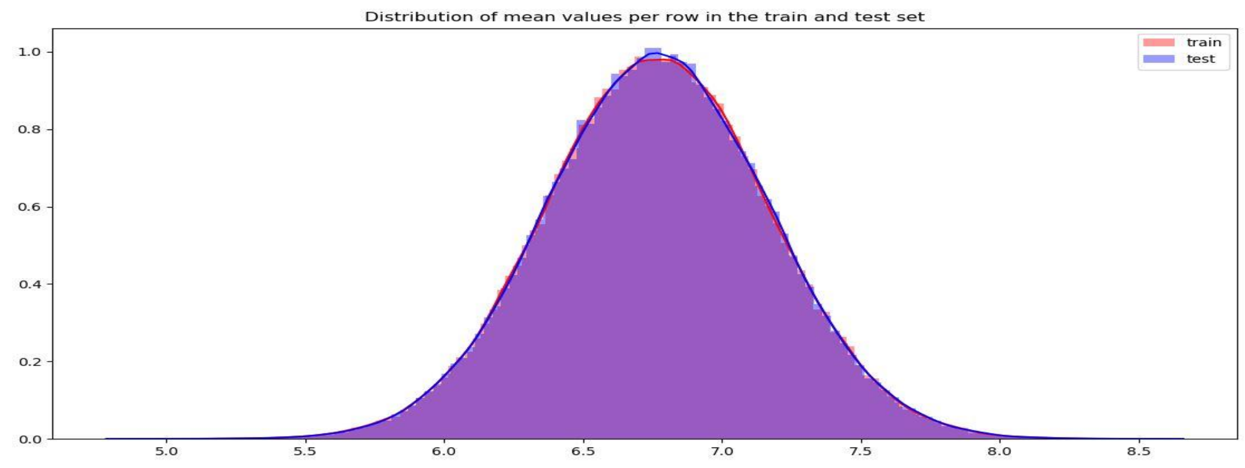




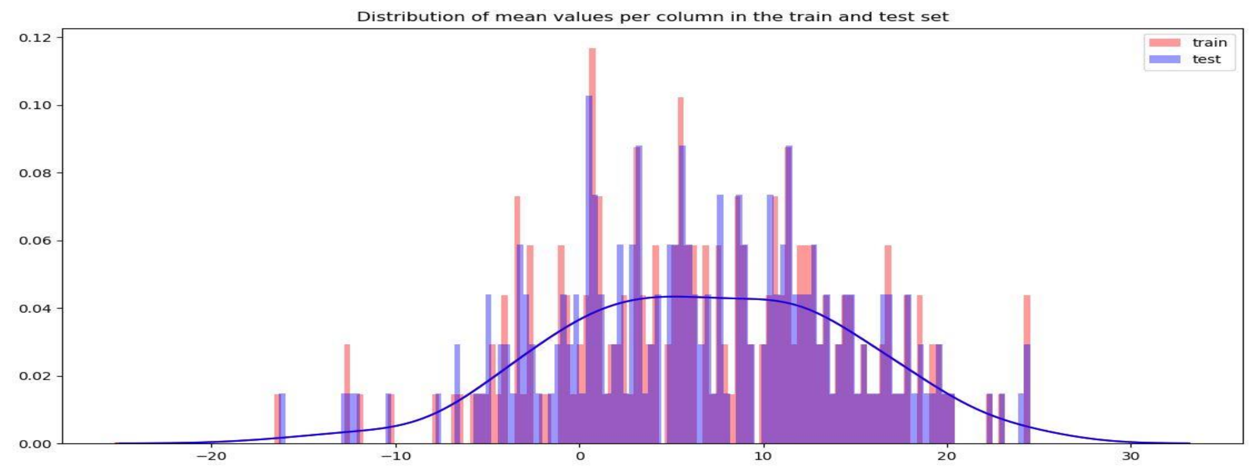
We can observe that both train and test are well balanced with respect to numerical features.

2.1.1.4.2 Distribution of mean and standard deviation

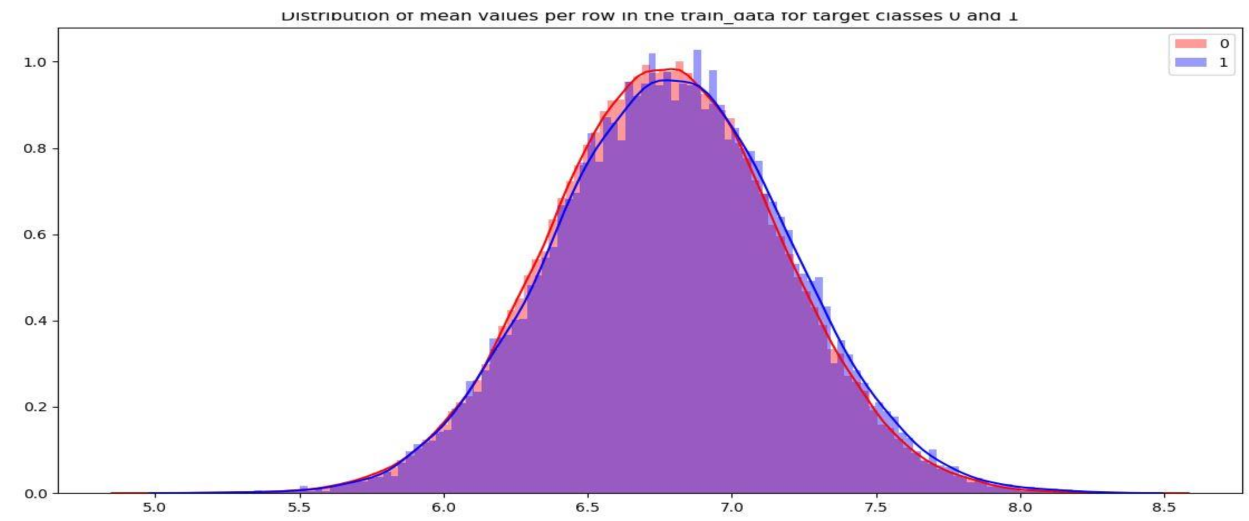
Let's check the distribution of the mean values per row in the train and test set.



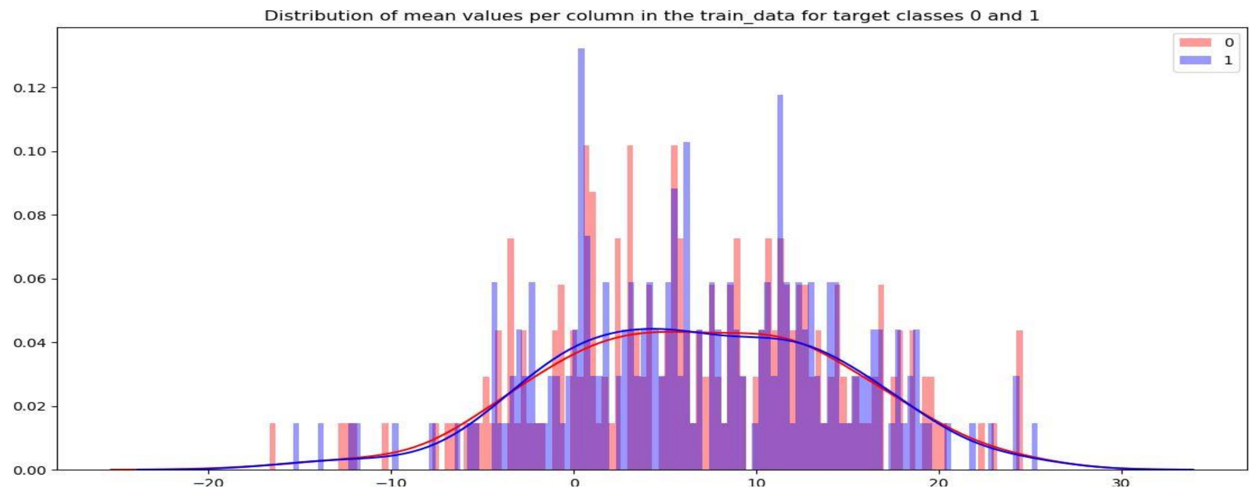
Let's check the distribution of the mean values per column in the train and test set.



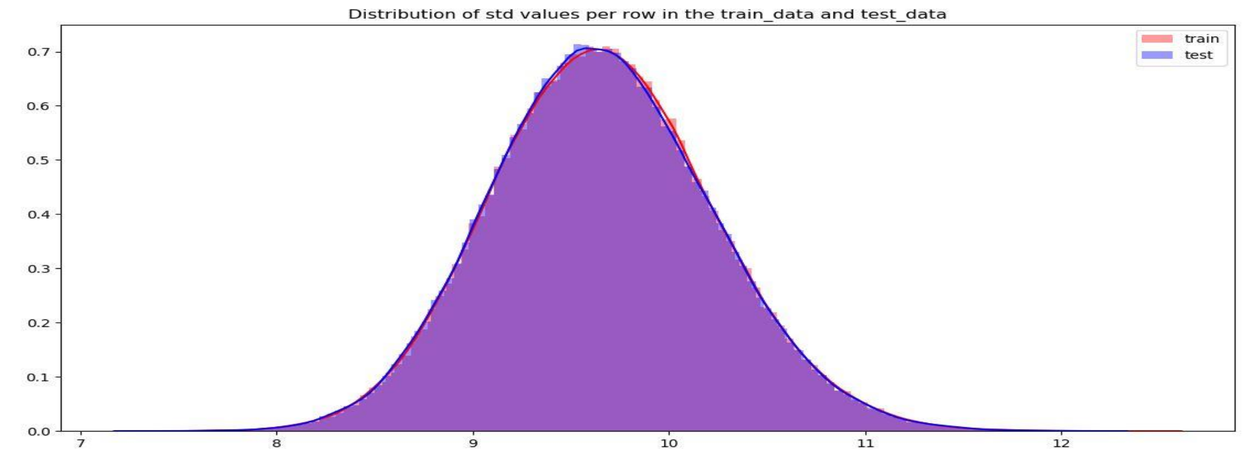
Let's check the distribution of the mean values per row in the train data set by target classes.



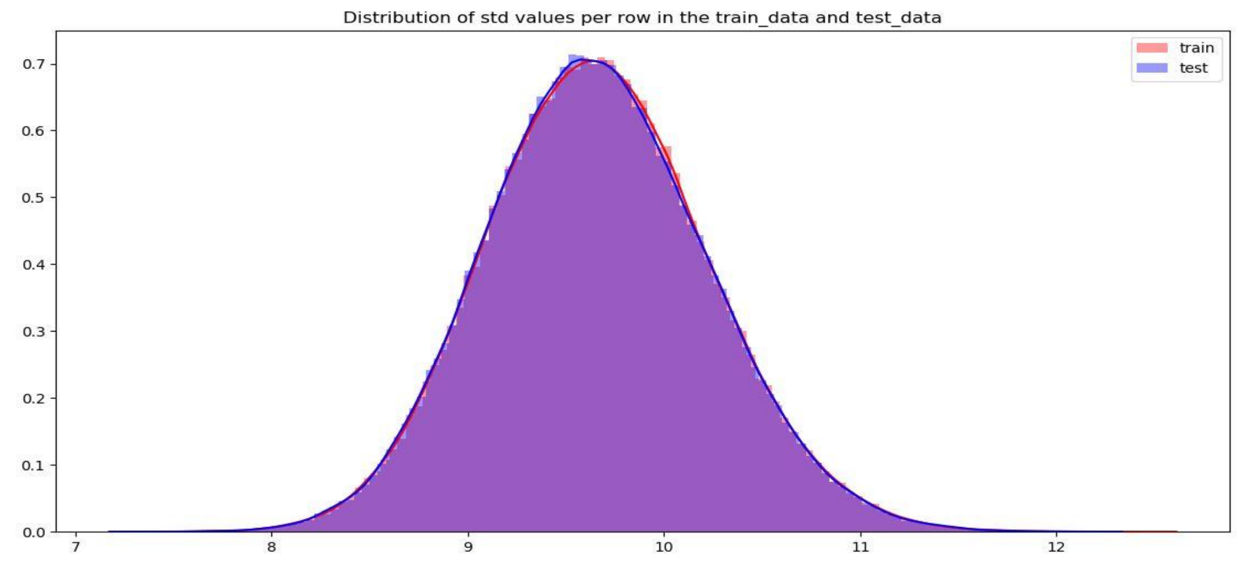
Let's check the distribution of the mean values per column in the train data set by target classes.



Let's check the distribution of the standard deviation values per row in the train and test set.

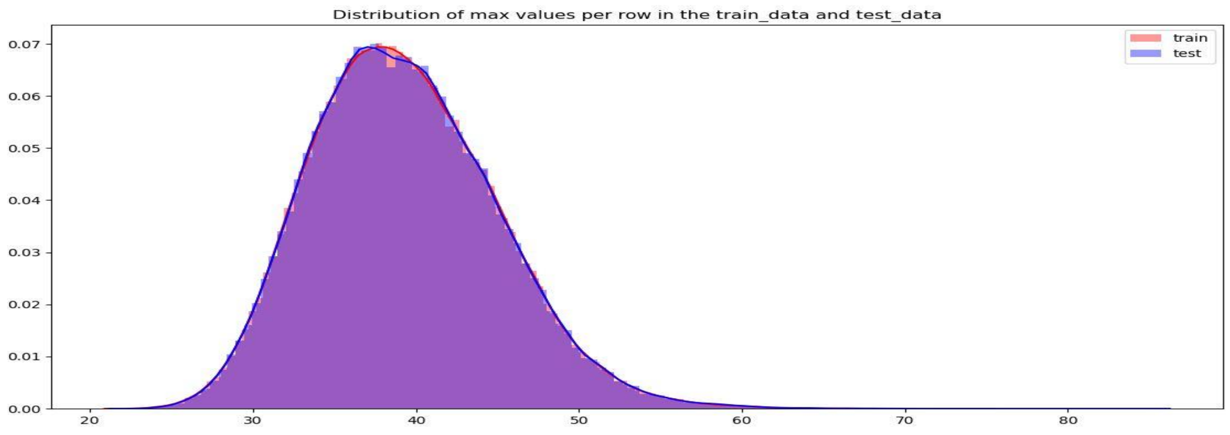


Let's check the distribution of the standard deviation values per column in the train and test set.

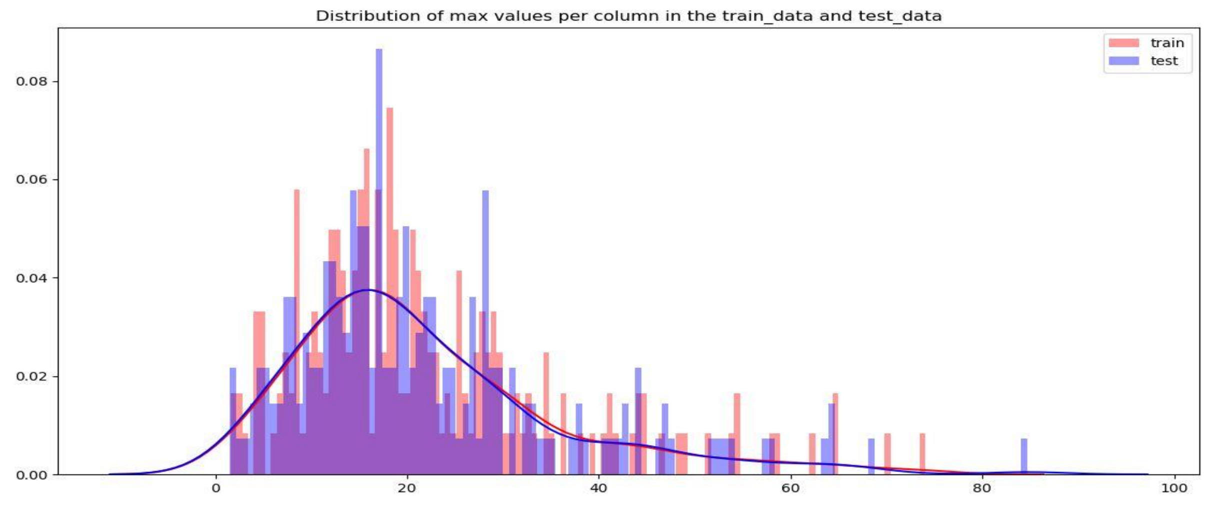


2.1.1.4.2 Distribution of max and min

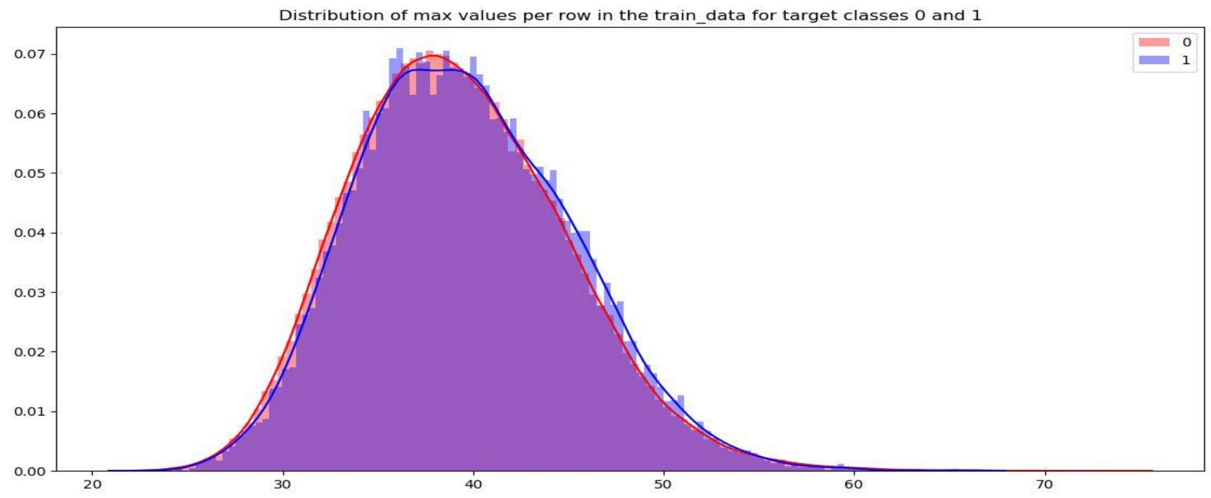
Let's check the distribution of the max values per row in the train and test set.



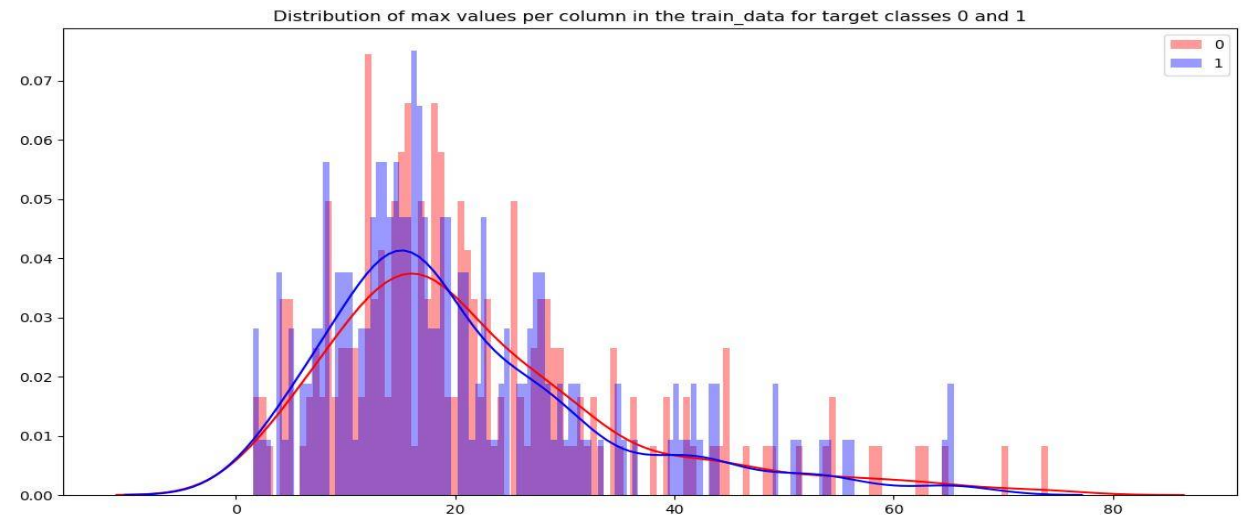
Let's check the distribution of the max values per column in the train and test set.



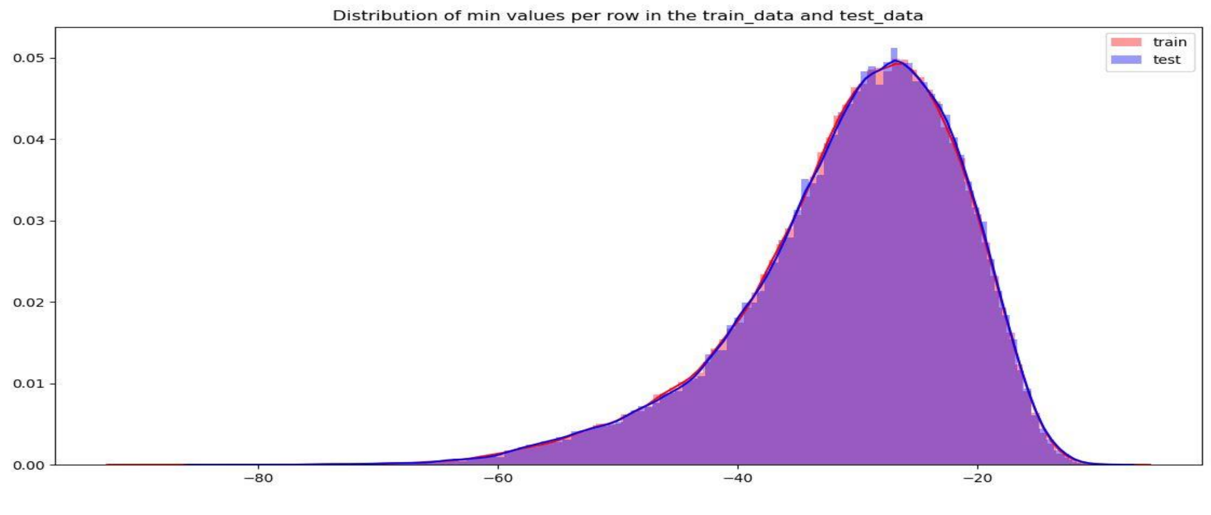
Let's check the distribution of the max values per row in the train data set by target classes.



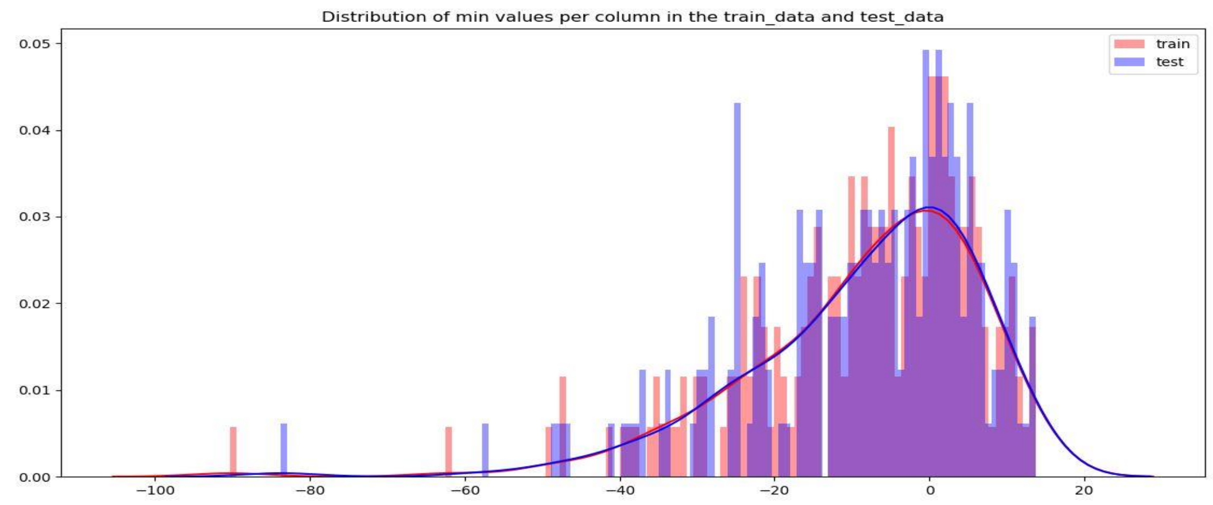
Let's check the distribution of the max values per column in the train data set by target classes.



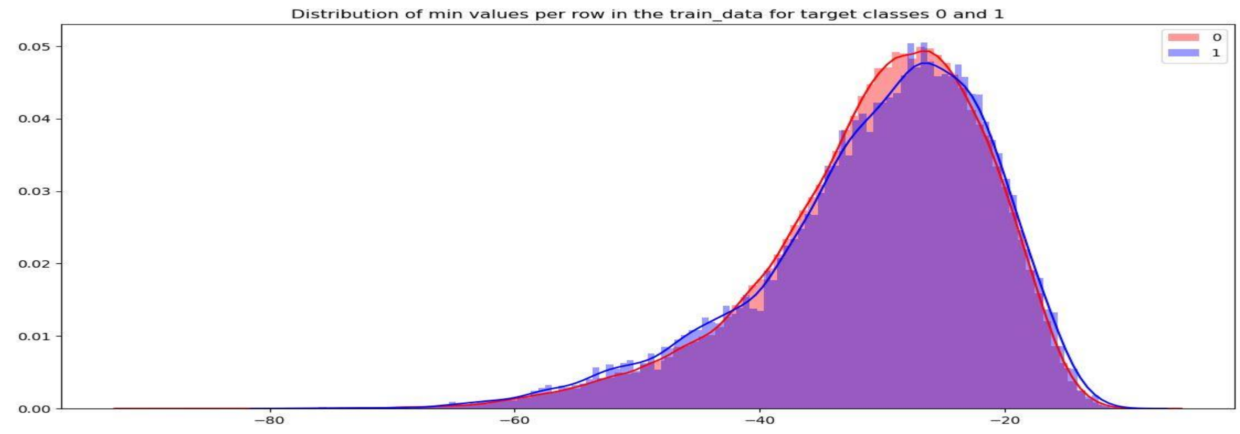
Let's check the distribution of the min values per row in the train and test set.



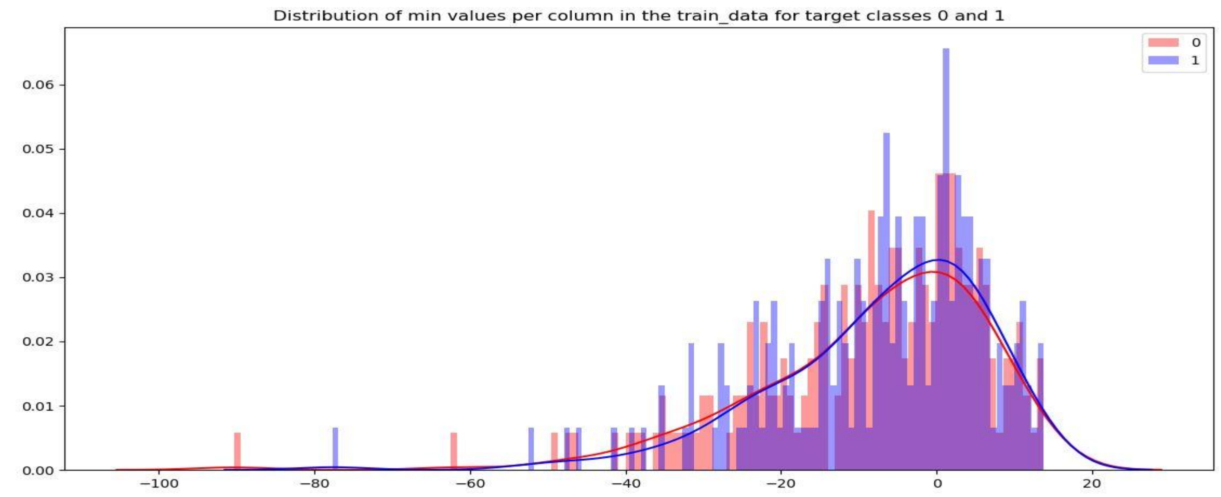
Let's check the distribution of the min values per column in the train and test set.



Let's check the distribution of the min values per row in the train data set by target classes.



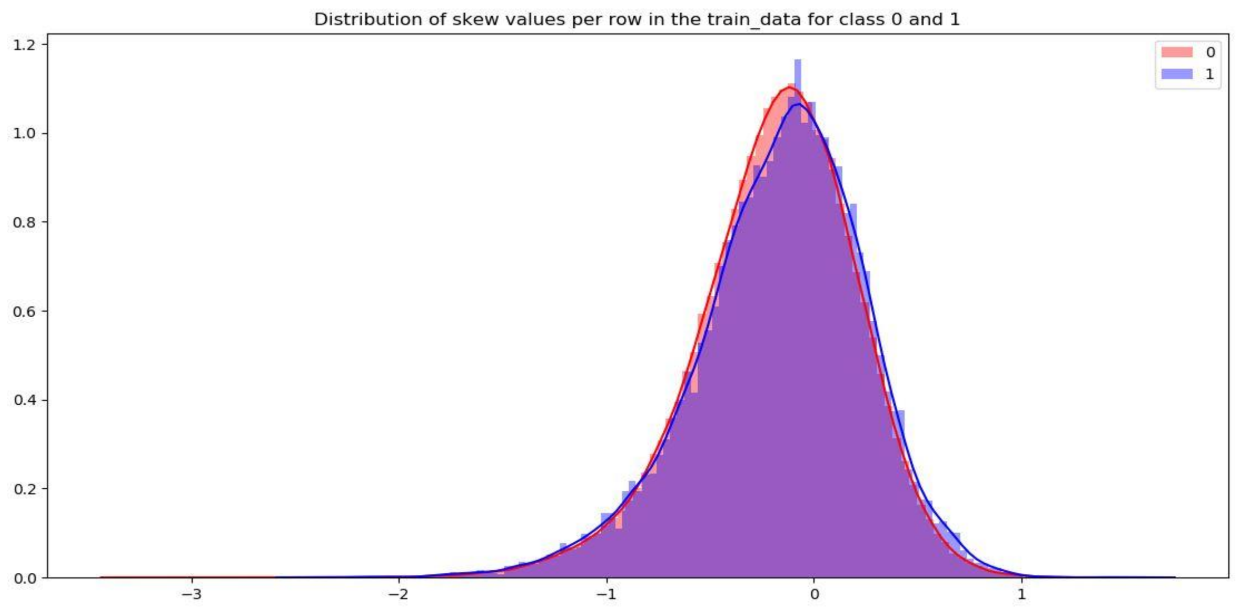
Let's check the distribution of the min values per column in the train data set by target classes.



2.1.1.4.3 Distribution of Skewness

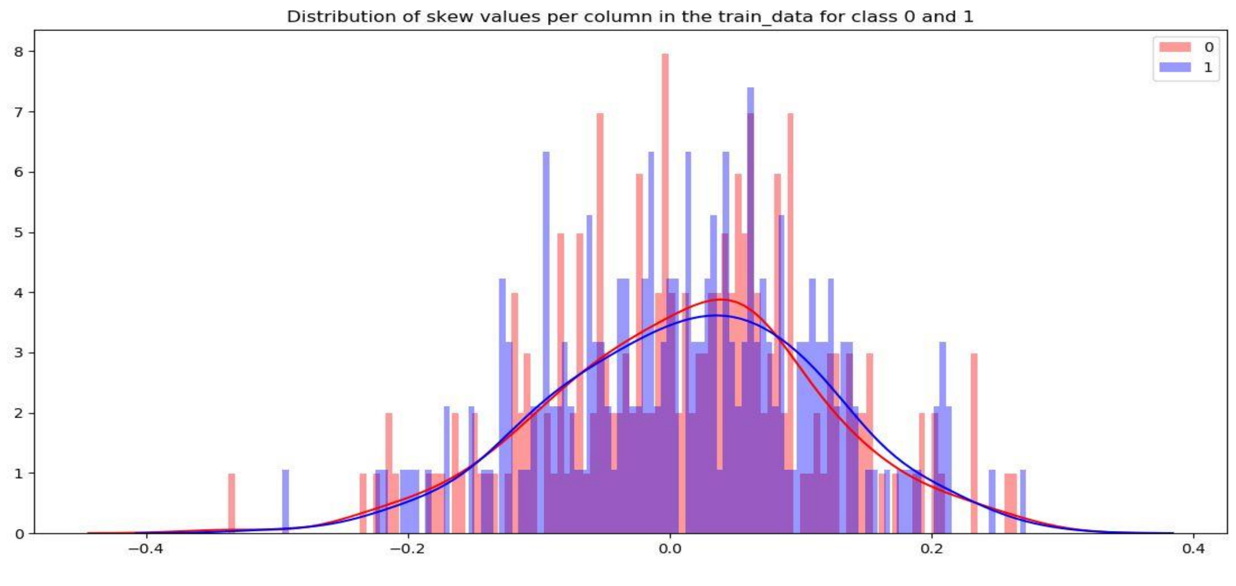
Skewness analysis tells about the symmetry of distribution of data.

Let's check the distribution of the skewness values per row in the train data set by target classes.



We found that the distribution is left-skewed.

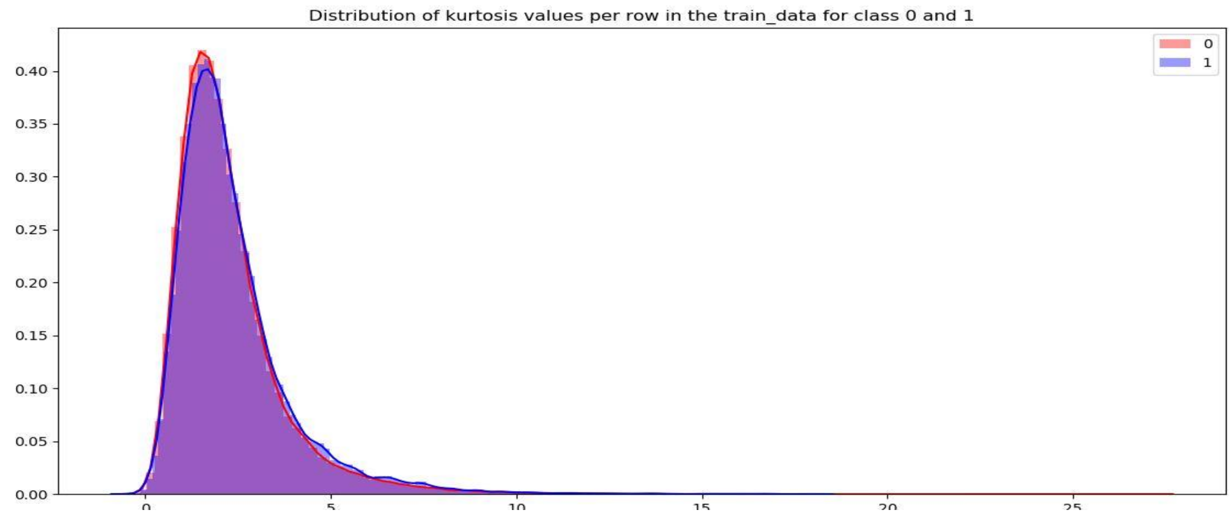
Let's check the distribution of the skewness values per column in the train data set by target classes.



2.1.1.4.4 Distribution of Kurtosis

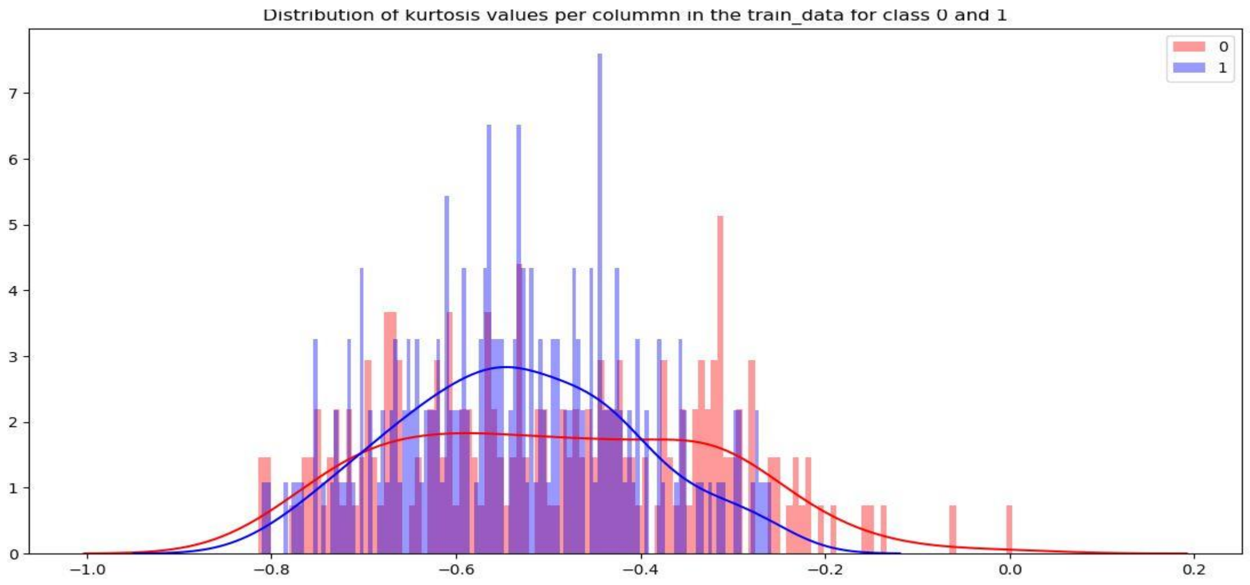
Kurtosis is crucial in financial domain where it identifies whether the tails of a given distribution contain extreme values.

Let's see now the distribution of kurtosis on rows in train separated for values of target classes.



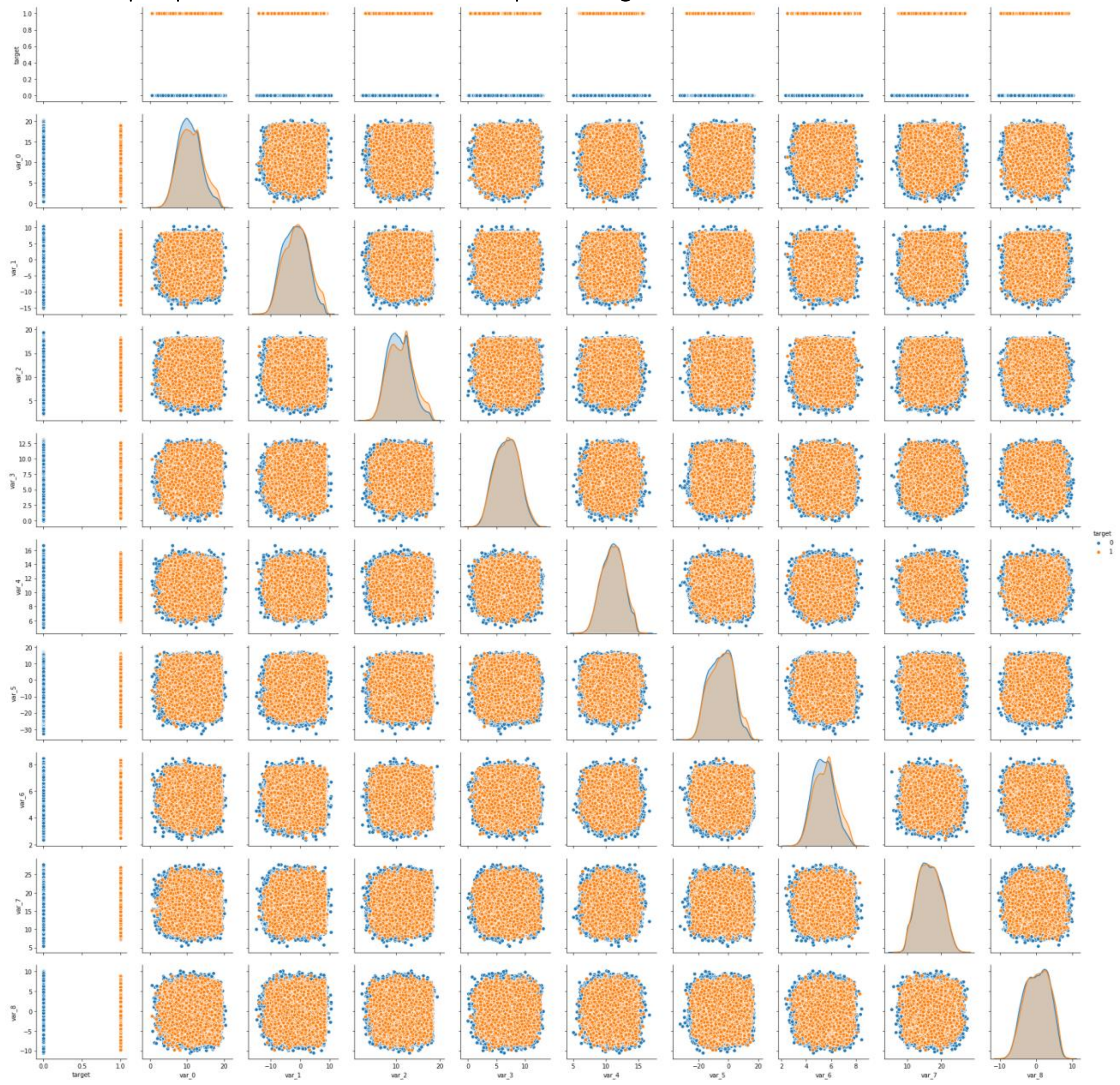
We found the distribution to be Leptokurtic where it means there will be extreme values on tails which indicate the outliers. Where in this data it indicate they are outliers but we are not sure either to analyze the outliers because they haven't mentioned about feature names.

Let's see now the distribution of kurtosis on columns in train separated for values of target classes.



2.1.1.4.5 Pair Plot

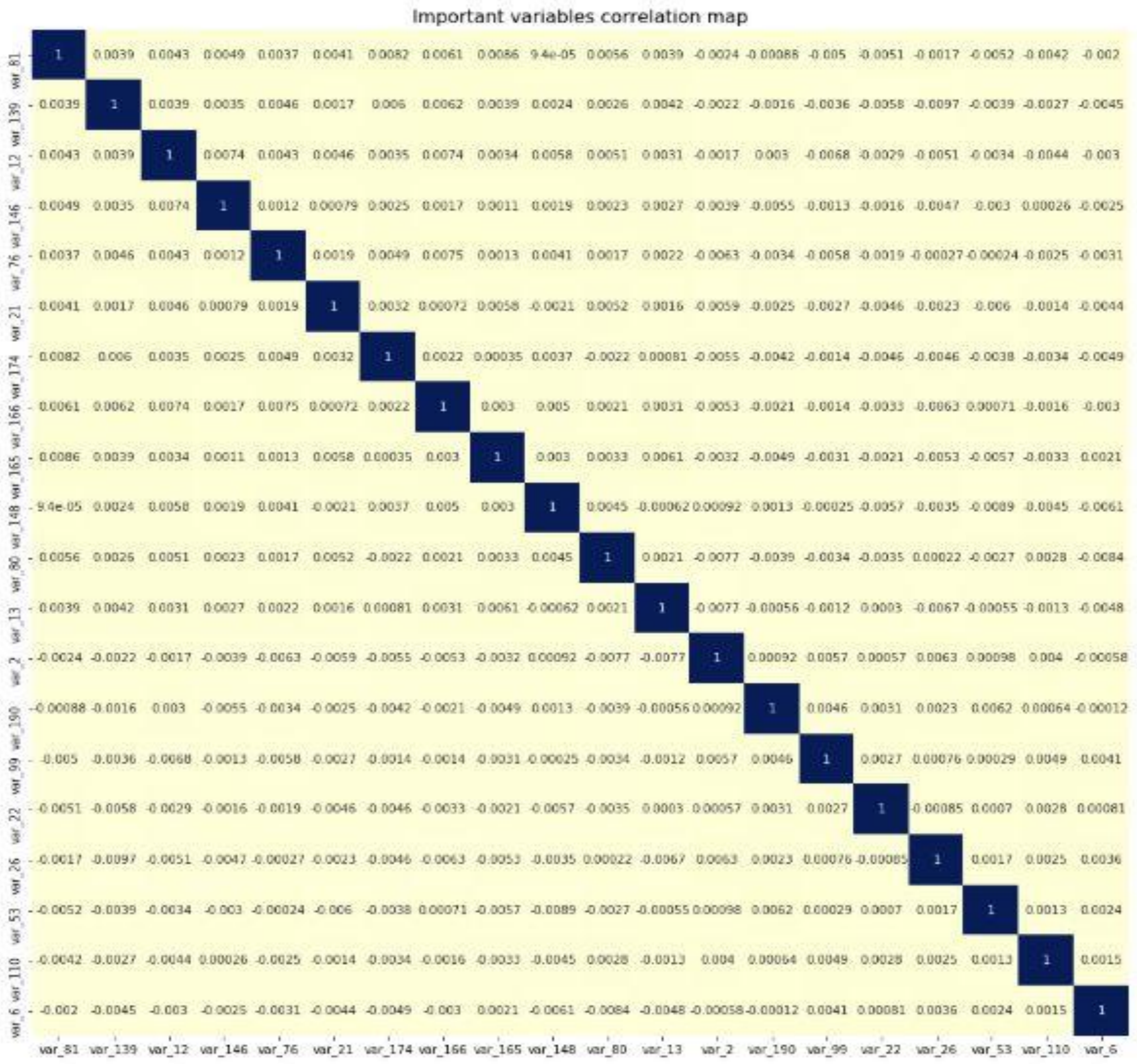
Let's check pair plot of numerical variable for separate target classes.



For my intuition data is too much overlapped where tree based algoritms, Knn might work well. And San tander team might applied the SVD or PCA.

2.1.1.5 Feature Correlation

We calculate now the correlations between the features in train set.

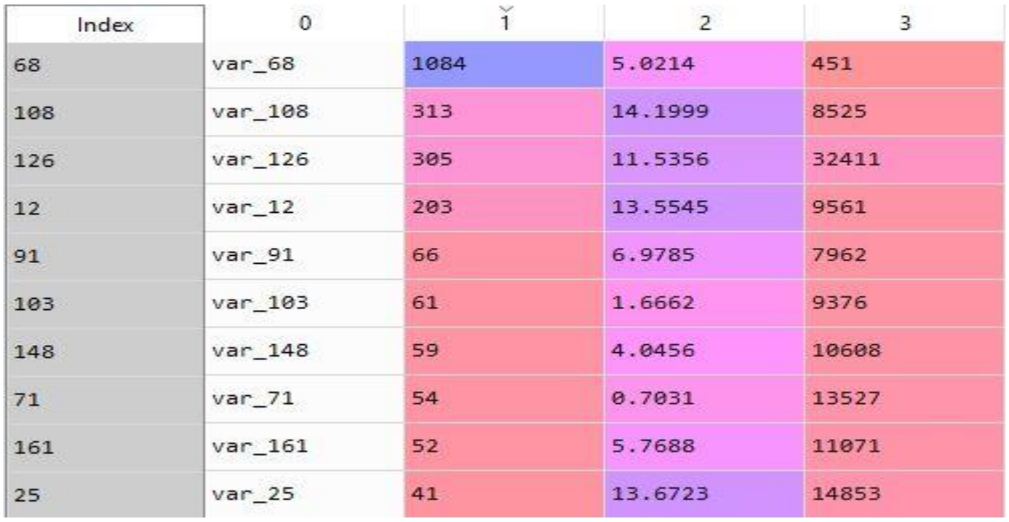


As we observe that the correlation between the features is very small. Overall, the correlation of the fea tures with respect to target are very low. The above plots helped us in identifying the important individ ual variables which are correlated with target. But we can't derive any importance of feature from her e. We can discuss in feature importance section.

2.1.1.6 Duplicate Values

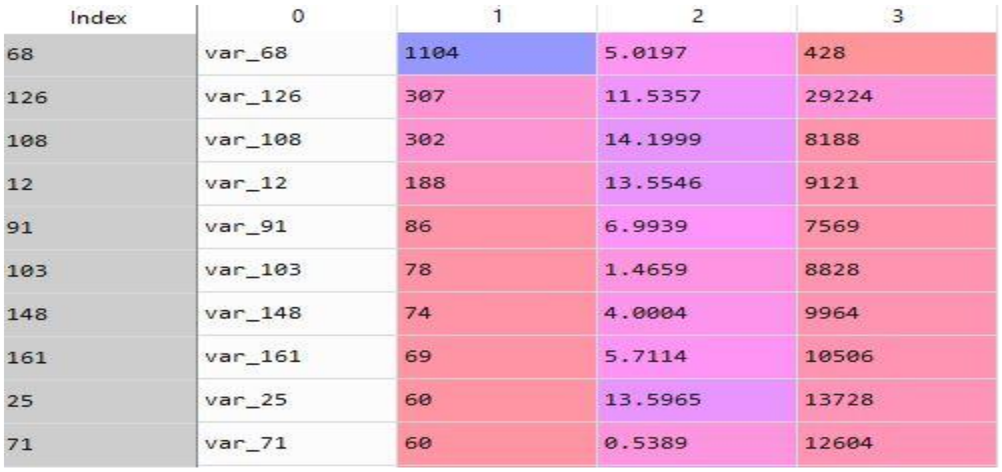
Let's now check how many duplicate values exists per columns.

For train_data:



Index	0	1	2	3
68	var_68	1084	5.0214	451
108	var_108	313	14.1999	8525
126	var_126	305	11.5356	32411
12	var_12	203	13.5545	9561
91	var_91	66	6.9785	7962
103	var_103	61	1.6662	9376
148	var_148	59	4.0456	10608
71	var_71	54	0.7031	13527
161	var_161	52	5.7688	11071
25	var_25	41	13.6723	14853

For test_data:



Index	0	1	2	3
68	var_68	1104	5.0197	428
126	var_126	307	11.5357	29224
108	var_108	302	14.1999	8188
12	var_12	188	13.5546	9121
91	var_91	86	6.9939	7569
103	var_103	78	1.4659	8828
148	var_148	74	4.0004	9964
161	var_161	69	5.7114	10506
25	var_25	60	13.5965	13728
71	var_71	60	0.5389	12604

Out of which top 5 variables with high count are taken for the feature engineering in future sections.

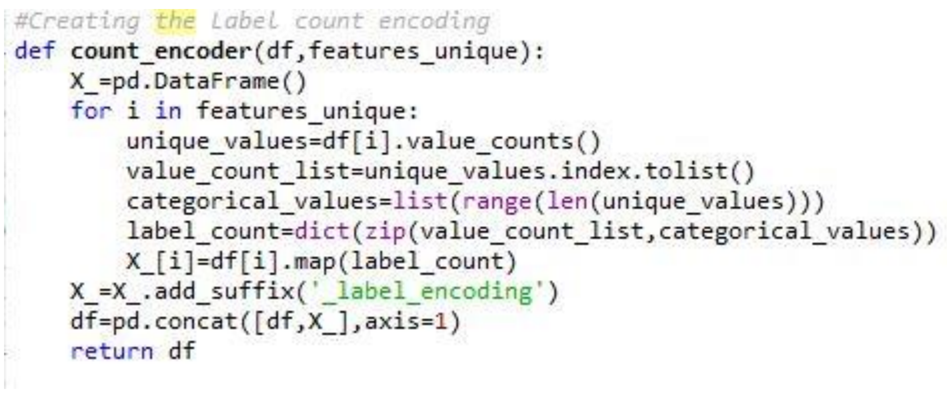
2.1.2 Feature Engineering

2.1.2.1 Feature Creation

In addition to existing independent variables, we will create new variables to improve the prediction po wer of model. Where we can check is there any increase in accuracy of model after a training with additional features.

Where these variables are created based on analysis of data. we follow label count encoding for creating variables where duplicate count is more. Where duplicate values is high for few columns so my intuition says these were may be categorical variable so one of the techniques for categorical variables is label count encoding. Where label count encoding is it revolves around *ranking categories by their counts* in the train set. This technique is not sensitive to outliers.

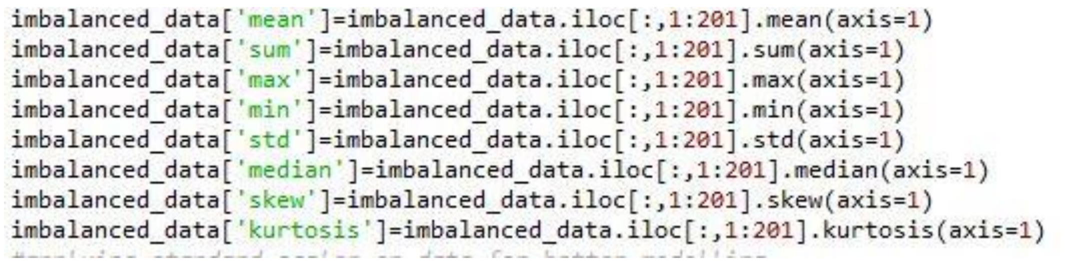
Code for label count encoding:



```
#Creating the Label count encoding
def count_encoder(df,features_unique):
    X_=pd.DataFrame()
    for i in features_unique:
        unique_values=df[i].value_counts()
        value_count_list=unique_values.index.tolist()
        categorical_values=list(range(len(unique_values)))
        label_count=dict(zip(value_count_list,categorical_values))
        X_[i]=df[i].map(label_count)
    X_=X_.add_suffix('_label_encoding')
    df=pd.concat([df,X_],axis=1)
    return df
```

Where comes to another variables we have created variable of mean,min,max,median,skew,kurtosis per row because per statistics of train and test dataset looks similar.

Code for creation for other variables:



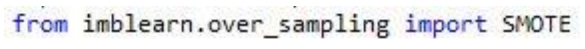
```
imbalanced_data['mean']=imbalanced_data.iloc[:,1:201].mean(axis=1)
imbalanced_data['sum']=imbalanced_data.iloc[:,1:201].sum(axis=1)
imbalanced_data['max']=imbalanced_data.iloc[:,1:201].max(axis=1)
imbalanced_data['min']=imbalanced_data.iloc[:,1:201].min(axis=1)
imbalanced_data['std']=imbalanced_data.iloc[:,1:201].std(axis=1)
imbalanced_data['median']=imbalanced_data.iloc[:,1:201].median(axis=1)
imbalanced_data['skew']=imbalanced_data.iloc[:,1:201].skew(axis=1)
imbalanced_data['kurtosis']=imbalanced_data.iloc[:,1:201].kurtosis(axis=1)
```

2.1.2.2 SMOTE(Synthetic Minority Oversampling Technique)

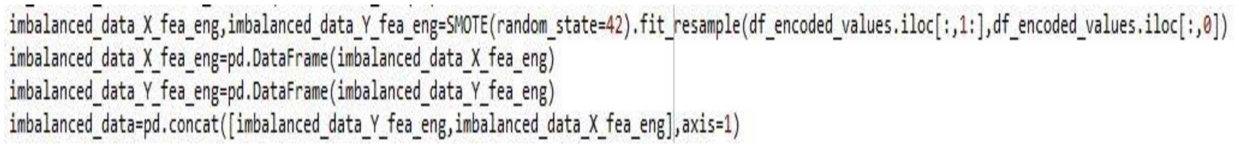
Where we have problem of imbalanced data set which means that there is class imbalance distribution which can be solved by three techniques 1. Down sampling 2. up sampling 3. Class weights. Where SMOTE is a technique of oversampling which increase the count of minority instances to count of majority instances.

The new instances are not just copies of existing minority cases, the algorithm takes samples of the feature space for each target class and its nearest neighbors, and generates new examples that combine features of the target case with features of its neighbors which means that it will connect the all instances by lines and new generated instances fall on those line in random.

Code for applying SMOTE to our dataset:



```
from imblearn.over_sampling import SMOTE
```



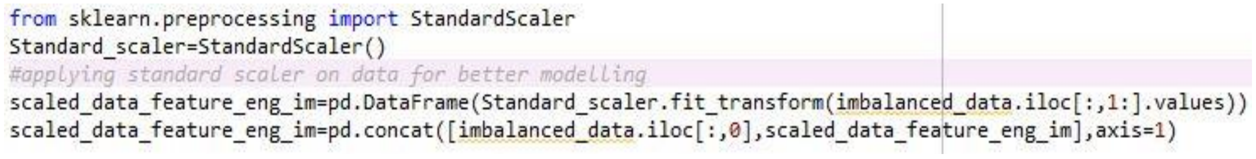
```
imbalanced_data_X_fea_eng,imbalanced_data_Y_fea_eng=SMOTE(random_state=42).fit_resample(df_encoded_values.iloc[:,1:],df_encoded_values.iloc[:,0])
imbalanced_data_X_fea_eng=pd.DataFrame(imbalanced_data_X_fea_eng)
imbalanced_data_Y_fea_eng=pd.DataFrame(imbalanced_data_Y_fea_eng)
imbalanced_data=pd.concat([imbalanced_data_Y_fea_eng,imbalanced_data_X_fea_eng],axis=1)
```

After applying SMOTE algorithm there is rise in AUC score for all algorithms but for random forest there is more rise in score.

2.1.2.3 Feature Scaling

Feature scaling includes two functions normalization and standardization. It is done reduce unwanted variation either within or between variables and to bring all of the variables into proportion with one another. Which helps for model not to show any bias on particular variable especially distance based algorithms.

Code for Feature Scaling:

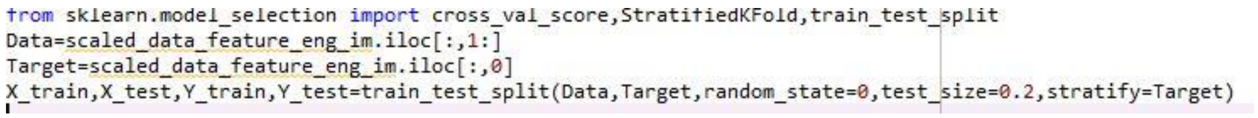


```
from sklearn.preprocessing import StandardScaler
Standard_scaler=StandardScaler()
#applying standard scaler on data for better modelling
scaled_data_feature_eng_im=pd.DataFrame(Standard_scaler.fit_transform(imbalanced_data.iloc[:,1:].values))
scaled_data_feature_eng_im=pd.concat([imbalanced_data.iloc[:,0],scaled_data_feature_eng_im],axis=1)
```

3. Modeling

3.1 Data Splitting

For Every model building we need train and test data. For Every model building we need train and test data so we will sample the data into train and test data in 80:20 ratio that means 80% of data is used as train data is used for training the model and 20% of data is used as test data to test the model for evaluative model.   So we are using Stratified method for splitting the data will take care of uniform class distribution. This can be done Stratify option in train_test_split method.



```
from sklearn.model_selection import cross_val_score,StratifiedKFold,train_test_split
Data=scaled_data_feature_eng_im.iloc[:,1:]
Target=scaled_data_feature_eng_im.iloc[:,0]
X_train,X_test,Y_train,Y_test=train_test_split(Data,Target,random_state=0,test_size=0.2,stratify=Target)
```

3.2 Model Choosing

In this case we have to classify where a customer do the specific transaction or not. So, the target variabl e here is a categorical variable. For categorical variable we can use various classification models. Model having more AUC score will be our final model. Models built are

1. Logistic regression

2. K Nearest Neighbors

3. Random Forest

4. Naive Bayes

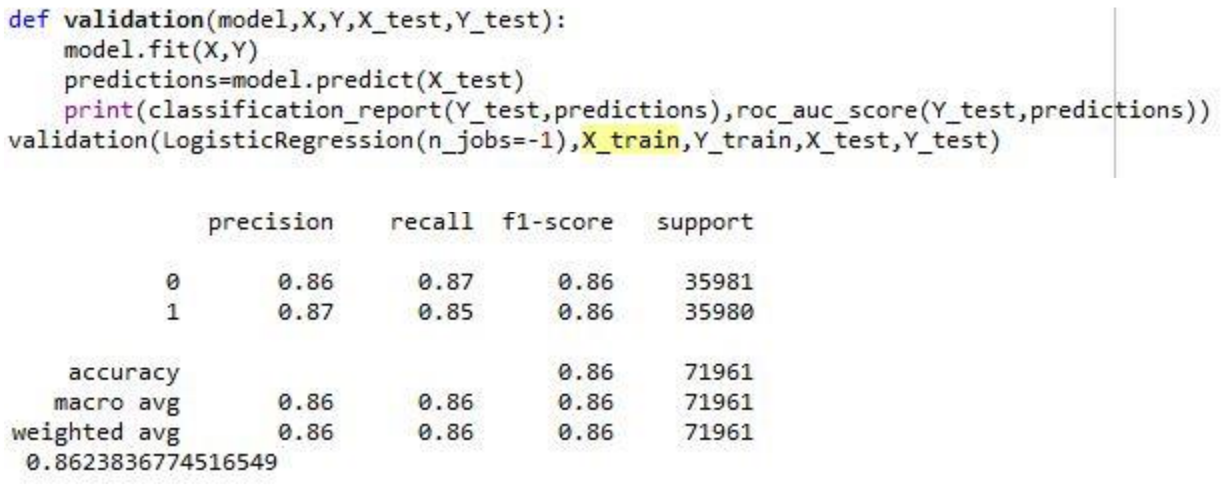
5. Xgboost

Where we have one dependent variables target such that model is used to classify into two classes 0 or 1.

3.2.1 Logistic regression

Here we are using the Logistic Algorithm for predicting the target variable which is best and simple algorithm for binary classification. Where it assumes data is linear and give non-linear output. Where we can use regularization which are ridge, lasso, elasticnet. Where logistic regression works on to reduce th e sum of maximum likelihood.

Let see the Classification report for logistic regression. Based on report we can move further on comparison with another model.



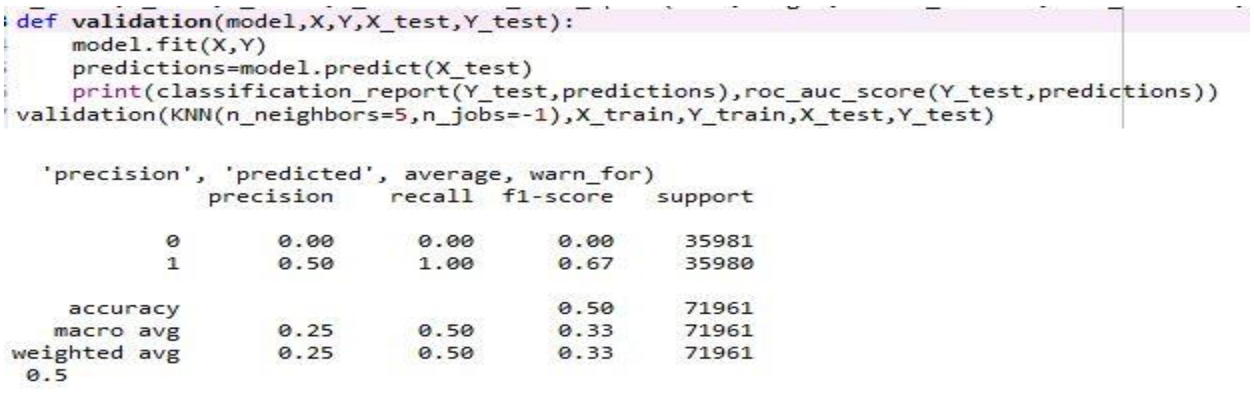
```
def validation(model,X,Y,X_test,Y_test):
    model.fit(X,Y)
    predictions=model.predict(X_test)
    print(classification_report(Y_test,predictions),roc_auc_score(Y_test,predictions))
validation(LogisticRegression(n_jobs=-1),X_train,Y_train,X_test,Y_test)
```

```
              precision    recall  f1-score   support

           0       0.86      0.87      0.86     35981
           1       0.87      0.85      0.86     35980

    accuracy                           0.86     71961
   macro avg       0.86      0.86      0.86     71961
weighted avg       0.86      0.86      0.86     71961
 0.8623836774516549
```

3.2.2 K Nearest Neighbors

Here we are using the K nearest neighbors Algorithm for predicting the target variable which is best and simple algorithm for binary classification. Where it is distance-based algorithm. It calculates distance by Euclidian and Manhattan method and take mode of nearest neighbors. Where KNN is effected by outlier s and curse of dimensionality.

Let see the Classification report for K nearest neighbors. Based on report we can move further on comparison with another model.



```
def validation(model,X,Y,X_test,Y_test):
    model.fit(X,Y)
    predictions=model.predict(X_test)
    print(classification_report(Y_test,predictions),roc_auc_score(Y_test,predictions))
validation(KNN(n_neighbors=5,n_jobs=-1),X_train,Y_train,X_test,Y_test)
```

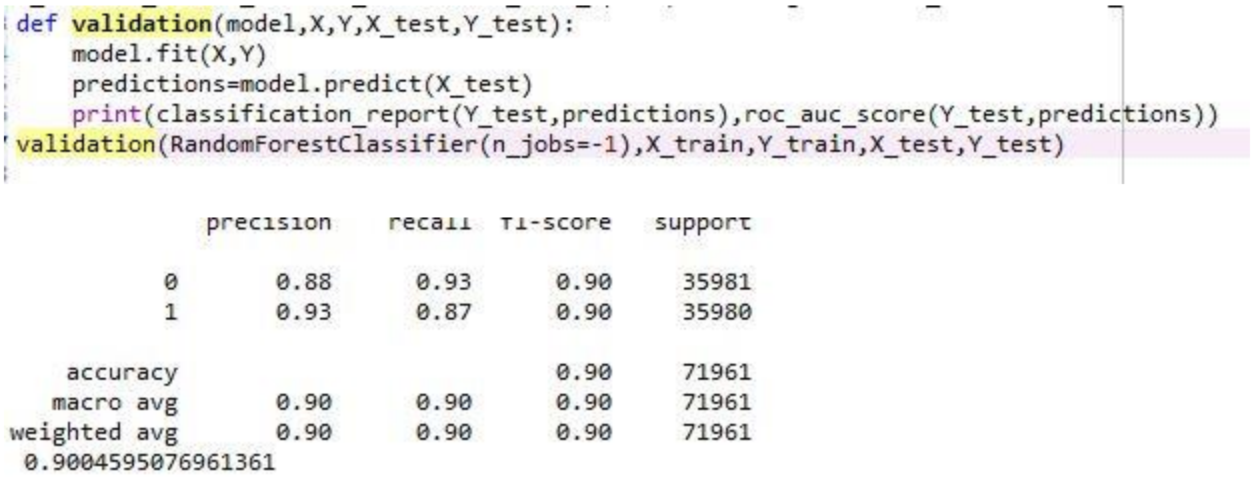
```
  'precision', 'predicted', average, warn_for)
              precision    recall  f1-score   support

           0       0.00      0.00      0.00     35981
           1       0.50      1.00      0.67     35980

    accuracy                           0.50     71961
   macro avg       0.25      0.50      0.33     71961
weighted avg       0.25      0.50      0.33     71961
 0.5
```

Knn algorithm is slow when compared to other algorithm

3.2.3 Random Forest

Here we are using the Random Forest Algorithm for predicting the target variable which is best for binar y classification. Where it is ensemble tree-based bagging algorithm.it will run n-trees parallelly with boot strap dataset created. And output will be take as mode of all tree output

Let see the Classification report for Random Forest. Based on report we can move further on comparison with another model.



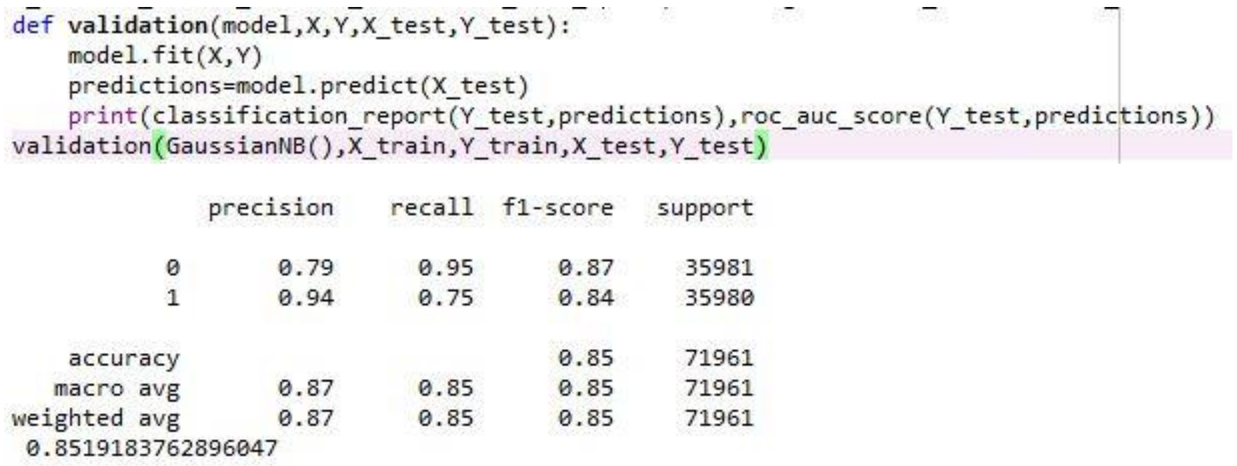
```
def validation(model,X,Y,X_test,Y_test):
    model.fit(X,Y)
    predictions=model.predict(X_test)
    print(classification_report(Y_test,predictions),roc_auc_score(Y_test,predictions))
validation(RandomForestClassifier(n_jobs=-1),X_train,Y_train,X_test,Y_test)
```

```
              precision    recall  f1-score   support

           0       0.88      0.93      0.90     35981
           1       0.93      0.87      0.90     35980

    accuracy                           0.90     71961
   macro avg       0.90      0.90      0.90     71961
weighted avg       0.90      0.90      0.90     71961
 0.9004595076961361
```

3.2.4 Naïve Bayes

Here we are using the Naïve Bayes Algorithm for predicting the target variable which is best and simple algorithm for binary classification. Where it is Bayesian statistics-based algorithm. It is based on conditio nal probabilities and assumes features are independent. where we have in feature correlation section th at variables are not that much correlated. It works fine if features are high. Where all variables are numerical, we will be using GaussianNB. It assumes numerical variables follows Gaussian distribution.

Let see the Classification report for Naïve Bayes. Based on report we can move further on comparison with another model.



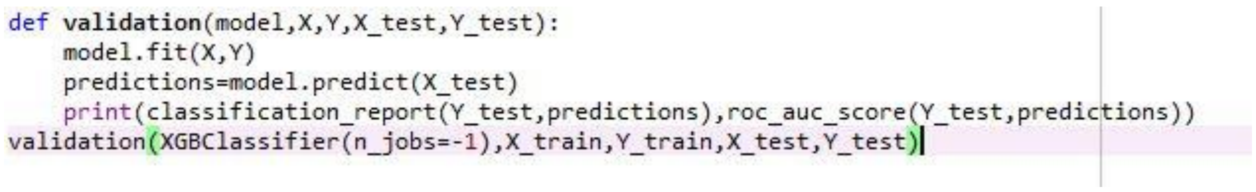
```
def validation(model,X,Y,X_test,Y_test):
    model.fit(X,Y)
    predictions=model.predict(X_test)
    print(classification_report(Y_test,predictions),roc_auc_score(Y_test,predictions))
validation(GaussianNB(),X_train,Y_train,X_test,Y_test)
```

```
              precision    recall  f1-score   support

           0       0.79      0.95      0.87     35981
           1       0.94      0.75      0.84     35980

    accuracy                           0.85     71961
   macro avg       0.87      0.85      0.85     71961
weighted avg       0.87      0.85      0.85     71961
 0.8519183762896047
```

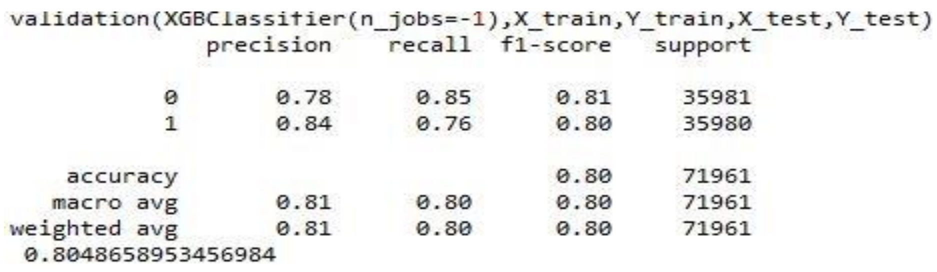
3.2.5 Xgboost

Here we are using the Xgboost Algorithm for predicting the target variable which is best for binary classif ication. Where it is ensemble tree-based boosting algorithm.It is also called as extreme gradient boosting algorithm. where by adding weak classifier will give the strong classifier. Where error rate of one classifier is passed to next classifier the data pints where first classifier failed to classify will be give high weight for next classifier.

Let see the Classification report for Random Forest. Based on report we can move further on compariso n with another model.



```
def validation(model,X,Y,X_test,Y_test):
    model.fit(X,Y)
    predictions=model.predict(X_test)
    print(classification_report(Y_test,predictions),roc_auc_score(Y_test,predictions))
validation(XGBClassifier(n_jobs=-1),X_train,Y_train,X_test,Y_test)
```



```
validation(XGBClassifier(n_jobs=-1),X_train,Y_train,X_test,Y_test)
              precision    recall  f1-score   support

           0       0.78      0.85      0.81     35981
           1       0.84      0.76      0.80     35980

    accuracy                           0.80     71961
   macro avg       0.81      0.80      0.80     71961
weighted avg       0.81      0.80      0.80     71961
 0.8048658953456984
```

3.3 Model Evaluation

Now that we have a few models for predicting the target class, we need to decide which one to choose. There are several criteria that exist for evaluating and comparing models. We can compare the models using any of the following criteria:

1. Predictive Performance

2. Interpretability

3. Computational Efficiency

In our case of Santander problem, Interpretability, do not hold much significance. Therefore we will use Predictive performance and Computational Efficiency as the criteria to compare and evaluate models.

Predictive performance can be measured by comparing Predictions of the models with AUC score , Precision, Recall scores,F1-score.

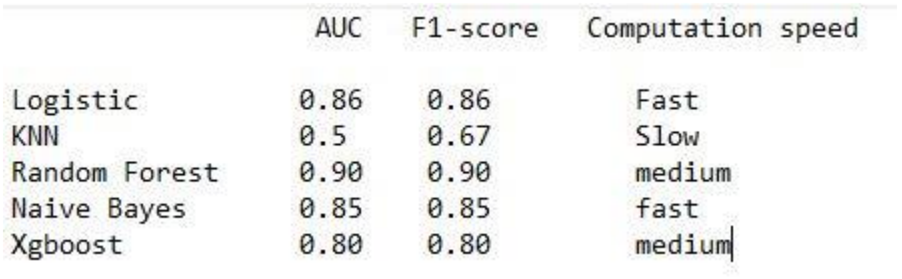
3.3.1 AUC score,Precision,Recall,F1-score

Precision: Precision is a measure which tells about how your model able to classify positive results.TP/TP +FP

Recall: It is nothing but how many actual positive cases were classified by the model which is TP/TP+FN

F1-score: which is weighted harmonic mean of precision and recall which takes consideration of false positives and false negatives also.it is best alternative to accuracy.

AUC score: Where Overall performance of a classifier is summarized over all threshold given by ROC (receiver operating characteristics).The model Performance is determined by looking at the area under the ROC curve which is AUC score. For best model AUC near to be 1.

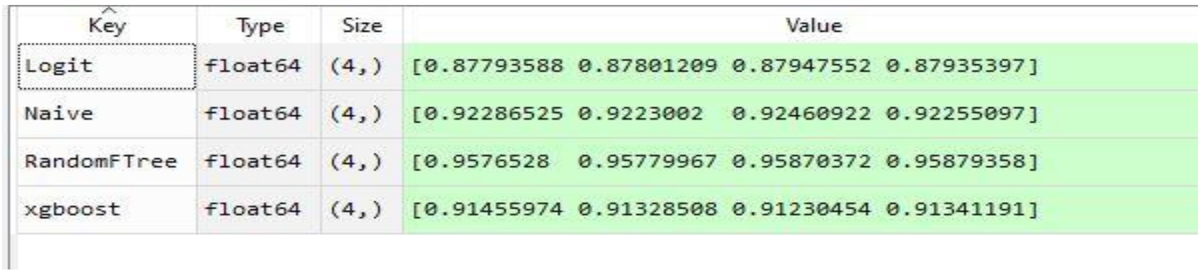


	AUC	F1-score	Computation speed
Logistic	0.86	0.86	Fast
KNN	0.5	0.67	Slow
Random Forest	0.90	0.90	medium
Naive Bayes	0.85	0.85	fast
Xgboost	0.80	0.80	medium

Based on Auc score random forest is good. So we considering this algorithm in further process.

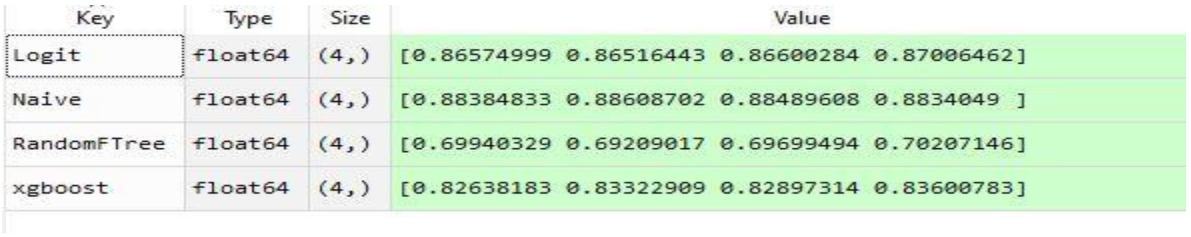
Where I have done some analysis by running the model on different like with adding additional features and with and with out adding artificial samples. Results are below:

When imbalance data with out adding features:



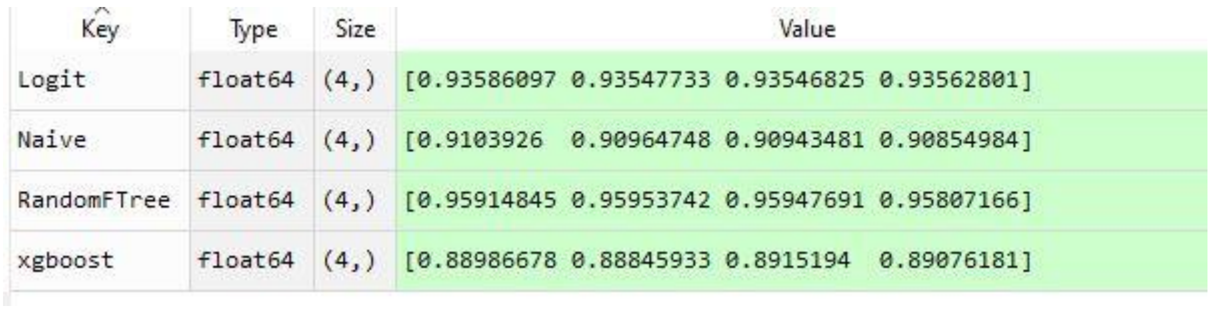
Key	Type	Size	Value
Logit	float64	(4,)	[0.87793588 0.87801209 0.87947552 0.87935397]
Naive	float64	(4,)	[0.92286525 0.9223002 0.92460922 0.92255097]
RandomFTree	float64	(4,)	[0.9576528 0.95779967 0.95870372 0.95879358]
xgboost	float64	(4,)	[0.91455974 0.91328508 0.91230454 0.91341191]

When features added:



Key	Type	Size	Value
Logit	float64	(4,)	[0.86574999 0.86516443 0.86600284 0.87006462]
Naive	float64	(4,)	[0.88384833 0.88608702 0.88489608 0.8834049]
RandomFTree	float64	(4,)	[0.69940329 0.69209017 0.69699494 0.70207146]
xgboost	float64	(4,)	[0.82638183 0.83322909 0.82897314 0.83600783]

When added imbalance and feature:



Key	Type	Size	Value
Logit	float64	(4,)	[0.93586097 0.93547733 0.93546825 0.93562801]
Naive	float64	(4,)	[0.9103926 0.90964748 0.90943481 0.90854984]
RandomFTree	float64	(4,)	[0.95914845 0.95953742 0.95947691 0.95807166]
xgboost	float64	(4,)	[0.88986678 0.88845933 0.8915194 0.89076181]

3.4.1 Boruta Algorithm:

Boruta is a feature ranking and selection algorithm which is based on random forest.

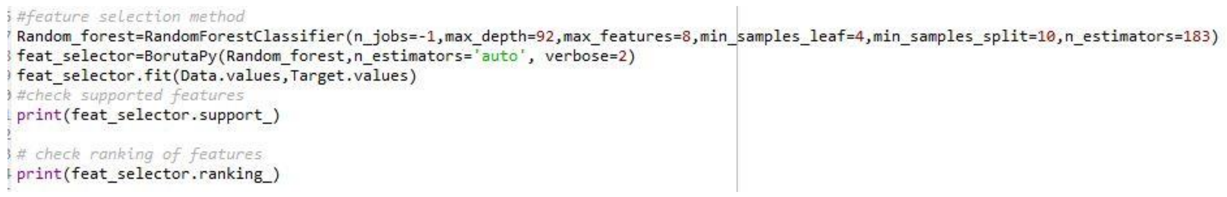
This algorithm is a kind of combination of both approaches I mentioned above.

1. Creating a “shadow” feature for each feature on our dataset, with the same feature values but only shuffled between the rows

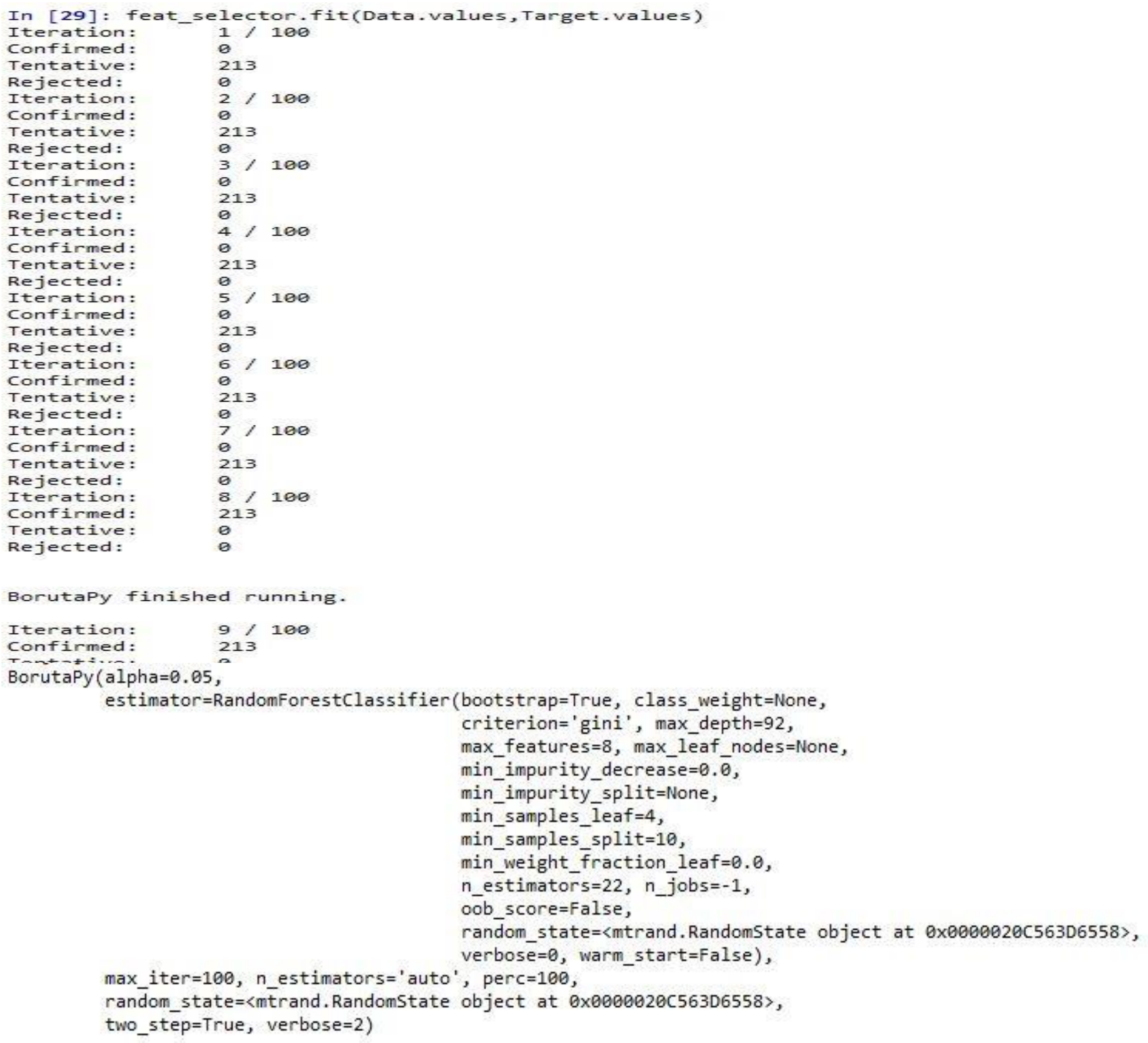
2. Run in a loop, until one of the stopping conditions:
   1. We are not removing any more features
   2. We removed enough features — we can say we want to remove 60% of our features
   3. We ran N iterations — we limit the number of iterations to not get stuck in an infinite loop

3. Run X iterations — which follow above steps

We are Boruta algorithm here to get consider the important features



```
#feature selection method
Random_forest=RandomForestClassifier(n_jobs=-1,max_depth=92,max_features=8,min_samples_leaf=4,min_samples_split=10,n_estimators=183)
feat_selector=BorutaPy(Random_forest,n_estimators='auto', verbose=2)
feat_selector.fit(Data.values,Target.values)
#check supported features
print(feat_selector.support_)

# check ranking of features
print(feat_selector.ranking_)
```

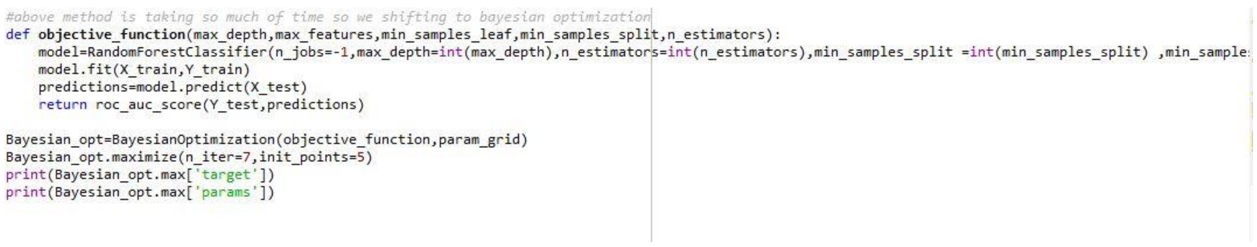


```
In [29]: feat_selector.fit(Data.values,Target.values)
Iteration:      1 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      2 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      3 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      4 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      5 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      6 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      7 / 100
Confirmed:      0
Tentative:      213
Rejected:       0
Iteration:      8 / 100
Confirmed:      213
Tentative:      0
Rejected:       0


BorutaPy finished running.

Iteration:      9 / 100
Confirmed:      213
Tentative:      0
BorutaPy(alpha=0.05,
         estimator=RandomForestClassifier(bootstrap=True, class_weight=None,
                                          criterion='gini', max_depth=92,
                                          max_features=8, max_leaf_nodes=None,
                                          min_impurity_decrease=0.0,
                                          min_impurity_split=None,
                                          min_samples_leaf=4,
                                          min_samples_split=10,
                                          min_weight_fraction_leaf=0.0,
                                          n_estimators=22, n_jobs=-1,
                                          oob_score=False,
                                          random_state=<mtrand.RandomState object at 0x0000020C563D6558>,
                                          verbose=0, warm_start=False),
         max_iter=100, n_estimators='auto', perc=100,
         random_state=<mtrand.RandomState object at 0x0000020C563D6558>,
         two_step=True, verbose=2)
```

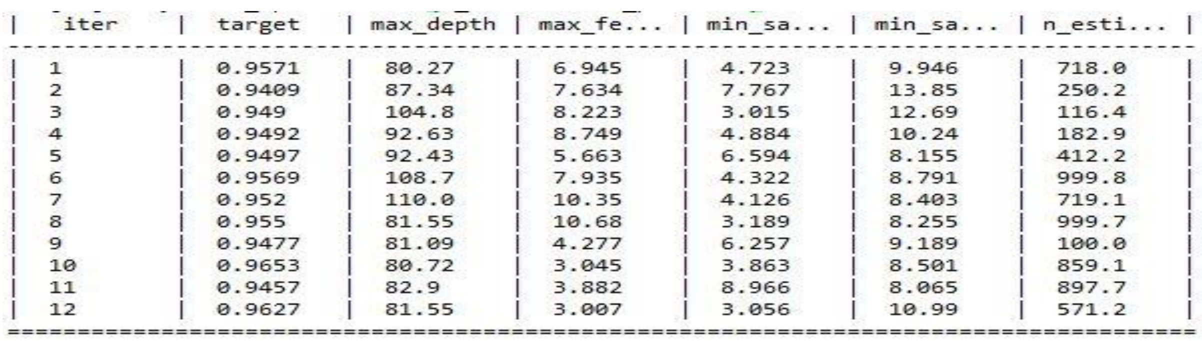
Where here we didn’t removed any variables where important or supported variables get by transform method.

3.5 Hyperparameter tuning

In python, we will use GridSearchCv for hyperparameter tuning but I ran that algorithm it took me one day to complete . GridSearchCv is effected by curse of dimensionality and it performs good when data points are less. RandomSerachCv is faster than grid search cv but computational expensive and both fails to memorize the previous values. RandomSerachCv may end up to local optima. So, for computational fast Bayesian optimization is used which the stores previous values to intelligently find the global optima. Bayesian optimization goal is to maximize the objective function. Which is defined by us.



```
#above method is taking so much of time so we shifting to bayesian optimization
def objective_function(max_depth,max_features,min_samples_leaf,min_samples_split,n_estimators):
    model=RandomForestClassifier(n_jobs=-1,max_depth=int(max_depth),n_estimators=int(n_estimators),min_samples_split =int(min_samples_split) ,min_sample
    model.fit(X_train,Y_train)
    predictions=model.predict(X_test)
    return roc_auc_score(Y_test,predictions)

Bayesian_opt=BayesianOptimization(objective_function,param_grid)
Bayesian_opt.maximize(n_iter=7,init_points=5)
print(Bayesian_opt.max['target'])
print(Bayesian_opt.max['params'])
```



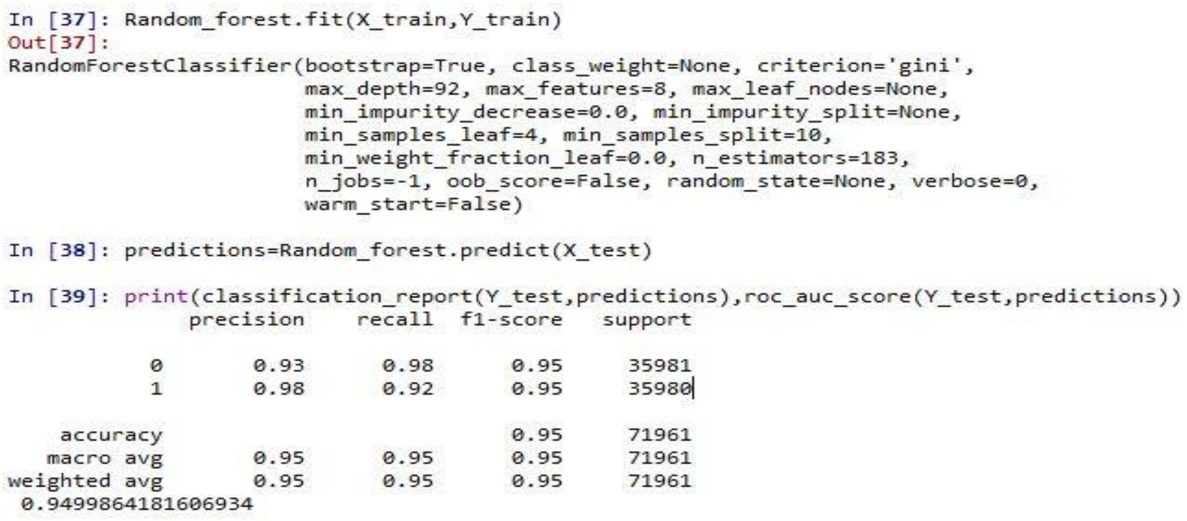
```
|   iter    |  target   | max_depth | max_fe... | min_sa... | min_sa... | n_esti... |
-------------------------------------------------------------------------------------
|  1        |  0.9571   |  80.27    |  6.945    |  4.723    |  9.946    |  718.0    |
|  2        |  0.9409   |  87.34    |  7.634    |  7.767    |  13.85    |  250.2    |
|  3        |  0.949    |  104.8    |  8.223    |  3.015    |  12.69    |  116.4    |
|  4        |  0.9492   |  92.63    |  8.749    |  4.884    |  10.24    |  182.9    |
|  5        |  0.9497   |  92.43    |  5.663    |  6.594    |  8.155    |  412.2    |
|  6        |  0.9569   |  108.7    |  7.935    |  4.322    |  8.791    |  999.8    |
|  7        |  0.952    |  110.0    |  10.35    |  4.126    |  8.403    |  719.1    |
|  8        |  0.955    |  81.55    |  10.68    |  3.189    |  8.255    |  999.7    |
|  9        |  0.9477   |  81.09    |  4.277    |  6.257    |  9.189    |  100.0    |
|  10       |  0.9653   |  80.72    |  3.045    |  3.863    |  8.501    |  859.1    |
|  11       |  0.9457   |  82.9     |  3.882    |  8.966    |  8.065    |  897.7    |
|  12       |  0.9627   |  81.55    |  3.007    |  3.056    |  10.99    |  571.2    |
=====================================================================================
```

Out of which 10th iteration gives best target score but seeing computation in consideration i choose 4th iteration parameters which also gives AUC score of 0.9492.

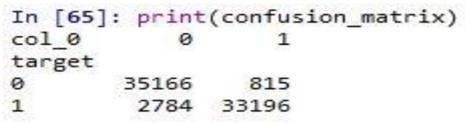
3.6 Final Model

After all tedious process I have chosen Random Forest with hyperparameter tuning

The code is below:



```
In [37]: Random_forest.fit(X_train,Y_train)
Out[37]:
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=92, max_features=8, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=4, min_samples_split=10,
                       min_weight_fraction_leaf=0.0, n_estimators=183,
                       n_jobs=-1, oob_score=False, random_state=None, verbose=0,
                       warm_start=False)

In [38]: predictions=Random_forest.predict(X_test)

In [39]: print(classification_report(Y_test,predictions),roc_auc_score(Y_test,predictions))
              precision    recall  f1-score   support

           0       0.93      0.98      0.95     35981
           1       0.98      0.92      0.95     35980

    accuracy                           0.95     71961
   macro avg       0.95      0.95      0.95     71961
weighted avg       0.95      0.95      0.95     71961
 0.9499864181606934
```



```
In [65]: print(confusion_matrix)
col_0       0      1
target
0       35166    815
1        2784  33196
```

Here we can see that TPR=0.92,TNR=0.98,FPR=0.022,FNR=0.0773

But Scores are pretty descent score with 0.9499 which AUC score.