Santander Customer Transaction Prediction

Aditya Kumar Ghosh

30 -7-2019

Contents

Introduction	
1.1 Problem Statement	2
1.2 Data	2
Methodology	
2.1 Pre Processing	4
2.1.2 Distribution of continuous variables (Train data set)	4-9
2.1.3 Distribution of categorical variables	10
2.1.4 Distribution of continuous variables (test data set)	11-15
2.1.5 Outlier Analysis	16-21
2.1.6 Feature Selection	21-23
Modelling	
3.1 Model Selection	24
3.2 Logistic Regression	24-25
3.2.1 Logistic Regression (Model 1)	25-26
3.2.2. Hyper Parameter tuning of Logistic regression	26-29
3.3 Decision Tree	30
3.3.1 Decision Tree Model 1	30-31
3.3.2. Hyper Parameter tuning of Decision Tree (Model 2)	32-33
3.3.3 Decision Tree model 3	34
3.4 Naive bayes	34
3.4.1 Naive bayes Model 1	34-36
3.4.2 Hyper Parameter tuning of Naive Bayes (Model 2)	36-37
3.5 XgBoost	37
3.5.1 XgBoost (Model 1)	37-39
3.5.2 Hyper Parameter tuning of XG boost (Model 2)	39-40
Conclusion	
4.1 Model Evaluation	41
4.2 Confusion Matrix:	41
4.2.1. Accuracy	42
4.2.2 Recall	42
4.2.3 precision	42
4.2.4 F-measure	43
4.2.5 AUC -ROC	43
4.12 Model Selection	43-44
Appendix B Python code	45-76
References	77

Chapter 1

Introduction

1.1 Problem Statement

In this challenge, we need to identify which customers will make a specific transaction in the future, irrespective of the amount of money transacted.

1.2 Data

Our task is to build a binary classification model to predict which customers will make a specific transaction in the future, irrespective of the amount of money transacted.
Training data set has dimension of (200000, 202)

Figure 1: Santander Customer Transaction training dataset (Id_code - var_18)

	ID_code	target	var_0	var_1	var_2	var_3	var_4	var_5	var_6	var_7	var_8	var_9	var_10	var_11	var_12	var_13	var_14	var_15	var_16	var_17	var_18
0	train_0	0	8.9255	-6.7863	11.9081	5.0930	11.4607	-9.2834	5.1187	18.6266	-4.9200	5.7470	2.9252	3.1821	14.0137	0.5745	8.7989	14.5691	5.7487	-7.2393	4.2840
1	train_1	0	11.5006	-4.1473	13.8588	5.3890	12.3622	7.0433	5.6208	16.5338	3.1468	8.0851	-0.4032	8.0585	14.0239	8.4135	5.4345	13.7003	13.8275	-15.5849	7.8000
2	train_2	0	8.6093	-2.7457	12.0805	7.8928	10.5825	-9.0837	6.9427	14.6155	-4.9193	5.9525	-0.3249	-11.2648	14.1929	7.3124	7.5244	14.6472	7.6782	-1.7395	4.7011
3	train_3	0	11.0604	-2.1518	8.9522	7.1957	12.5846	-1.8361	5.8428	14.9250	-5.8609	8.2450	2.3061	2.8102	13.8463	11.9704	6.4569	14.8372	10.7430	-0.4299	15.9426
4	train_4	0	9.8369	-1.4834	12.8746	6.6375	12.2772	2.4486	5.9405	19.2514	6.2654	7.6784	-9.4458	-12.1419	13.8481	7.8895	7.7894	15.0553	8.4871	-3.0680	6.5263
5	train_5	0	11.4763	-2.3182	12.6080	8.6264	10.9621	3.5609	4.5322	15.2255	3.5855	5.9790	0.8010	-0.6192	13.6380	1.2589	8.1939	14.9894	12.0763	-1.4710	6.7341
6	train_6	0	11.8091	-0.0832	9.3494	4.2916	11.1355	-8.0198	6.1961	12.0771	-4.3781	7.9232	-5.1288	-7.5271	14.1629	13.3058	7.8412	14.3363	7.5951	11.0922	21.1976
7	train_7	0	13.5580	-7.9881	13.8776	7.5985	8.6543	0.8310	5.6890	22.3262	5.0647	7.1971	1.4532	-6.7033	14.2919	10.9699	6.9190	14.2459	9.5376	-0.7226	5.1548
8	train_8	0	16.1071	2.4426	13.9307	5.6327	8.8014	6.1630	4.4514	10.1854	-3.1882	9.0827	0.9501	1.7982	14.0654	-3.0572	11.1642	14.8757	10.0075	-8.9472	3.8349
9	train_9	0	12.5088	1.9743	8.8960	5.4508	13.6043	-16.2859	6.0637	16.8410	0.1287	7.9682	0.8787	3.0537	13.9639	0.8071	9.9240	15.2659	11.3900	1.5367	5.4649

10 rows × 202 columns

Figure 2: Santander Customer Transaction training dataset (var_160 - var_177)

	var_160	var_161	var_162	var_163	var_164	var_165	var_166	var_167	var_168	var_169	var_170	var_171	var_172	var_173	var_174	var_175	var_176	var_177
.	15.4576	5.3133	3.6159	5.0384	6.6760	12.6644	2.7004	-0.6975	9.5981	5.4879	-4.7645	-8.4254	20.8773	3.1531	18.5618	7.7423	-10.1245	13.7241
.	29.4846	5.8683	3.8208	15.8348	-5.0121	15.1345	3.2003	9.3192	3.8821	5.7999	5.5378	5.0988	22.0330	5.5134	30.2645	10.4968	-7.2352	16.5721
.	13.2070	5.8442	4.7086	5.7141	-1.0410	20.5092	3.2790	-5.5952	7.3176	5.7690	-7.0927	-3.9116	7.2569	-5.8234	25.6820	10.9202	-0.3104	8.8438
.	31.8833	5.9684	7.2084	3.8899	-11.0882	17.2502	2.5881	-2.7018	0.5641	5.3430	-7.1541	-6.1920	18.2366	11.7134	14.7483	8.1013	11.8771	13.9552
.	33.5107	5.6953	5.4663	18.2201	6.5769	21.2607	3.2304	-1.7759	3.1283	5.5518	1.4493	-2.6627	19.8056	2.3705	18.4685	16.3309	-3.3456	13.5261
.	16.5552	5.3739	6.4487	11.5631	1.3847	14.9638	2.8455	-9.0953	3.8278	5.9714	-6.1449	-2.0285	18.4106	1.4457	21.8853	9.2654	-6.5247	10.7687
.	39.9599	5.5552	3.3459	9.2661	6.1213	23.7558	3.0298	5.9109	8.1035	6.1887	0.2619	-1.1405	25.1675	2.6965	17.0152	12.7942	-3.0403	8.1735
.	23.7765	5.4098	5.1402	10.7013	-8.2583	26.3286	2.6085	-10.9163	8.7362	5.2273	8.9519	-2.3522	6.1335	0.0876	19.5642	13.2008	-11.1786	17.3041
.	30.6740	5.7888	4.1180	9.1486	-5.2618	14.4422	2.6893	-9.5251	1.7455	5.9018	3.1838	-1.7865	4.9105	3.5803	32.9149	13.0201	-2.4845	11.0988
.	13.7379	5.4536	6.2403	17.1668	-5.3527	14.3780	2.4139	-9.1925	2.6859	5.8540	-3.0868	-1.2558	24.2683	-4.5382	18.2209	7.5652	6.3377	14.6223

Figure 3: Santander Customer Transaction training dataset (var_181 - var_199)

var_181	var_182	var_183	var_184	var_185	var_186	var_187	var_188	var_189	var_190	var_191	var_192	var_193	var_194	var_195	var_196	var_197	var_198	var_199
9.0164	3.0657	14.3691	25.8398	5.8764	11.8411	-19.7159	17.5743	0.5857	4.4354	3.9642	3.1364	1.6910	18.5227	-2.3978	7.8784	8.5635	12.7803	-1.0914
9.4878	-14.9100	9.4245	22.5441	-4.8622	7.6543	-15.9319	13.3175	-0.3566	7.6421	7.7214	2.5837	10.9516	15.4305	2.0339	8.1267	8.7889	18.3560	1.9518
9.3908	-13.2648	3.1545	23.0866	-5.3000	5.3745	-6.2660	10.1934	-0.8417	2.9057	9.7905	1.6704	1.6858	21.6042	3.1417	-6.5213	8.2675	14.7222	0.3965
8.4117	1.8986	7.2601	-0.4639	-0.0498	7.9336	-12.8279	12.4124	1.8489	4.4666	4.7433	0.7178	1.4214	23.0347	-1.2706	-2.9275	10.2922	17.9697	-8.9996
9.7685	4.8910	12.2198	11.8503	-7.8931	6.4209	5.9270	16.0201	-0.2829	-1.4905	9.5214	-0.1508	9.1942	13.2876	-1.5121	3.9267	9.5031	17.9974	-8.8104
11.1227	2.2257	6.4056	21.0550	-13.6509	4.7691	-8.9114	15.1007	2.4286	-6.3068	6.6025	5.2912	0.4403	14.9452	1.0314	-3.6241	9.7670	12.5809	-4.7602
10.0570	15.7862	3.3593	11.9140	-4.2870	7.5015	-29.9763	17.2867	1.8539	8.7830	6.4521	3.5325	0.1777	18.3314	0.5845	9.1104	9.1143	10.8869	-3.2097
10.5478	6.9736	6.9724	24.0369	-4.8220	8.4947	-5.9076	18.8663	1.9731	13.1700	6.5491	3.9906	5.8061	23.1407	-0.3776	4.2178	9.4237	8.6624	3.4806
7.7291	-11.4027	2.0696	-1.7937	-0.0030	11.5024	-18.3172	13.1403	0.7014	1.4298	14.7510	1.6395	1.4181	14.8370	-1.9940	-1.0733	8.1975	19.5114	4.8453
11.3457	-9.6774	10.3382	19.0645	-7.6785	6.7580	-21.6070	20.8112	-0.1873	0.5543	6.3160	1.0371	3.6885	14.8344	0.4467	14.1287	7.9133	16.2375	14.2514

As you can see in the table below we have the following 202 variables named (var_0 to var_199) , using which we have to correctly classify our *target* variable

Table 1: Predictor Variables

S.No	**Predictor**
1	‘var_99’
2	‘var_11’
3	'var_0'
4	'var_1'
5	'var_2'

6	'var_3'
7	'var_4'
8	,'var_5'
9	'var_6’
…….	…..
11	'var_196',
12	'var_197'
13	'var_198'

Chapter 2

Methodology

2.1 Pre Processing

Data preprocessing is a data mining technique that involves transforming raw data into an understandable format. Real-world data is often incomplete, inconsistent, and/or lacking in certain behaviors or trends, and is likely to contain many errors. Data preprocessing is a proven method of resolving such issues. Data preprocessing prepares raw data for further processing.

2.1.2 Distribution of continuous variables (Train data set)

Data visualization is an important part of any data analysis. It helps us to recognize relations between variables and also to find which variables are significant or which variables can affect the predicted variable

Figure 4 : Plot showing distribution of var_1 to var_19

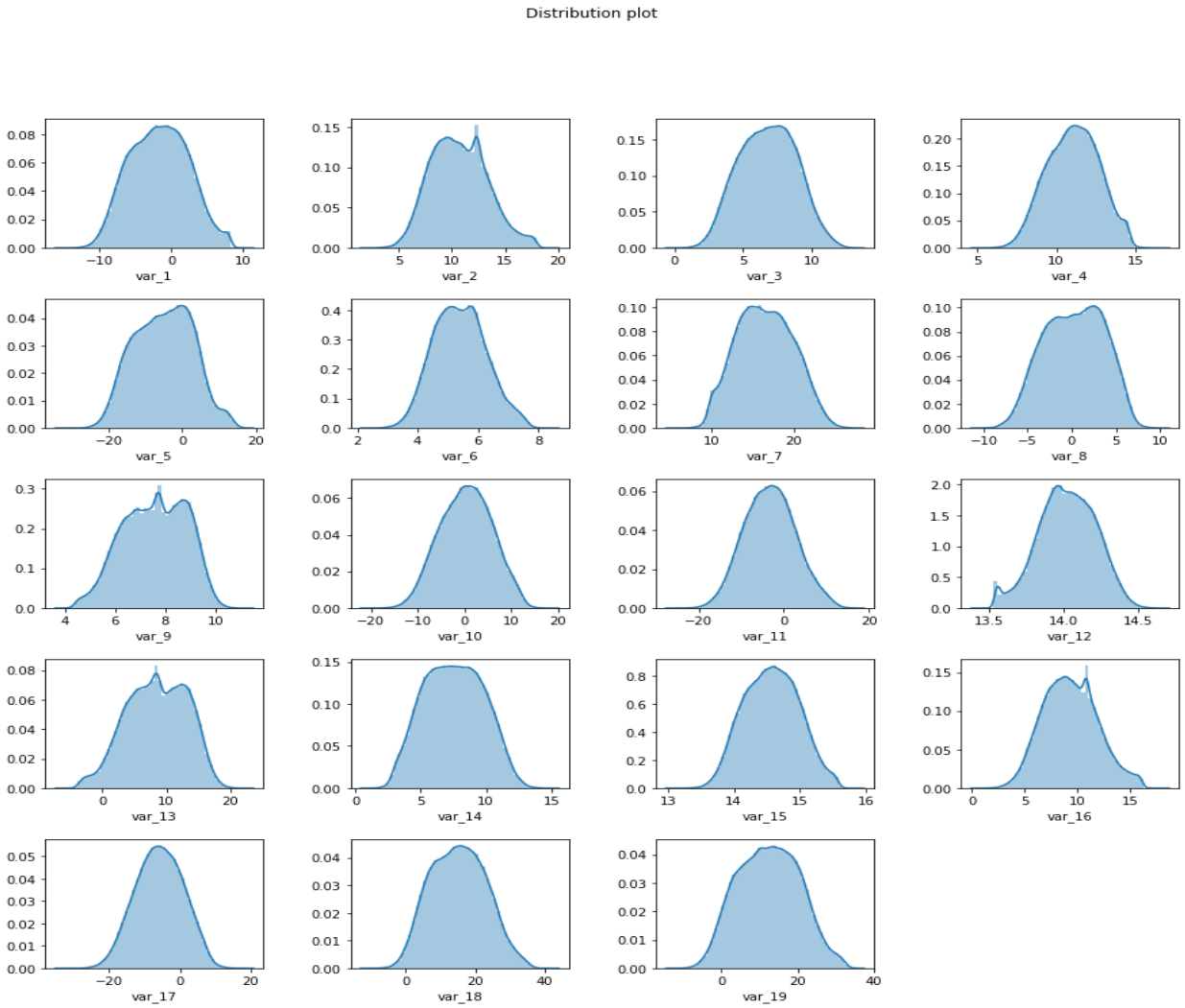


Figure 5 : Plot showing distribution of var_25 to var_44

Distribution plot

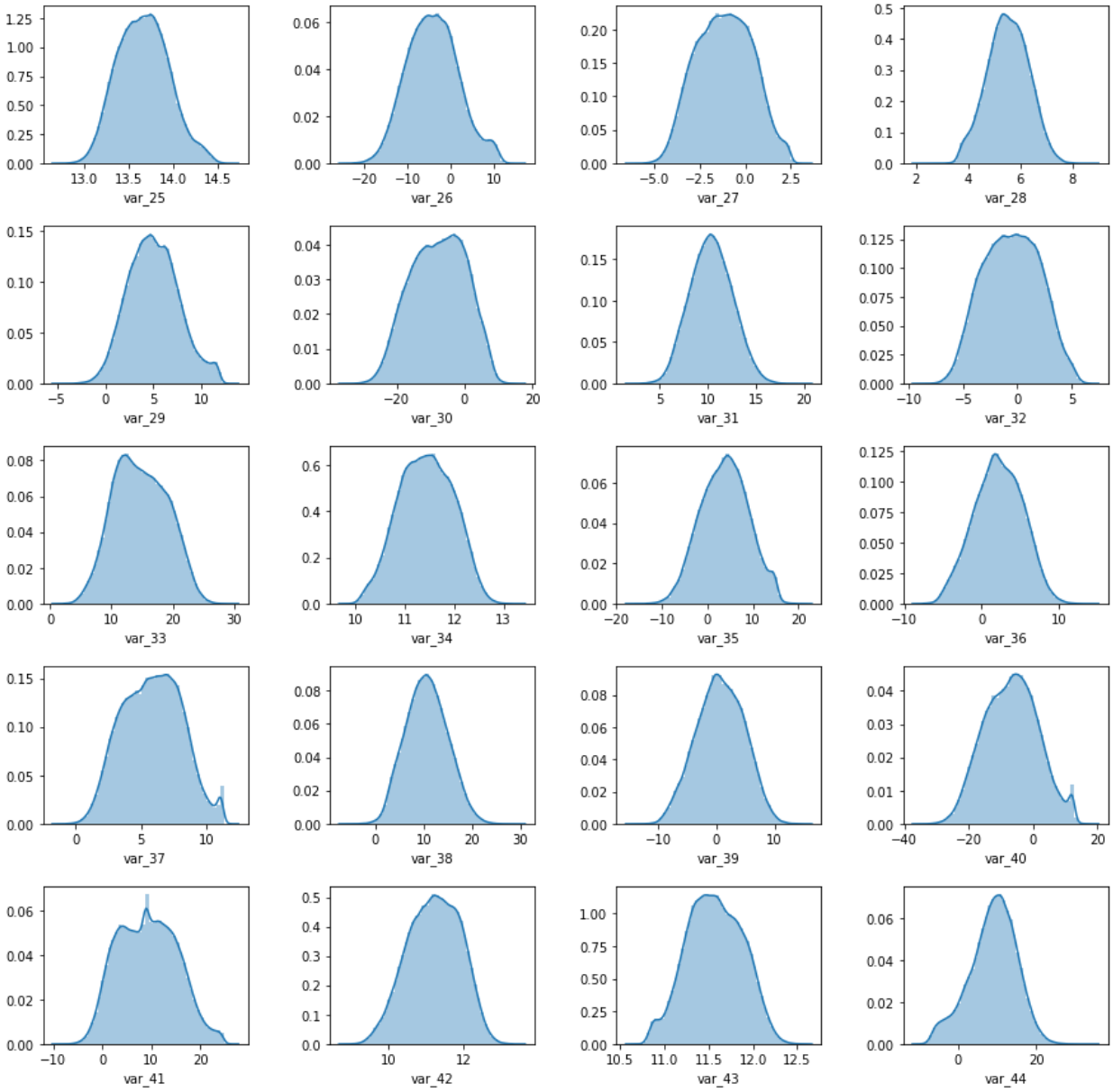


Figure 6 : Plot showing distribution of var_71 to var_89

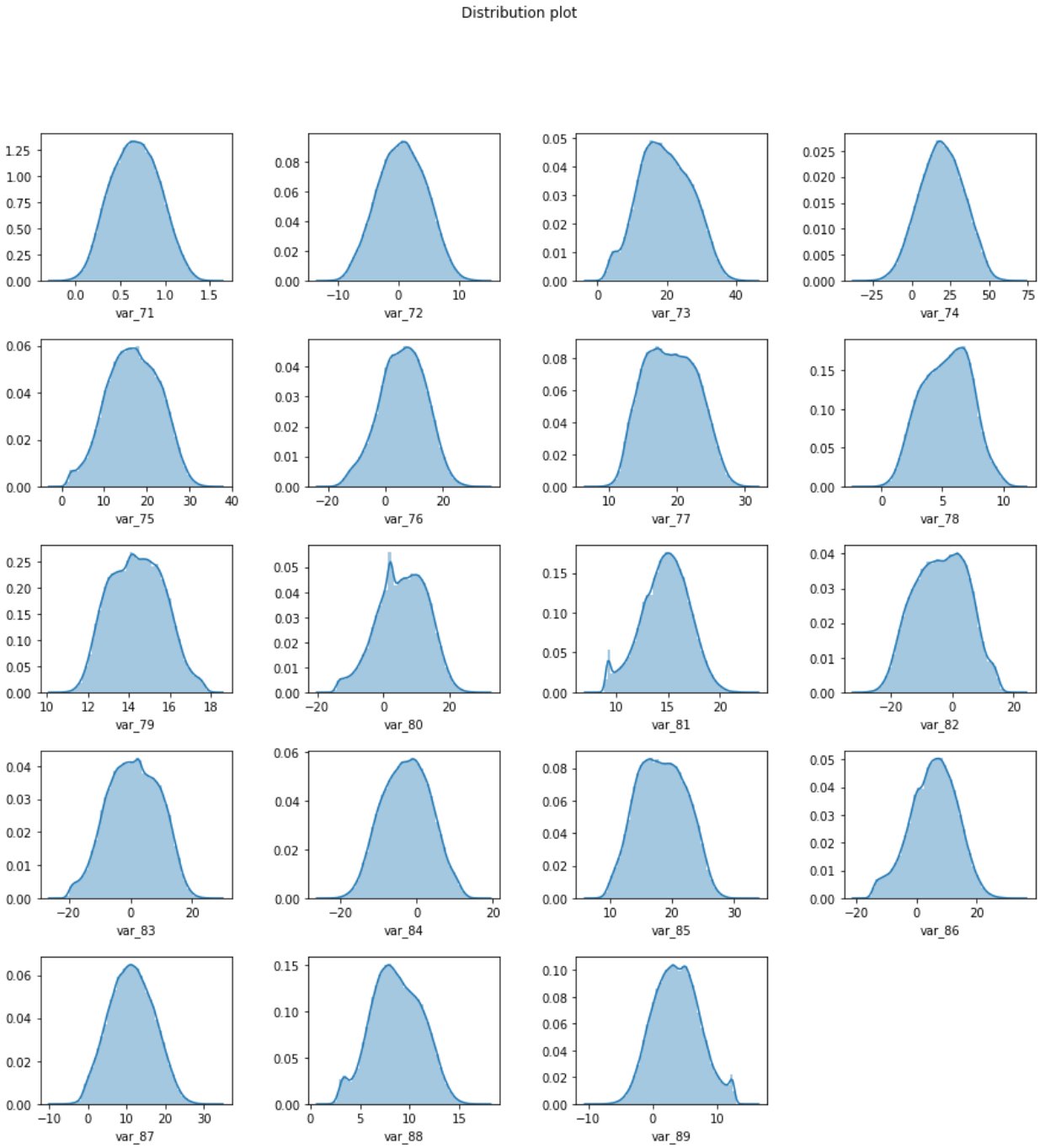


Figure 7 : Plot showing distribution of var_103 to var_121

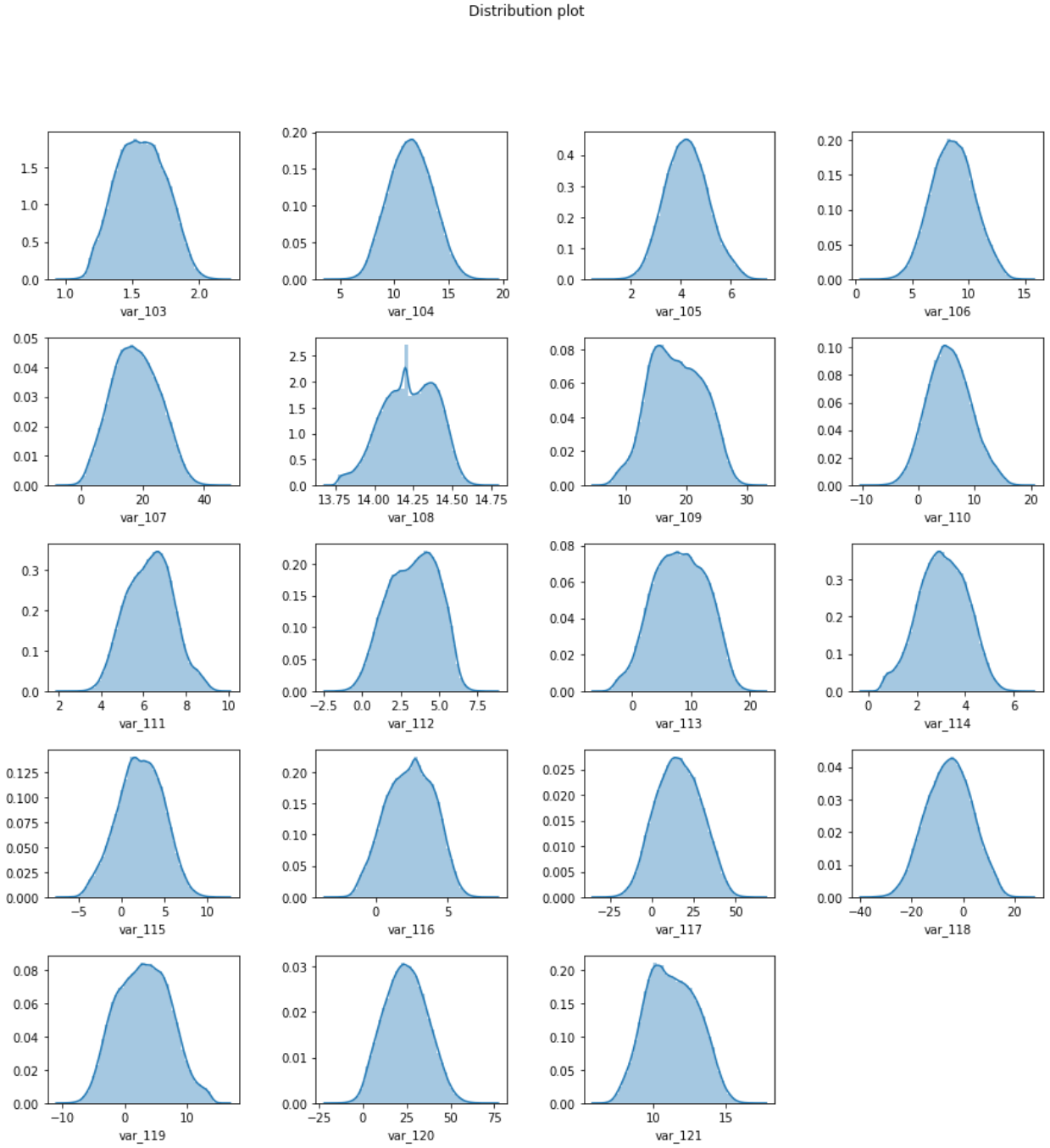


Figure 8 : Plot showing distribution of var_150 to var_169

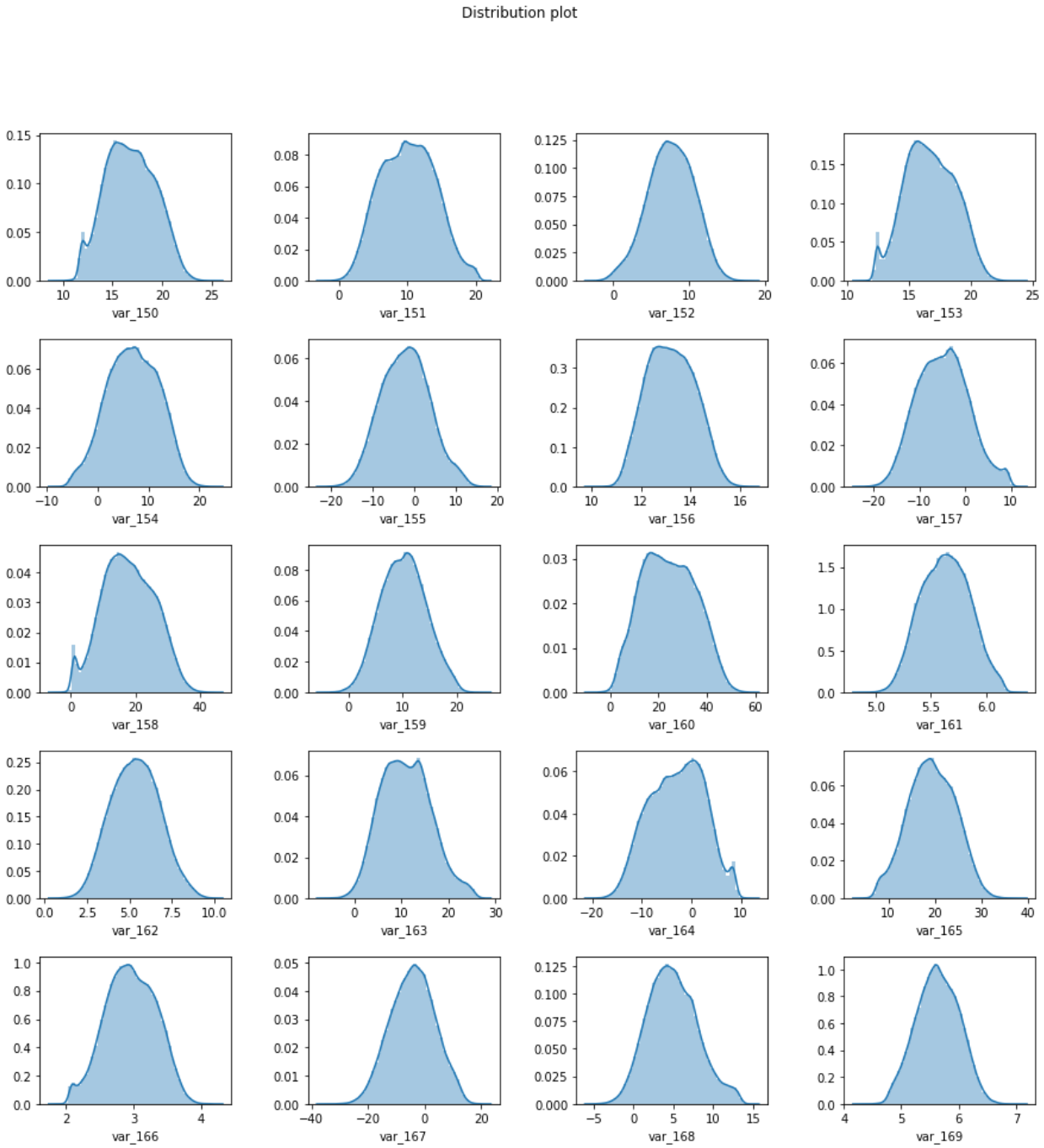
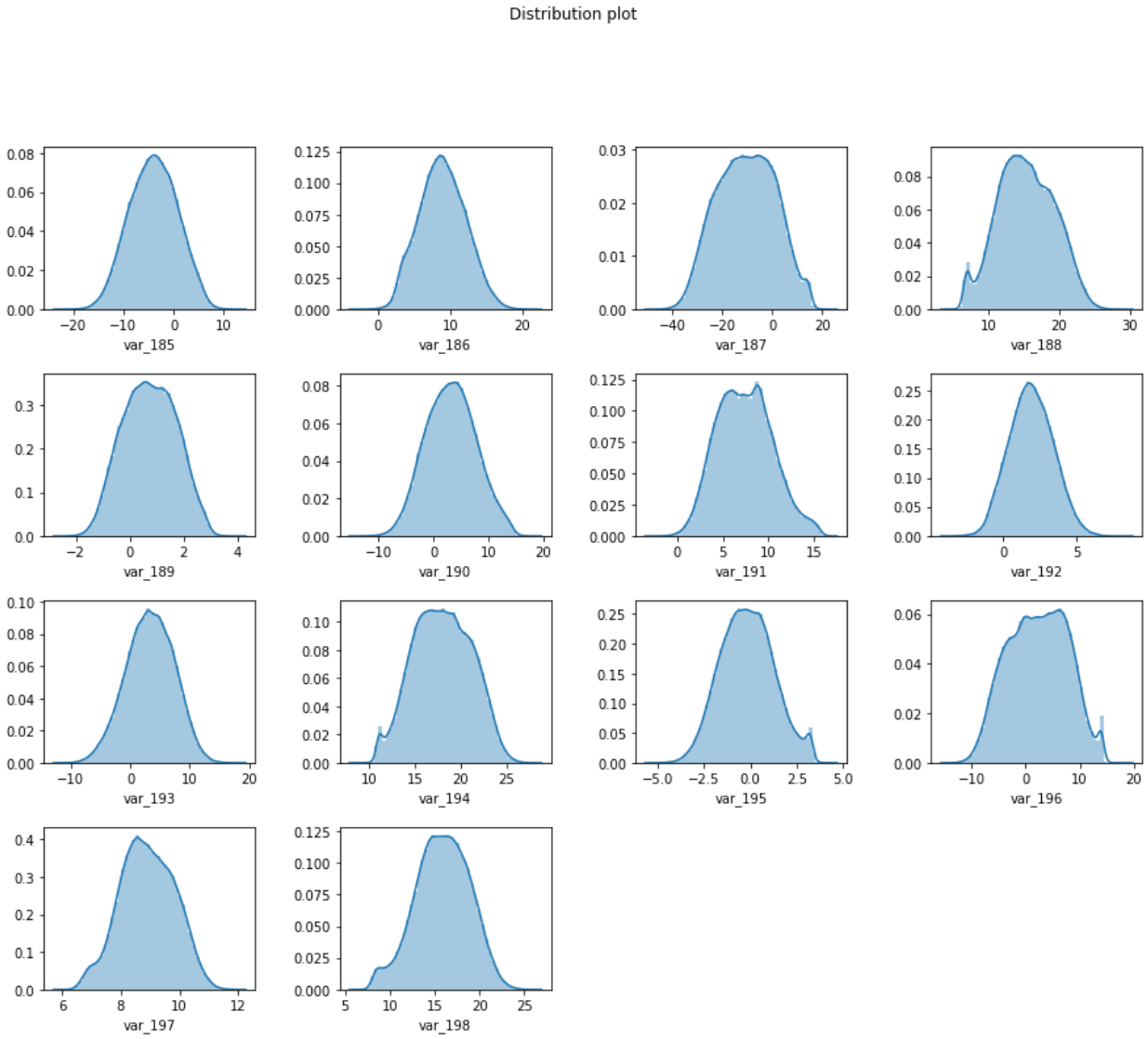


Figure 9 : Plot showing distribution of var_185 to var_198

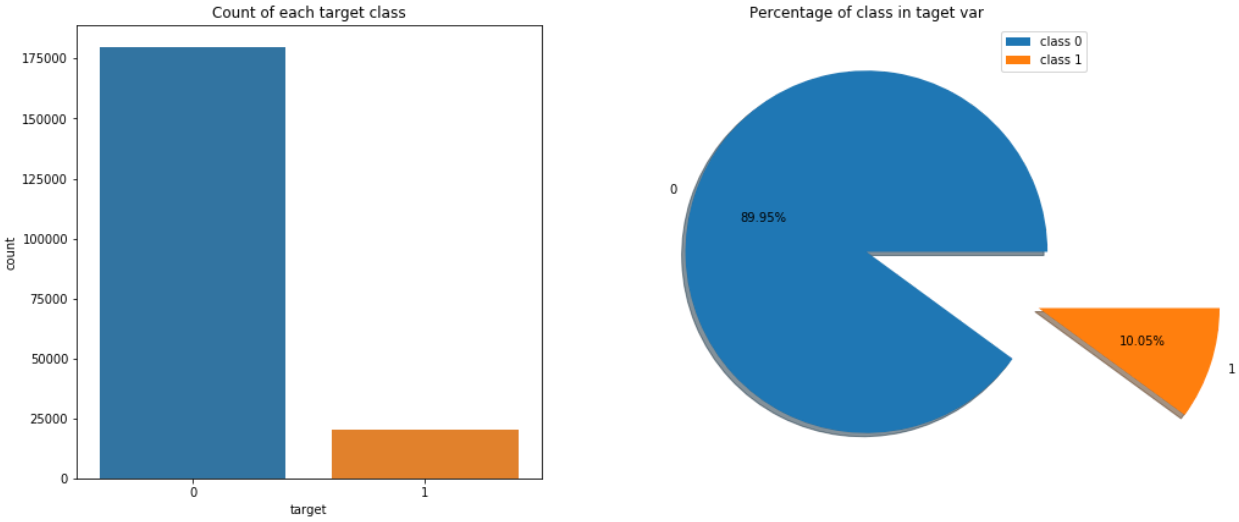


Observation Variable Distribution Train data

1. Almost all Distributions of variables are normal

2.1.3 Distribution of categorical variables

Figure 10 : Plot showing distribution of target variable in training data



Observations

- There is high imbalance between classes in target class
- The number of observations in class 0 is more the class 1 (90% class 0 ,10% class 1)
- The number of people that will not make transaction is more than the one who will make

2.1.4 Distribution of continuous variables (test data set)

Figure 11 : Plot showing distribution of var_1 to var_19

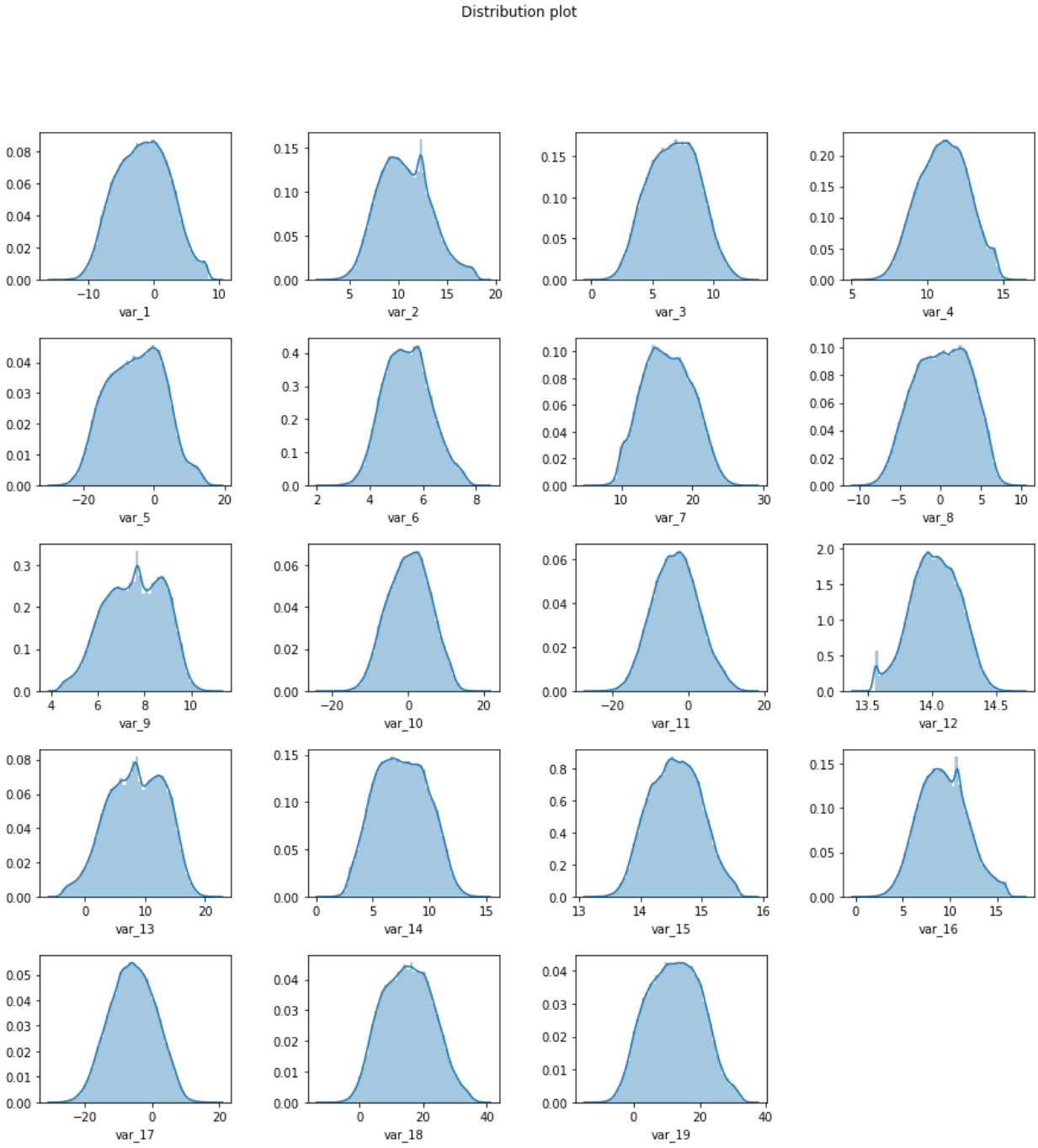


Figure 12 : Plot showing distribution of var_25 to var_44

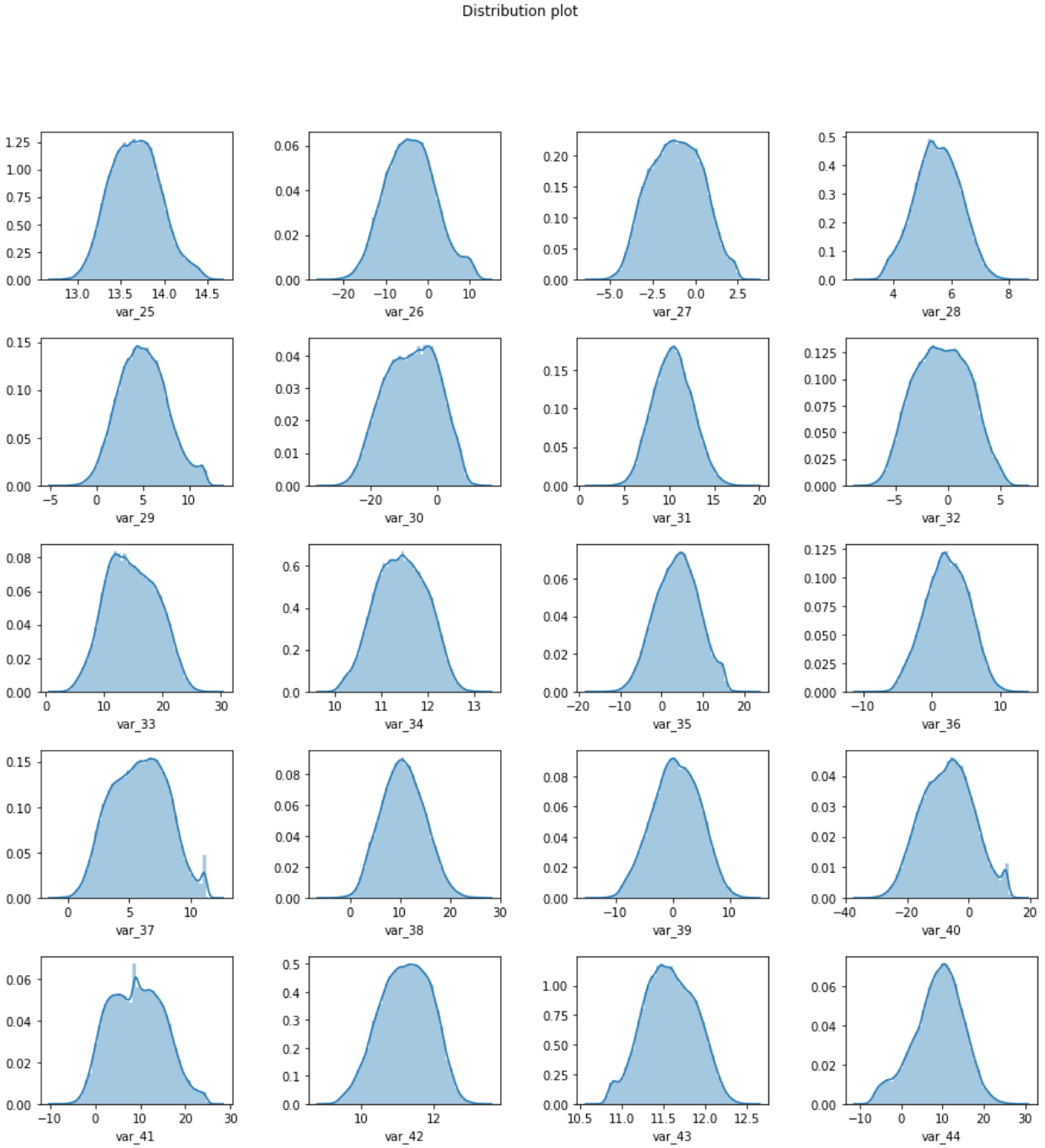


Figure 13 : Plot showing distribution of var_1 to var_71

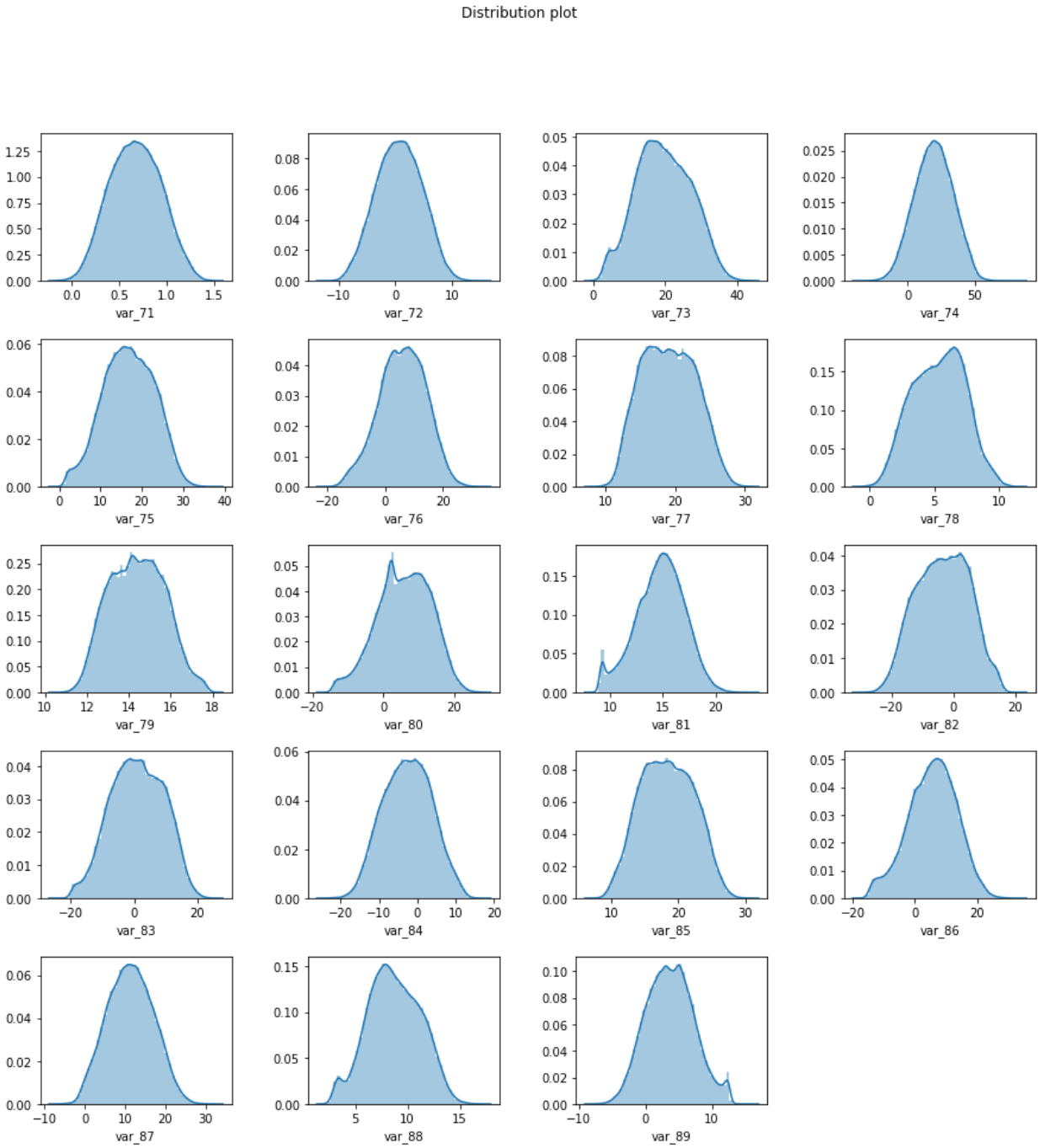


Figure 14 : Plot showing distribution of var_150 to var_169

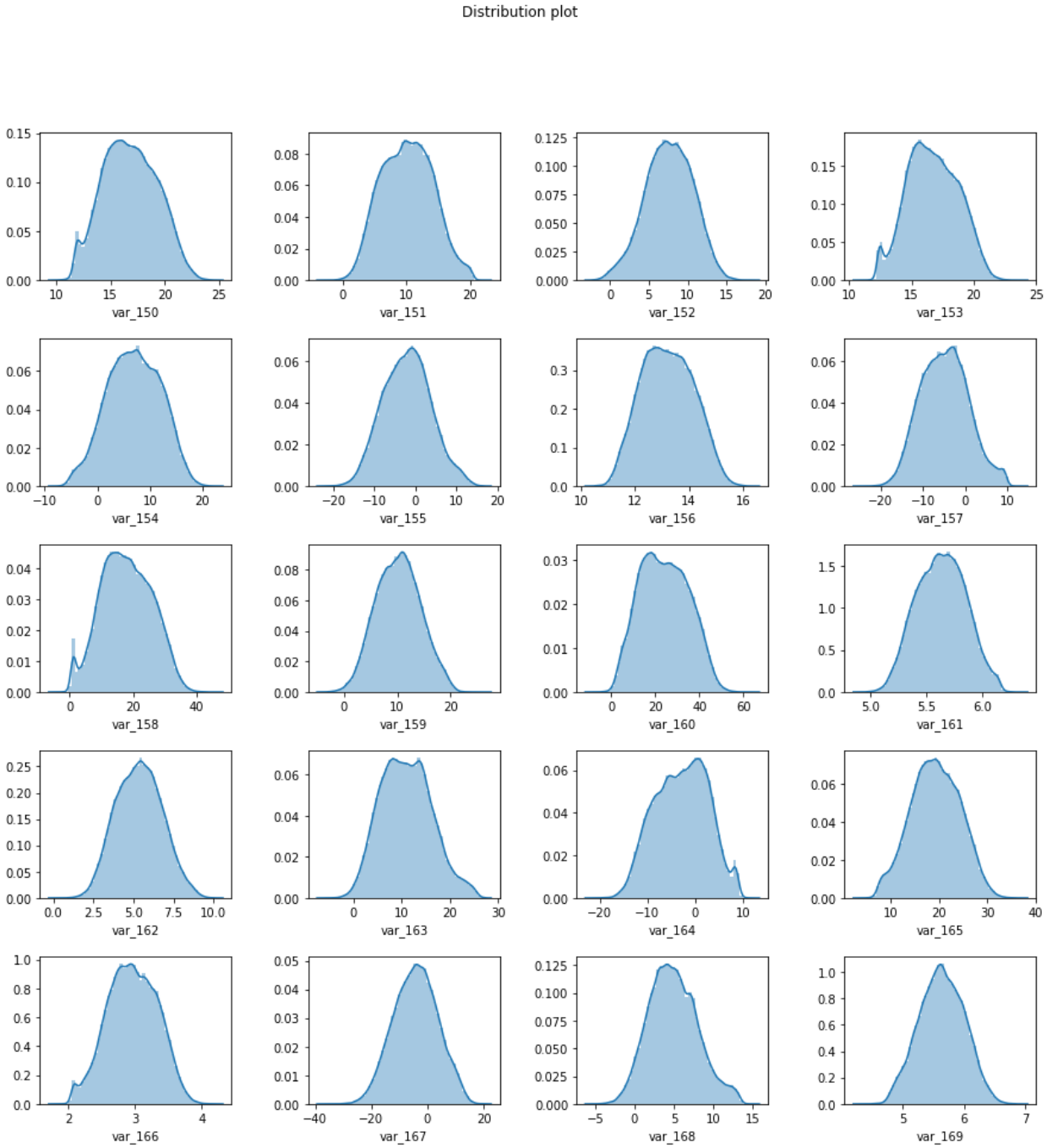
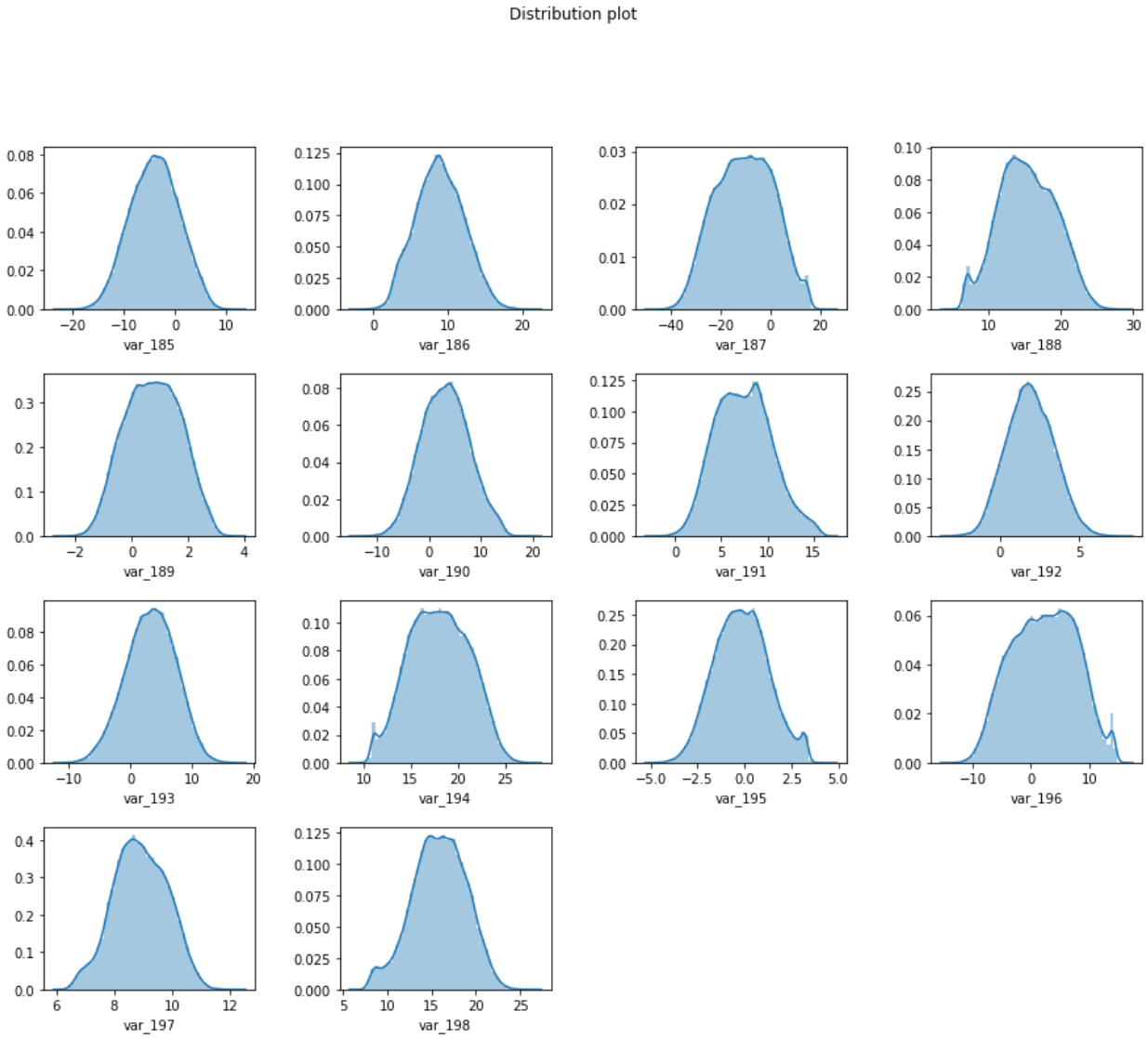


Figure 15 : Plot showing distribution of var_185 to var_198



Observation distribution Test data

1. Almost all Distributions of variables are normal
2. Test data is very similar to train data in terms of distribution

2.1.5 Outlier Analysis

In statistics, an outlier is an observation point that is distant from other observations.
We can clearly observe from these probability distributions that most of the variables are skewed , for example *casual ,hum,wind speed* .The skew in these distributions can be most likely explained by the presence of outliers and extreme values in the data

In descriptive statistics, a box plot is a method for graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles, hence the terms box-and-whisker plot and box-and-whisker diagrams. Outliers may be plotted as individual points

Any observations outside of upper fence and lower fence is treated as an outlier , we can either remove the outlier or impute values using mean ,mode ,median or knn imputation ,removing outliers can make data set small as whole row is removed this won't have any effect if we have large dataset but if you have small dataset removing outliers can make already small dataset more small

Figure 16 : Outlier Analysis of continuous variables (var_0 to var_19)

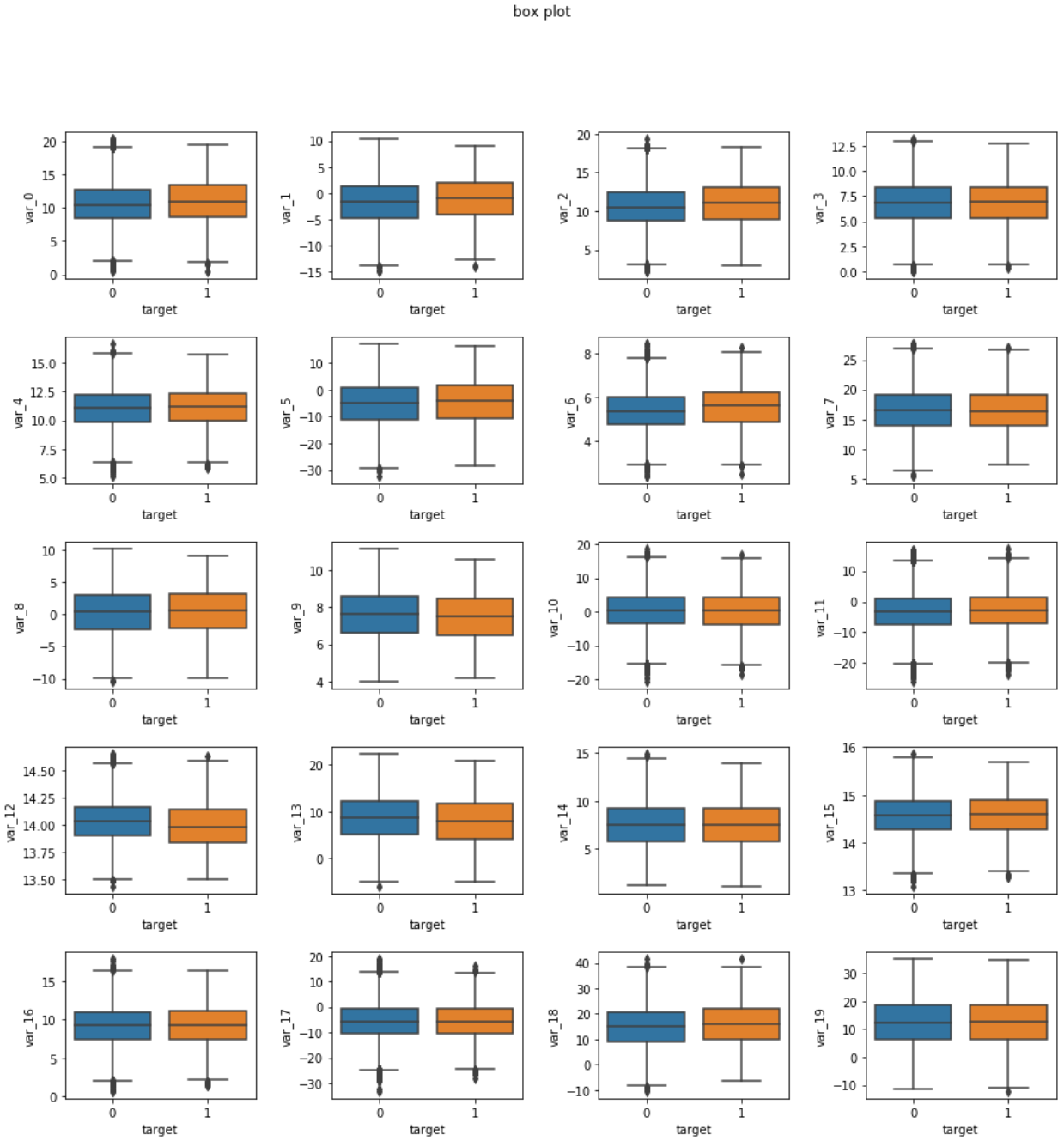


Figure 17 : Outlier Analysis of continuous variables (var_21 to var_39)

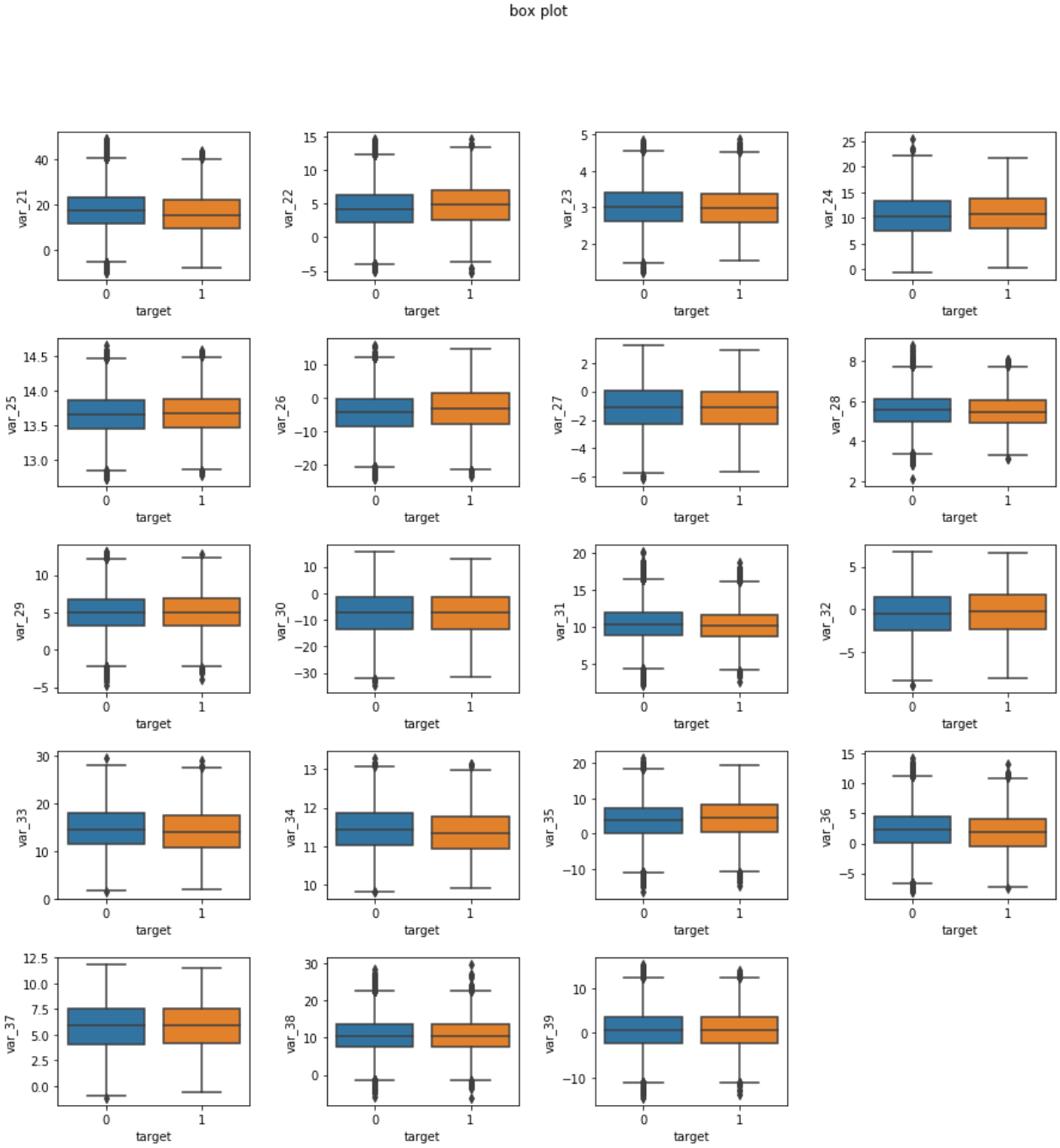


Figure 18 : Outlier Analysis of continuous variables (var_60 to var_78)

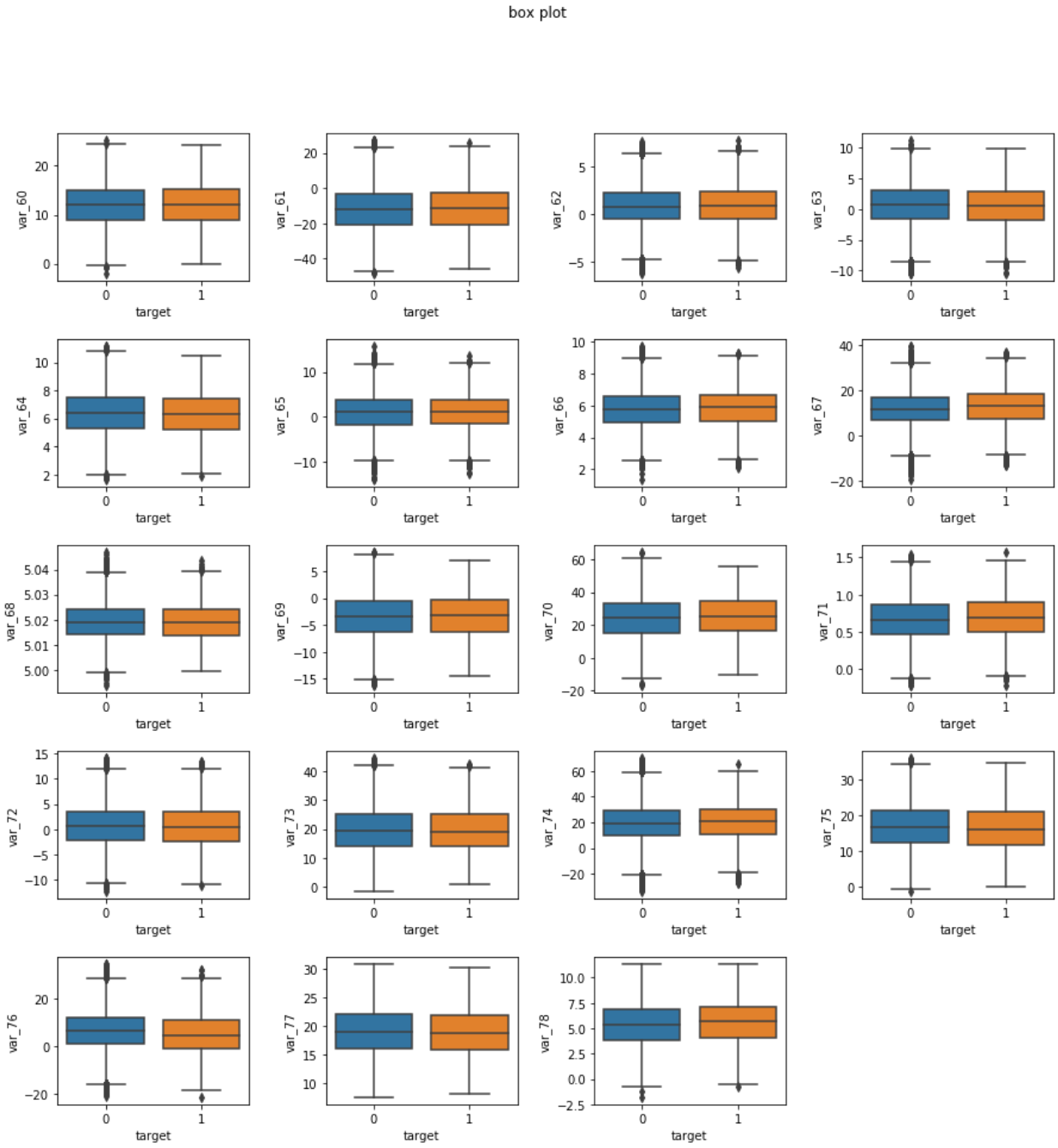


Figure 19 : Outlier Analysis of continuous variables (var_100 to var_118)

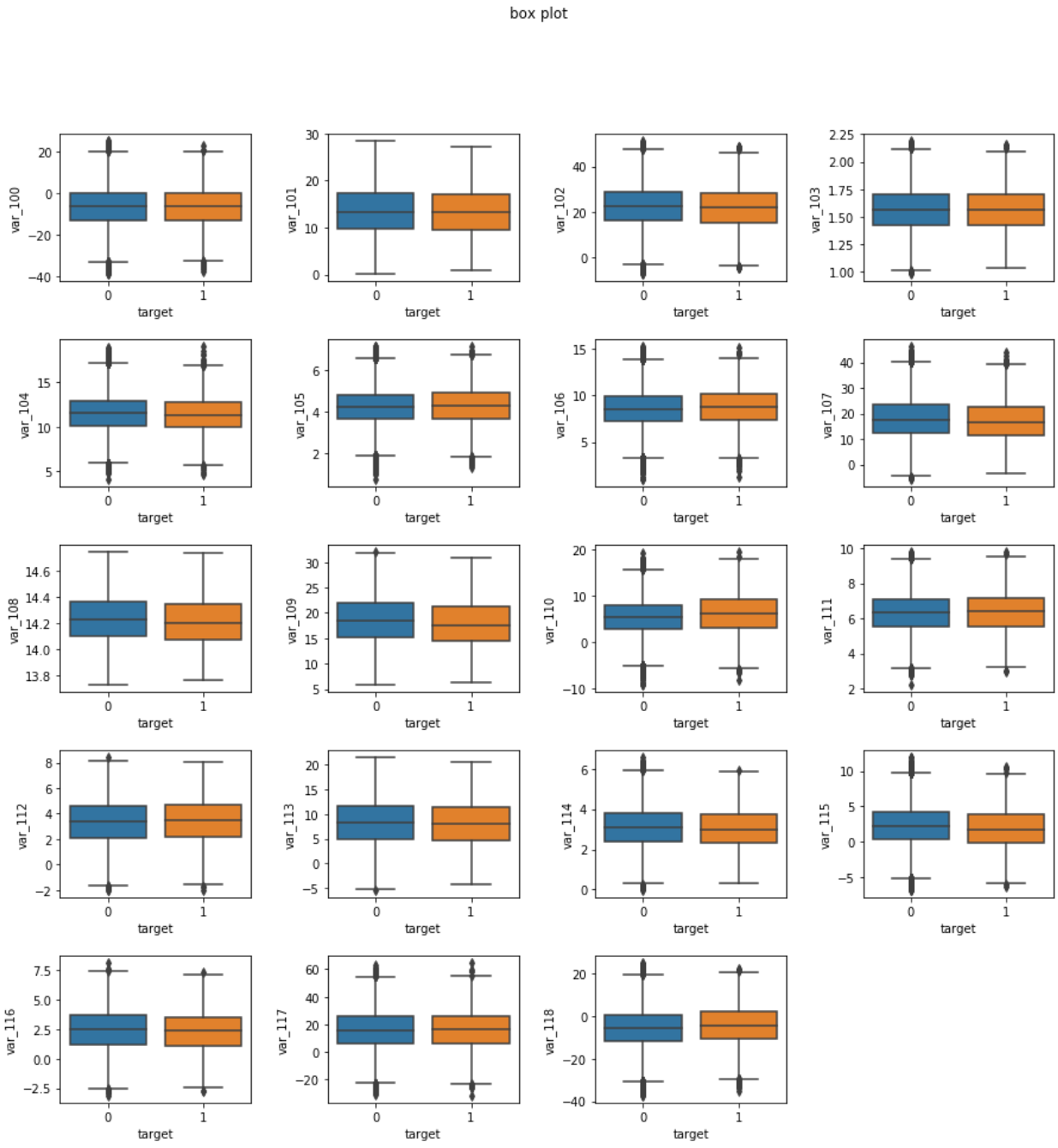
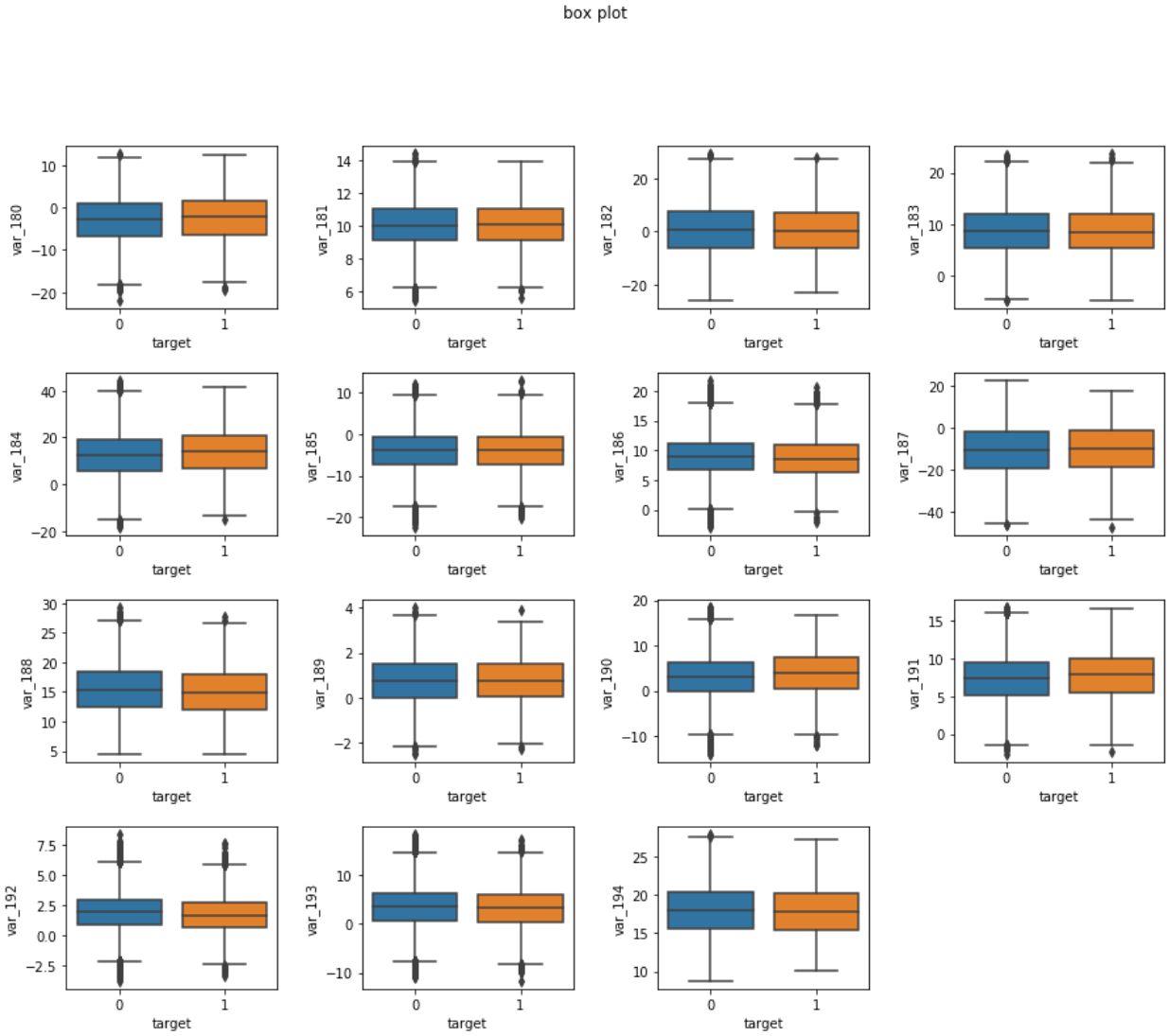


Figure 20 : Outlier Analysis of continuous variables (var_180 to var_194)



2.1.6 Feature Selection

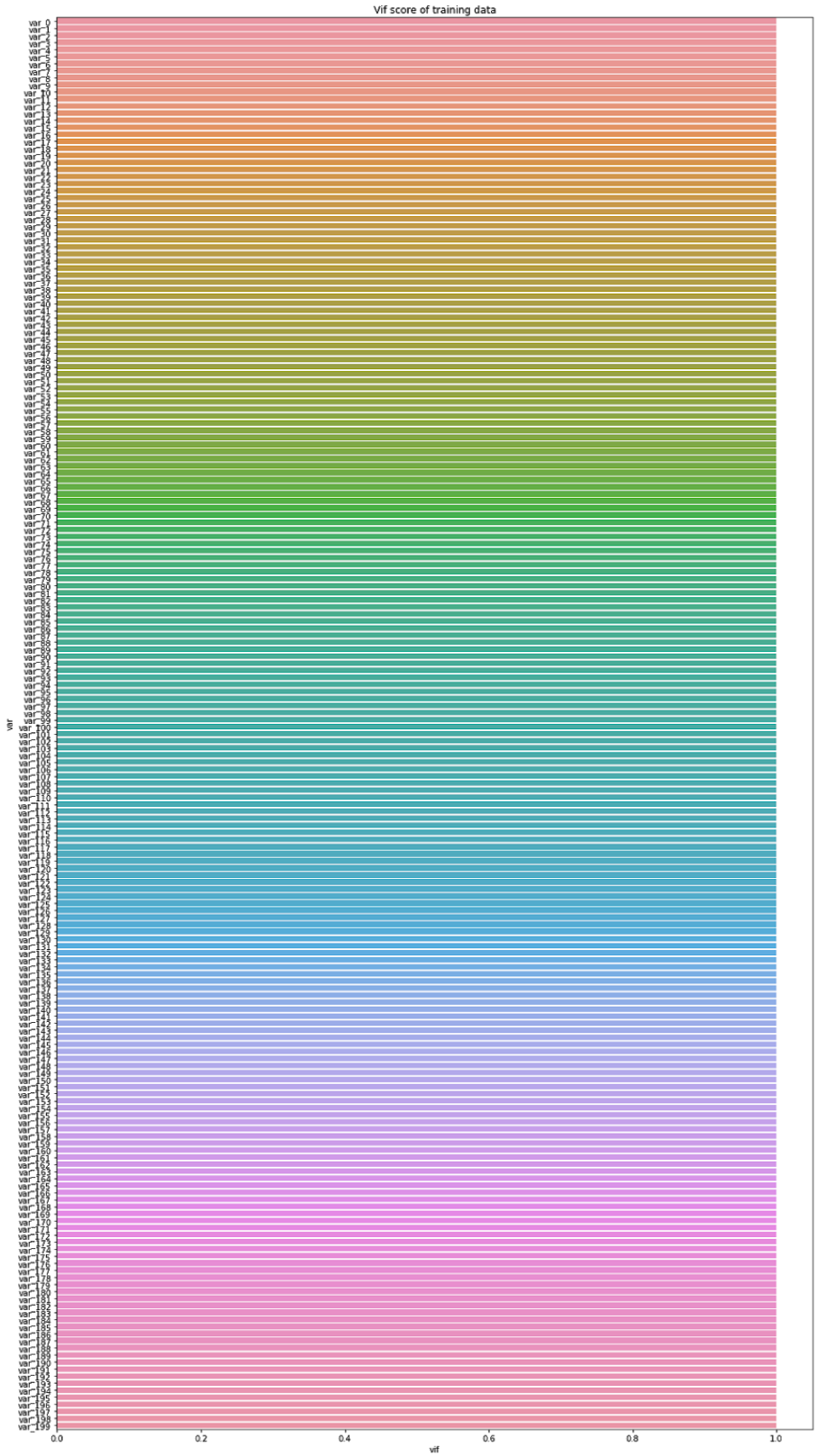
Feature Selection is one of the core concepts in machine learning which hugely impacts the performance of your model. The data features that you use to train your machine learning models have a huge influence on the performance you can achieve.

Feature Selection is the process where you automatically or manually select those features which contribute most to your prediction variable or output in which you are interested in.

Benefits are of feature selection first Reduces Overfitting Less redundant data means less opportunity to make decisions based on noise.Improves Accuracy Less misleading data means modeling accuracy improves.Reduces Training Time fewer data points reduce algorithm complexity and algorithms train faster.

Features are selected based on their scores in various statistical tests for their correlation with the outcome variable. Correlation plot is used to find out if there is any multicollinearity between variables. The highly collinear variables are dropped and then the model is executed.

Figure 21 : Vif plot for independent variables



Observation Multicollinearity

1. Vif of all variable is close to 1 .

Chapter 3: Modelling

3.1 Model Selection

During analysis of dataset we have come to know that classification model will be most suited for modeling as our target variable is binary, we are basically looking at binary classification problem ,our training data set is imbalanced 90 percent of case 0 and 10 percent of case 1 . We are basically dealing with class imbalanced problem ,It is the problem in machine learning where the total number of a class of data (positive) is far less than the total number of another class of data (negative). This problem is extremely common in practice and can be observed in various disciplines including fraud detection, anomaly detection, medical diagnosis, oil spillage detection, facial recognition, etc.

3.2 Logistic Regression

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable, although many more complex extensions exist. In regression analysis, logistic regression (or logit regression) is estimating the parameters of a logistic model (a form of binary regression). Mathematically, a binary logistic model has a dependent variable with two possible values, such as pass/fail which is represented by an indicator variable, where the two values are labeled "0" and "1". In the logistic model, the log-odds (the logarithm of the odds) for the value labeled "1" is a linear combination of one or more independent variables ("predictors"); the independent variables can each be a binary variable (two classes, coded by an indicator variable) or a continuous variable (any real value). The corresponding probability of the value labeled "1" can vary between 0 (certainly the value "0") and 1 (certainly the value "1"), hence the labeling; the function that converts log-odds to probability is the logistic function, hence the name. The unit of measurement for the log-odds scale is called a logit, from logistic unit, hence the alternative names. Analogous models with a different sigmoid function instead of the logistic function can also be used, such as the probit model; the defining characteristic of the logistic model is that increasing one of the independent variables multiplicatively scales the odds of the given outcome at a constant rate, with each independent variable having its own parameter; for a binary dependent variable this generalizes the odds ratio.

3.2.1 Logistic Regression (Model 1)

Model object:
Code:

```
logit_model =LogisticRegression(class_weight='balanced' , random_state=5 )
```

Parameter used :

Class_weight: “balanced” (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)))

Table 2 : Table showing various metrics of model 1

Logistic regression	
Metric	Model 1 %
'precision'	28%
'accuracy'	78%
'recall'	78%
'specificity'	78%
'fpr'	28%
'fnr'	21%
'f1'	41%
'Roc AUC'	86%
'Precision recall AUC'	50%

Figure 22 : fig showing auc roc curve of Logistic regression (model 1)

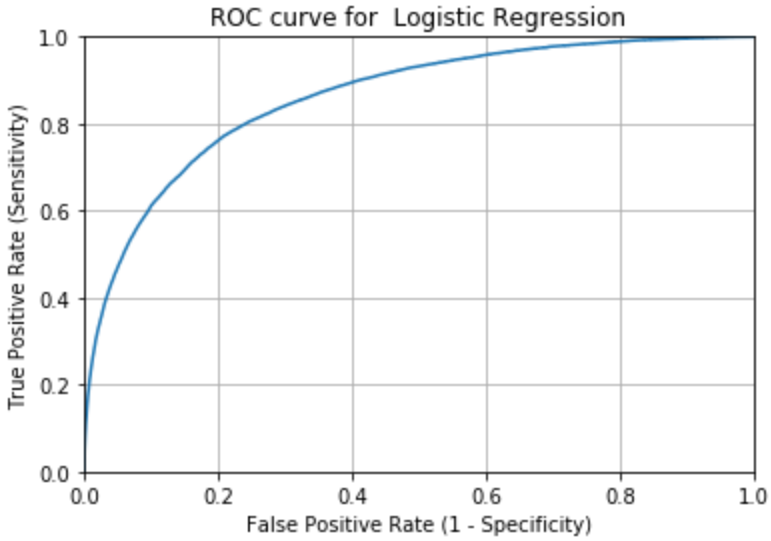
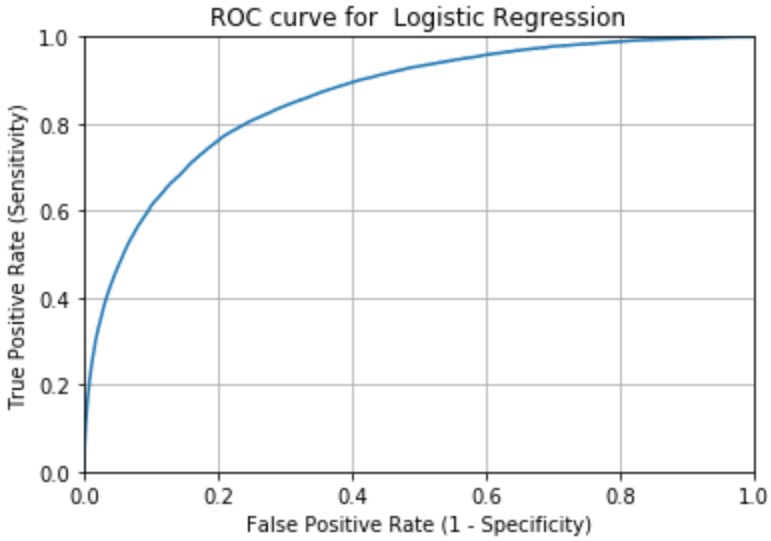


Figure 23: fig showing precision recall auc curve of logistic regression (model 1)



Observation :

Model is 78% accurate which is not that great , precision is only 28% which is low ,Auc roc is 86 % which is good ,fnr is of 21% ,overall model is not that great let try to hyper tune model and if there is any improvement.

3.2.2. Hyper Parameter tuning of Logistic regression

1. We will focus on reducing false negative rate as we dont want to lose customers that have higher chance of making transaction
2. We will also try to maximise roc_auc
3. Precision recall auc is also important metric incase of class imbalance problem which we will try to maximise
4. We try to reduce false positive rate as we dont want to predict user that are not going to make transaction as the one to make transaction
5. We are more worried to predict positive class properly

Model 2

Optimising parameter :

1. penalty= [L1, L2]

L1-norm is also known as least absolute deviations (LAD), least absolute errors (LAE). It is basically minimizing the sum of the absolute differences (S) between the target value (Y_i) and the estimated values ($f(x_i)$):

$$S = \sum_{i=1}^{n} |y_i - f(x_i)|.$$

L2-norm is also known as least squares. It is basically minimizing the sum of the square of the differences (S) between the target value (Y_i) and the estimated values ($f(x_i)$):

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

2. C :Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

3. max_iter :number of iteration algorithms will run

Code :

param_dist = dict(penalty=['l1','l2'] ,C=np.logspace(-4, 4, 20),class_weight=['balanced'] ,max_iter=[5,10,15,20])

Best parmeter and best score (Roc_auc) as given by Random grid search

1. Best Roc_Auc score:85%

2. Best param :{'penalty': 'l2', 'max_iter': 5, 'class_weight': 'balanced', 'C': 0.0001}

Table 3: Table showing various metrics of model 2

Logistic Regression		
Metric	Model 1	Model 2
'precision'	28%	22%
'accuracy'	78%	74%
'recall'	78%	81%
'specificity'	78%	74%
'fpr'	28%	25%
'fnr'	21%	18%
'f1'	41%	39%
'Roc AUC'	86%	86%
'Precision recall AUC'	50%	50%

Figure 24: fig showing auc roc curve of logistic regression (model 2)

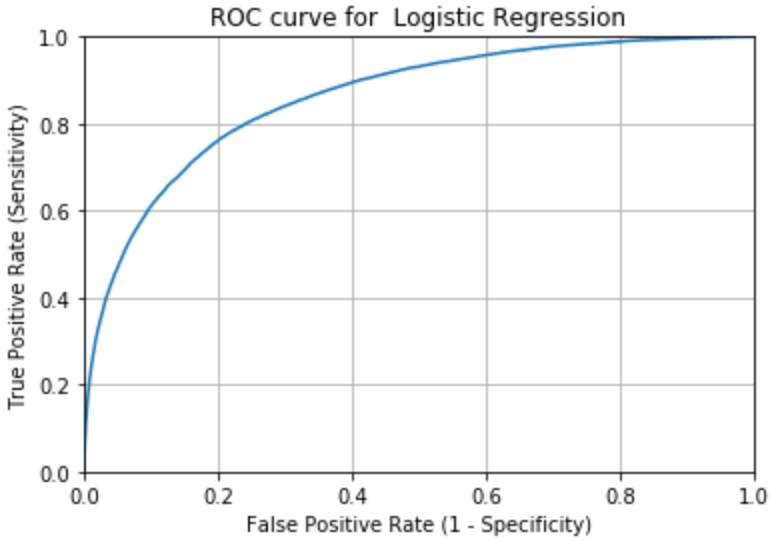
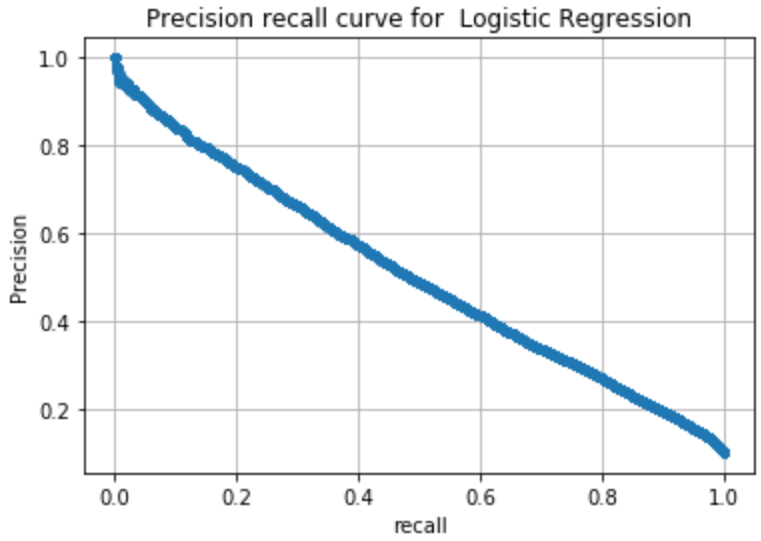


Figure 25: fig showing precision recall auc curve of logistic regression (model 2)



Observation

1. Fnr have decreased 28% to 22%
2. Precision recall AUC: is same 50%
3. Fpr have decreased from 28% to 25%
4. Roc_auc score is same 86%
5. Fnr have decreased from 21% to 18%
6. Accuracy have decreased from 78% to 74%
7. F1 score have reduced from 41% to 39%
8. Optimed model is reducing fnr but over all f1 score is reduced
9. Previous and current model perform almost identical

3.3 Decision tree

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

3.3.1 Decision tree Model 1

Code
dt_model =DecisionTreeClassifier(max_depth= 5,class_weight='balanced' ,random_state =5)

Parameter used :

1. Max depth : maximum depth that tree is allowed to grow

2. class_weght :(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)))

Table 4: Table showing various metrics of model 1

Decision Tree	
Metric	Model 1
'precision'	16%
'accuracy'	66%
'recall'	54%
'specificity'	68%
'fpr'	31%
'fnr'	45%
'f1'	24%
‘Roc AUC’	64%
‘Precision recall AUC’	19%

Figure 26: fig showing auc roc curve of Decision Tree (model 1)

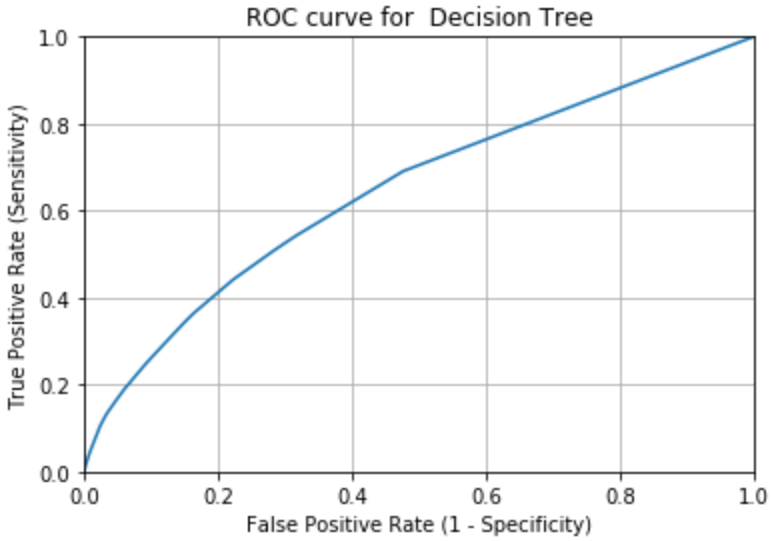
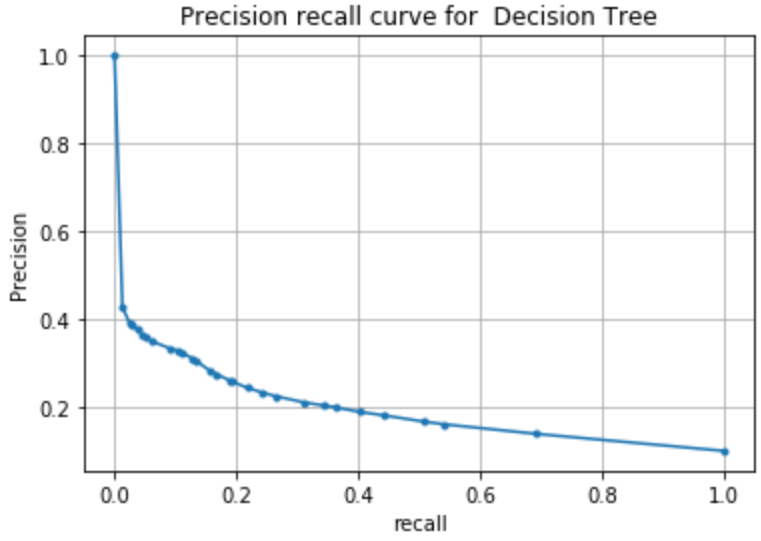


Figure 27: fig showing precision recall auc curve of Decision Tree (model 1)



Observation :

1. Model performance is very poor with Roc_Auc only 64% its worse than logistic regression
2. Fnr is higher than what we had seen in logistic regression

3.3.2. Hyper Parameter tuning of Decision tree (Model 2)

Optimising parameter :

Code:
param_dist =dict(criterion= ['gini', 'entropy'],max_depth= [1, 2, 3, 4, 5],min_samples_split= [2, 3] ,class_weight=['balanced' ,None])

1. criterion: either 'gini' or 'entropy' based algo for decision tree
2. max_depth: max dept the tree will be made to grow
3. min_samples_split :specifies the minimum number of samples required to split an internal node

Best parmeter and best score (Roc_auc) as given by Random grid search

Best Roc_Auc score:63%
Best parameters : {'min_samples_split': 3, 'max_depth': 5, 'criterion': 'entropy', 'class_weight': None}

Table 5: Table showing various metrics of model 2

Metric	Previous %	Current %
'precision'	16%	67%
'accuracy'	66%	90%
'recall'	54%	1%
'specificity'	68%	99%
'fpr'	31%	0.05%
'fnr'	45%	98%
'f1'	24%	2%
'Roc AUC'	64%	64%
'Precision recall AUC'	19%	20%

Figure 28: fig showing auc roc curve of Decision Tree (model 2)

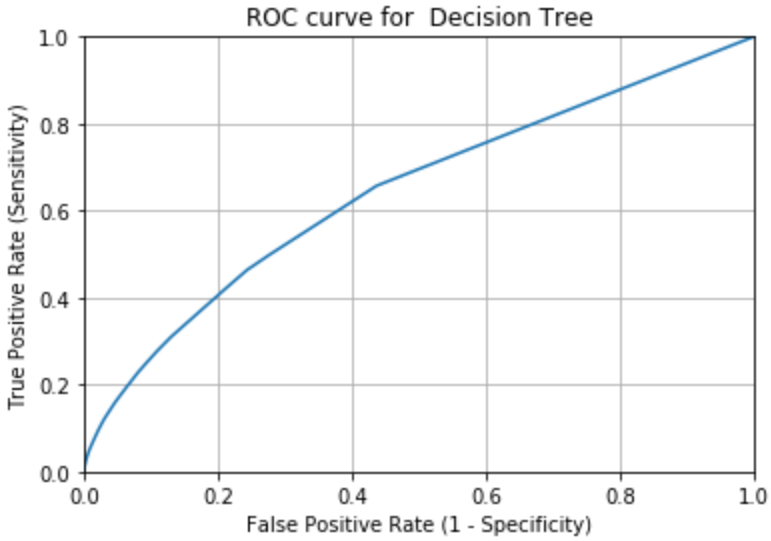
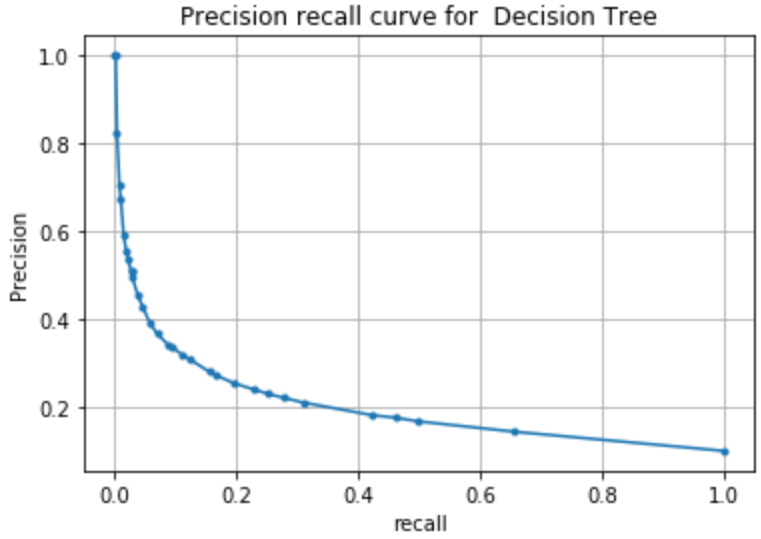


Figure 29: fig showing precision recall auc curve of Decision Tree (model 2)



Observations:

1. Fnr have increased from 45% to 98%
2. Precision recall AUC: increased 1%
3. Fpr have reduced from 31% to 0.05%
4. Roc_auc score almost same 64%
6. Accuracy have increased from 66% to 90%
7. F1 score have reduced from 24% to 2%
8. Optimed model perform worse than base model

3.3.3 Decision Tree model 3

Let's try to make class_wieght balanced and then see if model perform any better (Model 3)

Table 6: Table showing various metrics of model 2

Decision Tree			
Metric	Model1 %	Model 2 %	Model 3 %
'precision'	16%	67%	16%
'accuracy'	66%	90%	67%
'recall'	54%	1%	53%
'specificity'	68%	99%	69%
'fpr'	31%	0.05%	30%
'fnr'	45%	98%	46%
'f1'	24%	2%	24%
'Roc AUC'	64%	64%	64%
'Precision recall AUC'	19%	20%	19%

Model 3 perform better than model 2 as seen in table x above ,we can see that false negative rate has decreased form 98% to 46% ,over all model 1 and model 3 performance are identical .

3.4 Naive bayes

A naive Bayes classifier is an algorithm that uses Bayes' theorem to classify objects. Naive Bayes classifiers assume strong, or naive, independence between attributes of data points. Popular uses of naive Bayes classifiers include spam filters, text analysis and medical diagnosis. These classifiers are widely used for machine learning because they are simple to implement.

3.4.1 Naive bayes Model 1

Model object:
code:
nb_model =GaussianNB( )

Table 7: Table showing various metrics of model 1

Naive Bayes	
Metric	Model 1
'precision'	71%
'accuracy'	92%
'recall'	36%
'specificity'	98%
'fpr'	1%
'fnr'	63%
'f1'	48%
'Roc AUC'	89%
'Precision recall AUC'	58%

Figure 30: fig showing auc roc curve of Naive Bayes (model 1)

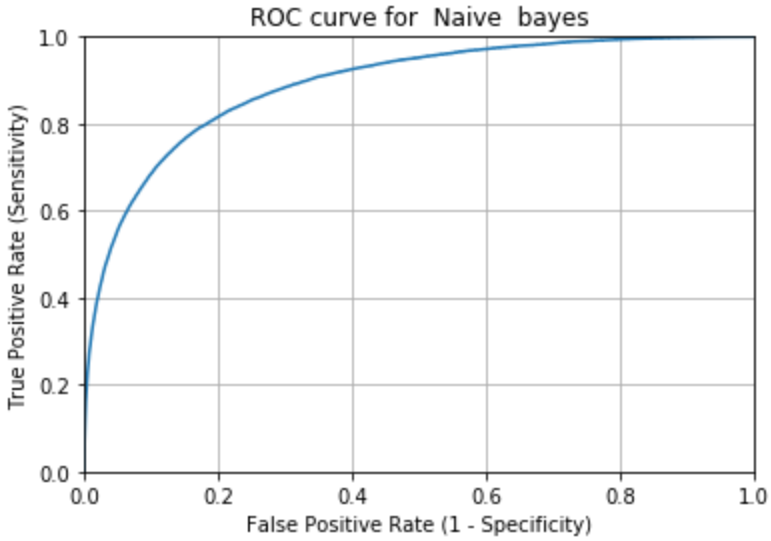
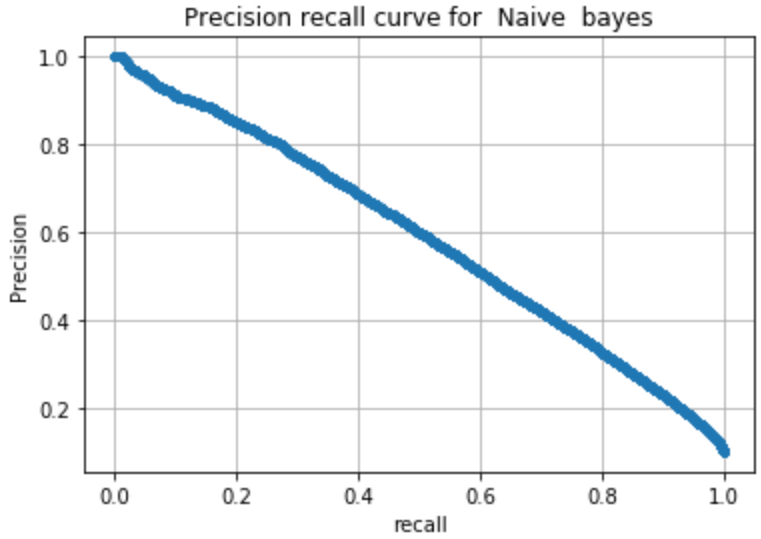


Figure 31: fig showing Precision Recall auc curve of Naive Bayes (model 1)



Model 1 ok but need to reduce fnr as that is more important than accuracy of 92%

3.4.2 Hyper Parameter tuning of Naive Bayes (Model 2)

Optimising parameter :
Var_smoothing :Portion of the largest variance of all the features that is added to variances for calculation stability
Code:
param_dist = dict(var_smoothing =np.logspace(0,-9, num=20))

Best Roc_auc :89%
Best parameter : {'var_smoothing': 7.847599703514623e-08}

Table 8: Table showing various metrics of model 2

Naive Bayes		
Metric	Model 1	Model 2
'precision'	71%	71%
'accuracy'	92%	92%
'recall'	36%	36%
'specificity'	98%	98%
'fpr'	1%	1%
'fnr'	63%	63%
'f1'	48%	48%

'Roc AUC'	89%	89%
'Precision recall AUC'	58%	58%

Observation
1. Fnr same 63%
2. Precision recall AUC: same 89%
3. Fpr same 1 percent
4. Roc_auc score almost same 58%
5. Accuracy same 92%
6. F1 score same 48%
7. Optimed model perform same as base model
8. Still this model is better than decision tree but not better than Logistic Regression as f1 score of logistic is higher than naive bayes and also fnr is lower with same roc_auc score ,only thing is that logistic regression has poorer accuracy.

3.5 Xgboost

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

3.5.1 XgBoost (Model 1)

Code:

XGBClassifier(max_depth=1,eta =1)

Model parameter :
1.max_depth :The maximum depth of a tree, same as GBM.
2.eta :Analogous to learning rate in GBM

Table 9: Table showing various metrics of model 1

XgBoost	
Metric	Model 1
'precision'	100%
'accuracy'	89%
'recall'	0.009%

'specificity'	100%
'fpr'	0%
'fnr'	99%
'f1'	001%
'Roc AUC'	78%
'Precision recall AUC'	36%

Figure 32: fig showing auc roc curve of Xgboost (model 1)

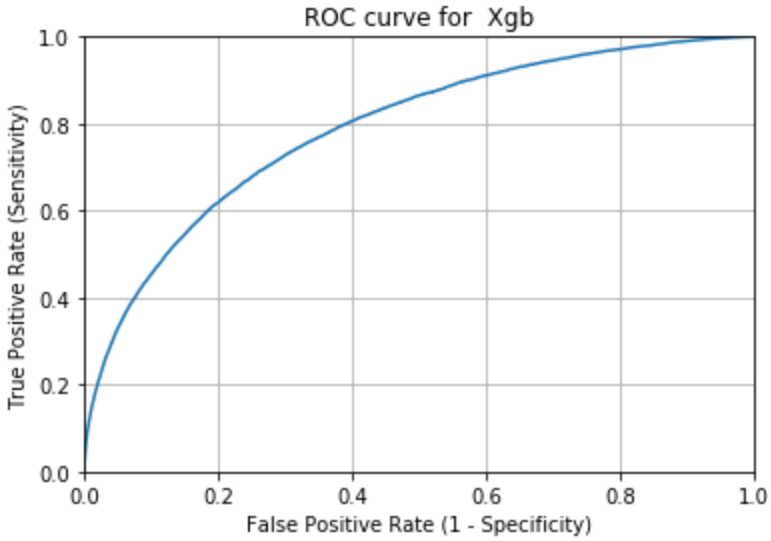
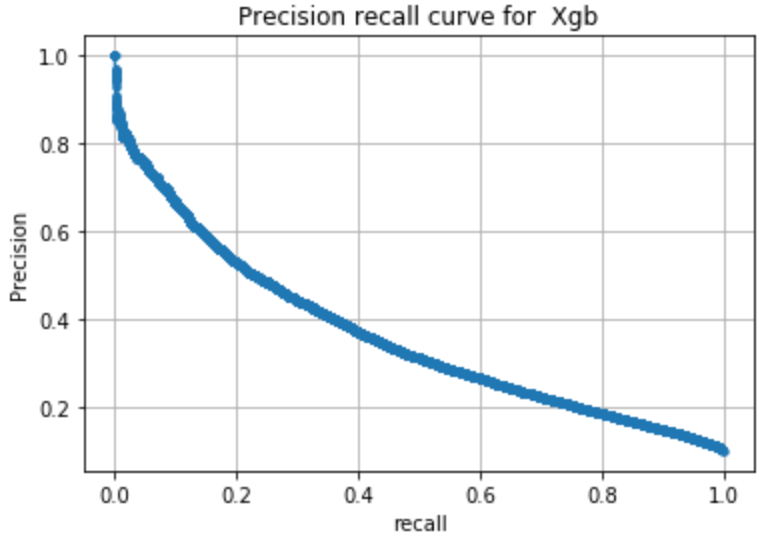


Figure 33: fig showing Precision Recall auc curve of Xgboost (model 1)



Observation :
1. precision 100%
2. accuracy 90%
3. recall less than 1%
4.fpr 0%
5.fnr 99%
6.f1 score less than 1 %
7.Precision recall AUC: 36%
8.AUC: 79%
9. false negative rate is very high ,very very poor model

3.5.2 Hyper Parameter tuning of XG boost (Model 2)

Parameters:

1. Learning_rate:Makes the model more robust by shrinking the weights on each step
2. N_estimators : Number of tree to create
3. Max_depth :Max depth tree is allowed to grow
4. Min_child_weight :Defines the minimum sum of weights of all observations required in a child.
5. Gamma :A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.

6. Subsample:Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.

7. Colsample_bytree:Similar to max_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.

8. Objective :This defines the loss function to be minimized.
9. Scale_pos_weight :A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

Code:

```
xgb=XGBClassifier(learning_rate=0.1,n_estimators=500,max_depth=5,min_child_weight=1,gamma=0,subsample=0.8,colsample_bytree=0.8,objective='binary:logistic',nthread=4,seed=27,scale_pos_weight=2)
```

Table 10: Table showing various metrics of model 2

XGBoost		
Metric	Model 1	Model 2

'precision'	100%	89%
'accuracy'	89%	95%
'recall'	0.009%	67%
'specificity'	100%	99%
'fpr'	0%	0.009%
'fnr'	99%	32%
'f1'	001%	77%
'Roc AUC'	78%	97.5%
'Precision recall AUC'	36%	87.4%

Figure 34 : fig showing auc roc curve of Xgboost (model 1)

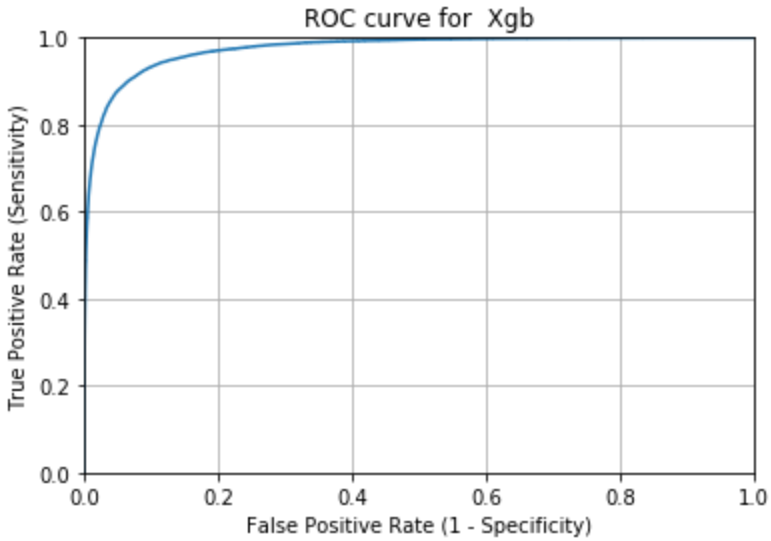
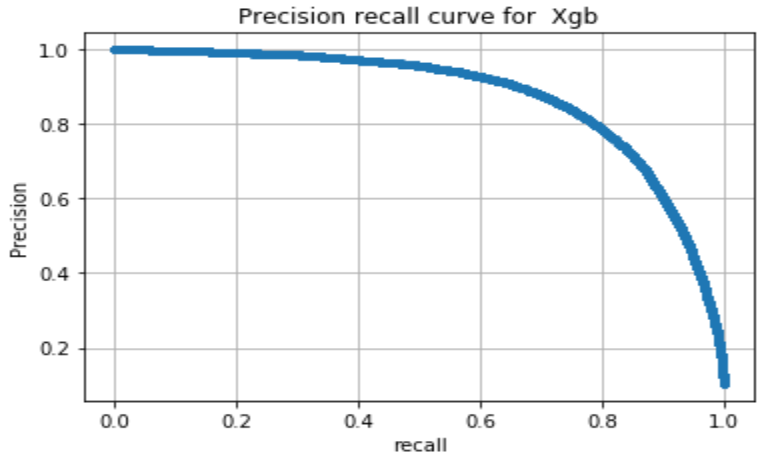


Figure 35: fig showing auc roc curve of Xgboost (model 1)



Chapter 4

Conclusion

4.1 Model Evaluation

Now that we have a few models for predicting the target variable, 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 bike rental Data, the latter two, *Interpretability* and *Computation Efficiency,* do not hold much significance. Therefore we will use *Predictive performance* as the criteria to compare and evaluate models. Predictive performance can be measured by comparing the Predictions of the models with real values of the target variables, and calculating some average error measure.

4.2 Confusion Matrix:

A confusion matrix is a summary of prediction results on a classification problem.
The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix.
The confusion matrix shows the ways in which your classification model is confused when it makes predictions.

It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made.

Figure 36 : fig showing confusion matrix

	Class 1 Predicted	*Class 2 Predicted*
Class 1 Actual	**TP**	**FN**
Class 2 Actual	**FP**	**TN**

Here,
• Class 1 : Positive
• Class 2 : Negative

Definition of Terms:

• Positive (P) : Observation is positive (for example: is an apple).
• Negative (N) : Observation is not positive (for example: is not an apple).
• True Positive (TP) : Observation is positive, and is predicted to be positive.
• False Negative (FN) : Observation is positive, but is predicted to be negative.
• True Negative (TN) : Observation is negative, and is predicted to be negative.
• False Positive (FP) : Observation is negative, but is predicted to be positive.

4.2.1. Accuracy :

$$\textbf{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

However, there are problems with accuracy. It assumes equal costs for both kinds of errors. A 99% accuracy can be excellent, good, mediocre, poor or terrible depending upon the problem.

4.2.2.Recall:

Recall can be defined as the ratio of the total number of correctly classified positive examples divide to the total number of positive examples. High Recall indicates the class is correctly recognized (small number of FN).

Recall is given by the relation:

$$\textbf{\textit{Recall}} = \frac{TP}{TP + FN}$$

4.2.3.Precision:

To get the value of precision we divide the total number of correctly classified positive examples by the total number of predicted positive examples. High Precision indicates an example labeled as positive is indeed positive (small number of FP).

Precision is given by the relation:

$$Precision = \frac{TP}{TP + FP}$$

High recall, low precision:This means that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives.

Low recall, high precision:This shows that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP)

4.2.4. F-measure:

Since we have two measures (Precision and Recall) it helps to have a measurement that represents both of them. We calculate an F-measure which uses Harmonic Mean in place of Arithmetic Mean as it punishes the extreme values more.
The F-Measure will always be nearer to the smaller value of Precision or Recall.

$$F\text{ - }measure = \frac{2*Recall*Precision}{Recall + Precision}$$

4.2.5. AUC - ROC

AUC -ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents degree or measure of separability. It tells how much model is capable of distinguishing between classes.

4.2.6 Model Selection

Table 11: table showing various machine learning model and its classification metric

	precision	accuracy	recall	specificity	fpr	fnr	f1
xgb_opti	0.940694	0.977905	0.832620	0.994136	0.005864	0.167380	0.883364
logit_opti	0.259775	0.748370	0.813215	0.741126	0.258874	0.186785	0.393765
logit_base	0.284939	0.781095	0.780625	0.781148	0.218852	0.219375	0.417488
dt_base	0.160813	0.669755	0.541994	0.684028	0.315972	0.458006	0.248033
nb_base	0.714825	0.921695	0.367300	0.983630	0.016370	0.632700	0.485259

nb_opti	0.714825	0.921695	0.367300	0.983630	0.016370	0.632700	0.485259
dt_opti	0.672078	0.900040	0.010300	0.999439	0.000561	0.989700	0.020288
xgb_base	1.000000	0.899520	0.000100	1.000000	0.000000	0.999900	0.000199

1. Choosing model that has high f1 score
2. low false negative rate
3. High roc_auc
4. low false positive rate
5. All this criteria is fulfilled by xg boost
6. so final model is xgboost

Appendix B - Python Code

1. Distribution of continuous variables (Train data set) (fig)

```
def check_distribution(X ,start ,end):
    fig = plt.figure(figsize=(15,15))
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    names =list(range(start,end))
    if((end>start )and (end-start <=20) and (end>=1 and start<200 )):
        for i , realindex in  enumerate(names,1):
            ax = fig.add_subplot( 5,4, i)
            sns.distplot(X.iloc[:,realindex] ,ax=ax)
        plt.suptitle("Distribution plot ")
        plt.show()

    else:
        print("Check index start index should be less end index")
        print("Max 20 plot can be printed ")
        print("Start and end index range is (0-200)")


# distribution of train data
check_distribution(X,25,45)
check_distribution(X,1,20)
check_distribution(X,71,90)
check_distribution(X,103,122)
check_distribution(X,150,170)
check_distribution(X,185,199)
```

2. Distribution of categorical variables (fig)

```
def plot_target_class(var):

    plt.figure(figsize=(15,15))
    plt.subplot(221)
    plt.title("Count of each target class")
    sns.countplot(x = var)

    plt.subplot(222)
    plt.pie(var.value_counts() , labels=[0,1] ,autopct='%1.2f%%' ,explode=(0,1)
,shadow=True)
    plt.legend(['class 0','class 1'])
    plt.title("Percentage of class in taget var")
    plt.show()
```

3.Distribution of continuous variables (test data set) (fig)

```
def check_distribution(X ,start ,end):
    fig = plt.figure(figsize=(15,15))
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    names =list(range(start,end))
    if((end>start )and (end-start <=20) and (end>=1 and start<200 )):
        for i , realindex in  enumerate(names,1):
            ax = fig.add_subplot( 5,4, i)
            sns.distplot(X.iloc[:,realindex] ,ax=ax)
        plt.suptitle("Distribution plot ")
        plt.show()

    else:
        print("Check index start index should be less end index")
        print("Max 20 plot can be printed ")
        print("Start and end index range is (0-200)")

#distribution of test data
check_distribution(test,25,45)
check_distribution(test,1,20)
check_distribution(test,71,90)
check_distribution(test,103,122)
check_distribution(test,150,170)
check_distribution(test,185,199)
```

4. Box plot (fig)

```
def plot_boxplot(df,start,end):
  selected_var =df.columns.values
  fig = plt.figure(figsize=(15,15))
  fig.subplots_adjust(hspace=0.4, wspace=0.4)
  if(start >=0 and end<len(selected_var)):
    for i , realindex in  enumerate(selected_var[start:end],1):
      ax = fig.add_subplot( 5,4, i)
      sns.boxplot(y=X.loc[:,realindex],x=Y ,ax=ax)
      plt.suptitle("box plot ")

  plt.show()

#Train data set
# plot box plot for 20 variables
```

```
# X is independent varaiable in train
plot_boxplot(X ,0,20)
plot_boxplot(X ,21,40)
plot_boxplot(X ,60,79)
plot_boxplot(X ,100,119)
plot_boxplot(X ,150,169)
plot_boxplot(X ,180,195)
```

5. Vif plot (fig)

```
# This function Plot barplot with respect to varible name and its correponding vif score

def plot_vif(df):
    plt.figure(figsize=(14,26))
    sns.barplot(y='var',x ='vif' ,data =vif_df )
    plt.title(" Vif score of training data ")
    plt.tight_layout()
    plt.show()



#vif plot for training data
plot_vif(vif_df)
```

6. Roc -Auc plot

```
def roc_plot(Y_true ,Y_prob ,model="model") :
  fpr, tpr, thresholds =roc_curve(Y_true, Y_prob)
  plt.plot(fpr, tpr)
  plt.xlim([0.0, 1.0])
  plt.ylim([0.0, 1.0])
  plt.title('ROC curve for  %s'%model)
  plt.xlabel('False Positive Rate (1 - Specificity)')
  plt.ylabel('True Positive Rate (Sensitivity)')
  plt.grid(True)
  plt.show()
  print('  AUC: %.3f' % auc(fpr,tpr))
```

7. Precision recall plot

```
def precision_recall_plot(Y_true ,Y_prob ,model="model"):
  precision, recall, thresholds = precision_recall_curve(Y_true, Y_prob)
  auc_score = auc(recall, precision)
  plt.plot(recall, precision, marker='.')

  plt.title('Precision recall curve for  %s'%model)
  plt.xlabel('recall')
  plt.ylabel('Precision')
  plt.grid(True)
  plt.show()
  print(' Precision recall AUC: %.3f' % auc_score)
```

7. Complete Python File

```
import os

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.metrics import roc_curve

from sklearn.metrics import roc_auc_score

from sklearn.naive_bayes import GaussianNB

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestClassifier

from sklearn.feature_selection import SelectFromModel
```

```
from sklearn.model_selection   import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.linear_model import LogisticRegression

from sklearn.metrics  import confusion_matrix

from sklearn.model_selection import cross_val_score,cross_validate

from sklearn.model_selection import GridSearchCV

from sklearn.metrics import roc_curve,auc

from sklearn.metrics import roc_auc_score,precision_recall_curve

from sklearn.model_selection import RandomizedSearchCV


import os

%matplotlib inline


#Loading test train data

train = pd.read_csv("train.csv")

test =pd.read_csv("test.csv")


#checking head of train data

train.head()


#checking head of test data

test.head()


print("Train info")

print(train.info() ,end='\n\n')

print("Train Dim  :",train.shape ,end='\n\n')
```

```
print("Test info ",end='\n')

print(test.info(),end='\n\n')

print("Test Dim  :",test.shape)


#data types in train dataset

train.dtypes.value_counts()


#data types in test dataset

test.dtypes.value_counts()


#check Descriptive stats of train data

train.describe()

#mean and median of most of the data is same


#storing ID_code  of test tarin data

train_ID_code_orignal = train.ID_code

test_Id_code_orignal  = test.ID_code


#droping ID_code in train

train.drop(columns="ID_code" ,inplace =True )

print(train.shape)


#droping ID_code in test

test.drop(columns="ID_code" ,inplace =True )

print(test.shape)


#This function takes series as input and plot bar and pie chart
```

```
def plot_target_class(var):


    plt.figure(figsize=(15,15))

    plt.subplot(221)

    plt.title("Count of each target class")

    sns.countplot(x = var)


    plt.subplot(222)

    plt.pie(var.value_counts() , labels=[0,1] ,autopct='%1.2f%%' ,explode=(0,1)
,shadow=True)

    plt.legend(['class 0','class 1'])

    plt.title("Percentage of class in taget var")

    plt.show()


#ploting count plot of target varaible

plot_target_class(train.target)


def calculate_missing_values(df):

    """ this function takes dataframe as input and calculate missing value count in each
variable

        return a dataframe with missing value count per column in decending order

    """

    missing_value_df =pd.DataFrame(df.isnull().sum())

    missing_value_df.rename(columns={0 :'count'} ,inplace=True)

    missing_value_df.sort_values(by='count',ascending=False)

    return missing_value_df
```

```
# missing value for train and test

print("Train dataframe",calculate_missing_values(train).head(10),sep='\n')

print("Test dataframe",calculate_missing_values(test).head(10),sep='\n')

#removing target variable from train dataset as it is not needed for vif calculation

# creating target and independent variable from train dataset

independent_var= [i for i in train.columns.values if i!='target']

X = train[independent_var]

Y =train.target



def calulate_vif(df,column_name ):

    # This function take dataframe and columns name and then calculate vif score

    #taking one var as taget and rest as independent and then running OLS

    #to calculate r^2 score and then vif using formula =1/(1-r^2)

    #return a dict {colname:vif}


    vif_score ={}

    for i in range(0 , df.shape[1]):

        print(i,end="\n")

        Y =df.loc[: , df.columns==column_name[i]]

        X =df.drop(columns=column_name[i])

        model =LinearRegression().fit(X,Y)

        rsq=model.score(X,Y)
```

```
        vif =round(((1)/(1-rsq)) ,2)
        print("vif score:",vif,end="\n")
        vif_score[independent_var[i]]=vif

    return vif_score

# calculating vif
vif_dict=calulate_vif(X ,independent_var)

# writing values of vif in dataframe
vif_df=pd.DataFrame(list(vif_dict.values()) ,index=list(vif_dict.keys()) )
vif_df.reset_index(inplace=True)
vif_df.rename(columns={0:"vif" ,'index':"var"} ,inplace=True)

# This function Plot barplot with respect to varible name and its correponding vif score

def plot_vif(df):
    plt.figure(figsize=(14,26))
    sns.barplot(y='var',x ='vif' ,data =vif_df )
    plt.title(" Vif score of training data ")
    plt.tight_layout()
    plt.show()


#vif plot for training data
plot_vif(vif_df)
```

```
#This function pair plots train by subseting dataframe based on start and end index passed

#X =independent Var

#Y =dependent Var

#start =start index

#end =end index


def check_distribution(X ,start ,end):

    fig = plt.figure(figsize=(15,15))

    fig.subplots_adjust(hspace=0.4, wspace=0.4)

    names =list(range(start,end))

    if((end>start )and (end-start <=20) and (end>=1 and start<200 )):

        for i , realindex in  enumerate(names,1):

            ax = fig.add_subplot( 5,4, i)

            sns.distplot(X.iloc[:,realindex] ,ax=ax)

        plt.suptitle("Distribution plot ")

        plt.show()


    else:

        print("Check index start index should be less end index")

        print("Max 20 plot can be printed ")

        print("Start and end index range is (0-200)")
```

```
# distribution of train data

check_distribution(X,25,45)

check_distribution(X,1,20)

check_distribution(X,71,90)

check_distribution(X,103,122)

check_distribution(X,150,170)

check_distribution(X,185,199)

#distribution of test data

check_distribution(test,25,45)

check_distribution(test,1,20)

check_distribution(test,71,90)

check_distribution(test,103,122)

check_distribution(test,150,170)

check_distribution(test,185,199)

# box plot of first 20 variable

#this function plot boxplot

# start :start index

# end end index

# df data frame

def plot_boxplot(df,start,end):

  selected_var =df.columns.values

  fig = plt.figure(figsize=(15,15))
```

```
 fig.subplots_adjust(hspace=0.4, wspace=0.4)
 if(start >=0 and end<len(selected_var)):
   for i , realindex in  enumerate(selected_var[start:end],1):
     ax = fig.add_subplot( 5,4, i)
     sns.boxplot(y=X.loc[:,realindex],x=Y ,ax=ax)
     plt.suptitle("box plot ")

 plt.show()

#Train data set
# plot box plot for 20 variables
# X is independent varaiable in train
plot_boxplot(X ,0,20)
plot_boxplot(X ,21,40)
plot_boxplot(X ,60,79)
plot_boxplot(X ,100,119)
plot_boxplot(X ,150,169)
plot_boxplot(X ,180,195)

#test data set
# plot box plot for 20 variables

plot_boxplot(test ,0,20)
plot_boxplot(test ,21,40)
```

```
plot_boxplot(test ,60,79)

plot_boxplot(test ,100,119)

plot_boxplot(test ,150,169)

plot_boxplot(test ,180,195)


# this function fills values below lower fense and upper fense as nan

# data: data frame

# var_list : name of var to fill outlier with na

def remove_outlier_fillna(data ,var_list):

    for i in var_list:

        q75 ,q25 = np.percentile(data.loc[:,i] ,[75,25])

        iqr =q75-q25

        minimum = q25 -(1.5*iqr)

        maximum =q75+(1.5*iqr)

        print(i,"iqr",iqr,"mimimum",minimum ,"maximum",maximum,sep="  :")

        data.loc[data[i]<minimum ,i]=np.nan

        data.loc[data[i]>maximum ,i]=np.nan


    return data



#remove outlier for train data X is dataframe that contain train dataset independent var


# select columns name from dataframe
```

```
selected_var_train =X.columns.values

#remove outlier and fill na

X =remove_outlier_fillna(X.copy() ,selected_var_train)

# select columns name from dataframe

selected_var_test =X.columns.values

#remove outlier and fill na

test=remove_outlier_fillna(test.copy() ,selected_var_test)

# Before imputation

# total missing in train

print(X.isnull().sum().sum())

print('#'*30)

#total missing in test

print(test.isnull().sum().sum())

#percentage of missing value in train

print(X.isnull().mean()*100)

print('#'*30)

#percentage of missing value in test

print(test.isnull().mean()*100)

#this function imputes missingvalue with mean of that column

#data = dataframe

#column_name = name of  columns of data frame
```

```
def impute_na_with_mean(data, column_name):
    for i in column_name:
        data.loc[:,i]=data.loc[:,i].fillna(data.loc[:,i].mean())
    return data

# select columns name from dataframe
selected_var_train =X.columns.values
#impute mean
X =impute_na_with_mean(X.copy() ,selected_var_train)
# select columns name from dataframe
selected_var_test =X.columns.values
#impute mean
test=impute_na_with_mean(test.copy() ,selected_var_test)

# after Imputation
# total missing in train
print(X.isnull().sum().sum())

print('#'*30)
#total missing in test
print(test.isnull().sum().sum())

#percentage of missing value in train
print(X.isnull().mean()*100)
print('#'*30)
#percentage of missing value in test
```

```
print(test.isnull().mean()*100)

#standardising data

# df data passed to standardise

def std_data (df  ):

    scaler =StandardScaler()

    return pd.DataFrame(scaler.fit_transform(df) ,columns=df.columns)

# standardising  test train dataset

X= std_data(X)

test=std_data(test)



# ploting roc curve

#This function takes target variable and its predicted probality to plot roc curve

def roc_plot(Y_true ,Y_prob ,model="model") :

  fpr, tpr, thresholds =roc_curve(Y_true, Y_prob)

  plt.plot(fpr, tpr)

  plt.xlim([0.0, 1.0])

  plt.ylim([0.0, 1.0])

  plt.title('ROC curve for  %s'%model)

  plt.xlabel('False Positive Rate (1 - Specificity)')

  plt.ylabel('True Positive Rate (Sensitivity)')

  plt.grid(True)
```

```
  plt.show()

  print('  AUC: %.3f' % auc(fpr,tpr))

# Return model accuracy param

#  This function calculate model performance criteria

#     from confusion matrix    a

#    return a dictinary of model performance criteria

def model_accuracy(conf_matrix):


    model_pram={}

    tn =conf_matrix.iloc[0,0]

    tp =conf_matrix.iloc[1,1]

    fp =conf_matrix.iloc[0,1]

    fn =conf_matrix.iloc[1,0]

    model_pram['precision'] =(tp)/(tp+fp)

    model_pram['accuracy']  =(tp+tn)/(tp+tn+fp+fn)

    model_pram['recall']  =(tp)/(tp+fn)

    model_pram['specificity'] =(tn)/(tn +fp)

    model_pram['fpr'] =(fp)/(fp+tn)

    model_pram['fnr'] =(fn)/(fn+tp)

    model_pram['f1'] =2*(( model_pram['precision'] *model_pram['recall'])/(
model_pram['precision'] +model_pram['recall']))

    return model_pram
```

```
# ploting roc curve

#This function takes target variable and its predicted probality to plot Precision Recall curve

def precision_recall_plot(Y_true ,Y_prob ,model="model"):

  precision, recall, thresholds = precision_recall_curve(Y_true, Y_prob)

  auc_score = auc(recall, precision)

  plt.plot(recall, precision, marker='.')


  plt.title('Precision recall curve for  %s'%model)

  plt.xlabel('recall')

  plt.ylabel('Precision')

  plt.grid(True)

  plt.show()

  print(' Precision recall AUC: %.3f' % auc_score)



#This function runs model  and return confusion matrix

# plot roc_auc plot

# plot precision recall plot

# model_object = object of fitted model

# X = Independent var

# Y =target variable

# model name  = name of model

def run_model(model_object , X,Y ,model_name):
```

```
    # store the predicted probabilities for class 1

    pred_prob = model_object.predict_proba(X)[:, 1]


    # store the class prediction of model

    y_pred=model_object.predict(X)


    # create confusion matrix

    conf =pd.crosstab(Y,y_pred)


    # Print model accuracy parameter

    print(model_accuracy(conf))


    #calling roc_plot method

    roc_plot(Y ,pred_prob ,model_name)


    #Precision recall curve

    precision
train.dtypes.value_counts()

#data types in test dataset
test.dtypes.value_counts()

#check Descriptive stats of train data
train.describe()
#mean and median of most of the data is same

#storing ID_code  of test tarin data
train_ID_code_orignal = train.ID_code
test_Id_code_orignal  = test.ID_code

#droping ID_code in train
train.drop(columns="ID_code" ,inplace =True )
```

```
print(train.shape)

#droping ID_code in test
test.drop(columns="ID_code" ,inplace =True )
print(test.shape)

#This function takes series as input and plot bar and pie chart

def plot_target_class(var):

    plt.figure(figsize=(15,15))
    plt.subplot(221)
    plt.title("Count of each target class")
    sns.countplot(x = var)

    plt.subplot(222)
    plt.pie(var.value_counts() , labels=[0,1] ,autopct='%1.2f%%' ,explode=(0,1)
,shadow=True)
    plt.legend(['class 0','class 1'])
    plt.title("Percentage of class in taget var")
    plt.show()

#ploting count plot of target varaible
plot_target_class(train.target)

def calculate_missing_values(df):
    """ this function takes dataframe as input and calculate missing value count in each
variable
        return a dataframe with missing value count per column in decending order
    """
    missing_value_df =pd.DataFrame(df.isnull().sum())
    missing_value_df.rename(columns={0 :'count'} ,inplace=True)
    missing_value_df.sort_values(by='count',ascending=False)
    return missing_value_df

# missing value for train and test
print("Train dataframe",calculate_missing_values(train).head(10),sep='\n')
print("Test dataframe",calculate_missing_values(test).head(10),sep='\n')

#removing target variable from train dataset as it is not needed for vif calculation
# creating target and independent variable from train dataset
independent_var= [i for i in train.columns.values if i!='target']
X = train[independent_var]
Y =train.target
```

```
def calulate_vif(df,column_name ):
    # This function take dataframe and columns name and then calculate vif score
    #taking one var as taget and rest as independent and then running OLS
    #to calculate r^2 score and then vif using formula =1/(1-r^2)
    #return a dict {colname:vif}

    vif_score ={}
    for i in range(0 , df.shape[1]):
        print(i,end="\n")
        Y =df.loc[: , df.columns==column_name[i]]
        X =df.drop(columns=column_name[i])
        model =LinearRegression().fit(X,Y)
        rsq=model.score(X,Y)
        vif =round(((1)/(1-rsq)) ,2)
        print("vif score:",vif,end="\n")
        vif_score[independent_var[i]]=vif

    return vif_score

# calculating vif
vif_dict=calulate_vif(X ,independent_var)

# writing values of vif in dataframe
vif_df=pd.DataFrame(list(vif_dict.values()) ,index=list(vif_dict.keys()) )
vif_df.reset_index(inplace=True)
vif_df.rename(columns={0:"vif" ,'index':"var"} ,inplace=True)

# This function Plot barplot with respect to varible name and its correponding vif score

def plot_vif(df):
    plt.figure(figsize=(14,26))
    sns.barplot(y='var',x ='vif' ,data =vif_df )
    plt.title(" Vif score of training data ")
    plt.tight_layout()
    plt.show()


#vif plot for training data
plot_vif(vif_df)


#This function pair plots train by subseting dataframe based on start and end index
passed
#X =independent Var
#Y =dependent Var
#start =start index
#end =end index
```

```
def check_distribution(X ,start ,end):
    fig = plt.figure(figsize=(15,15))

# distribution of train data
check_distribution(X,25,45)
check_distribution(X,1,20)
check_distribution(X,71,90)
check_distribution(X,103,122)
check_distribution(X,150,170)
check_distribution(X,185,199)

#distribution of test data
check_distribution(test,25,45)
check_distribution(test,1,20)
check_distribution(test,71,90)
check_distribution(test,103,122)
check_distribution(test,150,170)
check_distribution(test,185,199)

# box plot of first 20 variable
#this function plot boxplot
# start :start index
# end end index
# df data frame
def plot_boxplot(df,start,end):
  selected_var =df.columns.values    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    names =list(range(start,end))
    if((end>start )and (end-start <=20) and (end>=1 and start<200 )):
        for i , realindex in  enumerate(names,1):
            ax = fig.add_subplot( 5,4, i)
            sns.distplot(X.iloc[:,realindex] ,ax=ax)
        plt.suptitle("Distribution plot ")
        plt.show()

    else:
        print("Check index start index should be less end index")
        print("Max 20 plot can be printed ")
        print("Start and end index range is (0-200)")

  fig = plt.figure(figsize=(15,15))
  fig.subplots_adjust(hspace=0.4, wspace=0.4)
  if(start >=0 and end<len(selected_var)):
    for i , realindex in  enumerate(selected_var[start:end],1):
      ax = fig.add_subplot( 5,4, i)
      sns.boxplot(y=X.loc[:,realindex],x=Y ,ax=ax)
      plt.suptitle("box plot ")
```

```
  plt.show()


#Train data set
# plot box plot for 20 variables
# X is independent varaiable in train
plot_boxplot(X ,0,20)
plot_boxplot(X ,21,40)
plot_boxplot(X ,60,79)
plot_boxplot(X ,100,119)
plot_boxplot(X ,150,169)
plot_boxplot(X ,180,195)


#test data set
# plot box plot for 20 variables

plot_boxplot(test ,0,20)
plot_boxplot(test ,21,40)
plot_boxplot(test ,60,79)
plot_boxplot(test ,100,119)
plot_boxplot(test ,150,169)
plot_boxplot(test ,180,195)


# this function fills values below lower fense and upper fense as nan
# data: data frame
# var_list : name of var to fill outlier with na
def remove_outlier_fillna(data ,var_list):
    for i in var_list:
        q75 ,q25 = np.percentile(data.loc[:,i] ,[75,25])
        iqr =q75-q25
        minimum = q25 -(1.5*iqr)
        maximum =q75+(1.5*iqr)
        print(i,"iqr",iqr,"mimimum",minimum ,"maximum",maximum,sep="  :")
        data.loc[data[i]<minimum ,i]=np.nan
        data.loc[data[i]>maximum ,i]=np.nan

    return data


#remove outlier for train data X is dataframe that contain train dataset independent var

# select columns name from dataframe
selected_var_train =X.columns.values
```

```
#remove outlier and fill na
X =remove_outlier_fillna(X.copy() ,selected_var_train)
# select columns name from dataframe
selected_var_test =X.columns.values
#remove outlier and fill na
test=remove_outlier_fillna(test.copy() ,selected_var_test)

# Before imputation
# total missing in train
print(X.isnull().sum().sum())

print('#'*30)
#total missing in test
print(test.isnull().sum().sum())

#percentage of missing value in train
print(X.isnull().mean()*100)
print('#'*30)
#percentage of missing value in test
print(test.isnull().mean()*100)

#this function imputes missingvalue with mean of that column
#data = dataframe
#column_name = name of  columns of data frame
def impute_na_with_mean(data, column_name):
    for i in column_name:
        data.loc[:,i]=data.loc[:,i].fillna(data.loc[:,i].mean())
    return data

# select columns name from dataframe
selected_var_train =X.columns.values
#impute mean
X =impute_na_with_mean(X.copy() ,selected_var_train)
# select columns name from dataframe
selected_var_test =X.columns.values
#impute mean
test=impute_na_with_mean(test.copy() ,selected_var_test)

# after Imputation
# total missing in train
print(X.isnull().sum().sum())

print('#'*30)
#total missing in test
print(test.isnull().sum().sum())

#percentage of missing value in train
```

```
print(X.isnull().mean()*100)
print('#'*30)
#percentage of missing value in test
print(test.isnull().mean()*100)

#standardising data
# df data passed to standardise
def std_data (df  ):
    scaler =StandardScaler()
    return pd.DataFrame(scaler.fit_transform(df) ,columns=df.columns)

# standardising  test train dataset
X= std_data(X)
test=std_data(test)




# ploting roc curve
#This function takes target variable and its predicted probality to plot roc curve
def roc_plot(Y_true ,Y_prob ,model="model") :
  fpr, tpr, thresholds =roc_curve(Y_true, Y_prob)
  plt.plot(fpr, tpr)
  plt.xlim([0.0, 1.0])
  plt.ylim([0.0, 1.0])
  plt.title('ROC curve for  %s'%model)
  plt.xlabel('False Positive Rate (1 - Specificity)')
  plt.ylabel('True Positive Rate (Sensitivity)')
  plt.grid(True)
  plt.show()
  print('  AUC: %.3f' % auc(fpr,tpr))




# Return model accuracy param
#  This function calculate model performance criteria
#     from confusion matrix    a
#    return a dictinary of model performance criteria
def model_accuracy(conf_matrix):



    model_pram={}
```

```
    tn =conf_matrix.iloc[0,0]
    tp =conf_matrix.iloc[1,1]
    fp =conf_matrix.iloc[0,1]
    fn =conf_matrix.iloc[1,0]
    model_pram['precision'] =(tp)/(tp+fp)
    model_pram['accuracy']  =(tp+tn)/(tp+tn+fp+fn)
    model_pram['recall']  =(tp)/(tp+fn)
    model_pram['specificity'] =(tn)/(tn +fp)
    model_pram['fpr'] =(fp)/(fp+tn)
    model_pram['fnr'] =(fn)/(fn+tp)
    model_pram['f1'] =2*(( model_pram['precision'] *model_pram['recall'])/(
model_pram['precision'] +model_pram['recall']))
    return model_pram



# ploting roc curve
#This function takes target variable and its predicted probality to plot Precision Recall
curve
def precision_recall_plot(Y_true ,Y_prob ,model="model"):
  precision, recall, thresholds = precision_recall_curve(Y_true, Y_prob)
  auc_score = auc(recall, precision)
  plt.plot(recall, precision, marker='.')

  plt.title('Precision recall curve for  %s'%model)
  plt.xlabel('recall')
  plt.ylabel('Precision')
  plt.grid(True)
  plt.show()
  print(' Precision recall AUC: %.3f' % auc_score)


#This function runs model  and return confusion matrix
# plot roc_auc plot
# plot precision recall plot
# model_object = object of fitted model
# X = Independent var
# Y =target variable
# model name  = name of model
def run_model(model_object , X,Y ,model_name):



    # store the predicted probabilities for class 1
    pred_prob = model_object.predict_proba(X)[:, 1]
```

```
    # store the class prediction of model
    y_pred=model_object.predict(X)

    # create confusion matrix
    conf =pd.crosstab(Y,y_pred)

    # Print model accuracy parameter
    print(model_accuracy(conf))

    #calling roc_plot method
    roc_plot(Y ,pred_prob ,model_name)

    #Precision recall curve
    precision_recall_plot(Y ,pred_prob ,model_name)

    return conf



#class_weight ='balanced '  for imbalanced data set
logit_model =LogisticRegression(class_weight='balanced' , random_state=5 )

#fit model
logit_model.fit(X,Y)

confMatrix=run_model(logit_model , X,Y ,"Logistic Regression")

# specify "parameter distributions" rather than a "parameter grid"
#penalty l1 l2 norm
# C regularistion term
# class_weight balanced as it make weight balanced it is usefull incase of imbalanced
class problem ,
# max_iter number of iteraton algorithm will run

param_dist = dict(penalty=['l1','l2'] ,C=np.logspace(-4, 4, 20),class_weight=['balanced']
,max_iter=[5,10,15,20])

#initilising    RandomizedSearchCV

rand = RandomizedSearchCV(logit_model, param_dist, cv=10, scoring='roc_auc',
n_iter=10, random_state=5, return_train_score=False)
rand.fit(X, Y)
pd.DataFrame(rand.cv_results_)[['mean_test_score', 'std_test_score', 'params']]

# Best hyper parameter of logistic regression
#best roc_auc score
```

```
print(rand.best_score_)
# best model parameter
print(rand.best_params_)

#Init logistic model with best param
logit_model =LogisticRegression(penalty= 'l2', max_iter= 5, class_weight= 'balanced', C=
0.0001,random_state =5)
logit_model.fit(X,Y)
confMatrix=run_model(logit_model , X,Y ,"Logistic Regression")

 depth that tree is allowed to grow
#class_weght :( 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)) )

dt_model =DecisionTreeClassifier( max_depth= 5,class_weight='balanced' ,random_state
=5)
#fit model
dt_model.fit(X,Y)

confMatrix=run_model(dt_model , X,Y ,"Decision Tree")# #logistic regression 10 fold
validation
# checking if model is over fitting

logit_scores = cross_validate(logit_model, X, Y, cv=10,
scoring=['accuracy','f1','precision','recall','roc_auc'],)
pd.DataFrame(logit_scores).mean()

# model is not over fitting of model accuracy metric is close to testing data

# lets try to fit data and get classifcation accuracy
#max depth : maximum

rand = RandomizedSearchCV(dt_model, param_dist, cv=5, scoring='roc_auc', n_iter=10,
random_state=5,n_jobs=-1 ,return_train_score=False)
rand.fit(X, Y)
pd.DataFrame(rand.cv_results_)[['mean_test_score', 'std_test_score', 'params']]

# Best hyper parameter of Decision tree
#Best Auc_roc score
print(rand.best_score_)
# Best Decision parmameter learned
print(rand.best_params_)


#Init Decision tree  model with best parameter
```

```
dt_model =DecisionTreeClassifier(class_weight=None ,random_state =5
,min_samples_split= 3, max_depth=5, criterion='entropy')
dt_model.fit(X,Y)# criterion: either 'gini' or 'entropy' based algo for decision tree
# max_depth: max dept the tree will be made to grow
# min_samples_split : minimum no of sample required to split


param_dist =dict(criterion= ['gini', 'entropy'],
          max_depth= [1, 2, 3, 4, 5],
          min_samples_split= [2, 3] ,class_weight=['balanced' ,None] )

# randomized search for given model parameter


confMatrix=run_model(dt_model , X,Y ,"Decision Tree")

# Lets try to make class_wieght balanced and then see if model perform any better
dt_model =DecisionTreeClassifier(class_weight='balanced' ,random_state =5
,min_samples_split= 3, max_depth=5, criterion='entropy')
dt_model.fit(X,Y)

confMatrix=run_model(dt_model , X,Y ,"Decision Tree")

#model performance is similar to base model no impovement but better than tuned
model


#init Gaussian NB model with default parmeters
nb_model =GaussianNB( )

#fit model
nb_model.fit(X,Y)

confMatrix=run_model(nb_model , X,Y ,"Naive  bayes")

#pram_distribution for randomgrid search

#var_smoothing: Portion of the largest variance of all features that is added to variances
for calculation stability.
param_dist = dict( var_smoothing =np.logspace(0,-9, num=20))



#init   RandomizedSearchCV
rand = RandomizedSearchCV(nb_model, param_dist, cv=10, scoring='roc_auc',
n_iter=50,random_state=5, return_train_score=False)
rand.fit(X, Y)
```

```
pd.DataFrame(rand.cv_results_)[['mean_test_score', 'std_test_score', 'params']]

# Best hyper parameter of naive bayes
#Best roc_auc score
print(rand.best_score_)
#Best parameter for naive bayes
print(rand.best_params_)

#init naive bayes with best hyper parameter
nb_model =GaussianNB( var_smoothing= 7.847599703514623e-08)
nb_model.fit(X,Y)
confMatrix=run_model(nb_model , X,Y ,"Naive  bayes")


#After selecting best hyperparameter previous and optimised model have all most same
accuracy parameter

from xgboost import XGBClassifier
xgb = XGBClassifier(max_depth=1,eta =1)
xgb.fit(X,Y)

confMatrix=run_model(xgb , X,Y ,"Xgb")

# tuning model

#Learning_rate:Makes the model more robust by shrinking the weights on each step
#N_estimators : Number of tree to create
#Max_depth :Max depth tree is allowed to grow
#Min_child_weight :Defines the minimum sum of weights of all observations required in a
child.
#Gamma :A node is split only when the resulting split gives a positive reduction in the
loss function. Gamma specifies the minimum loss reduction required to make a split.
#Subsample:Same as the subsample of GBM. Denotes the fraction of observations to be
randomly samples for each tree.
#Colsample_bytree:Similar to max_features in GBM. Denotes the fraction of columns to
be randomly samples for each tree.
#Objective :This defines the loss function to be minimized.
#Scale_pos_weight :A value greater than 0 should be used in case of high class
imbalance as it helps in faster convergence.

xgb = XGBClassifier(learning_rate =0.1,
 n_estimators=800,
 max_depth=5,
 min_child_weight=1,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,a
```

```
 objective= 'binary:logistic',
 nthread=4,
 seed=27,scale_pos_weight=2)
xgb.fit(X,Y)
confMatrix=run_model(xgb , X,Y ,"Xgb")


logistic_base ={'precision': 0.28493852272933656, 'accuracy': 0.781095, 'recall':
0.7806249378047567, 'specificity': 0.7811475136463185, 'fpr': 0.21885248635368146, 'fnr':
0.2193750621952433, 'f1': 0.41748825822589447}
logistic_opti={'precision': 0.2597749380125882, 'accuracy': 0.74837, 'recall':
0.8132152452980396, 'specificity': 0.7411257240052918, 'fpr': 0.25887427599470825, 'fnr':
0.1867847547019604, 'f1': 0.3937649071240995}
dt_base={'precision': 0.16081314495770407, 'accuracy': 0.669755, 'recall':
0.5419942282814211, 'specificity': 0.6840279707840936, 'fpr': 0.31597202921590645, 'fnr':
0.458005771718579, 'f1': 0.24803324415096492}

dt_opti={'precision': 0.672077922077922, 'accuracy': 0.90004, 'recall':
0.010299532291770325, 'specificity': 0.9994385832286468, 'fpr': 0.0005614167713532923,
'fnr': 0.9897004677082297, 'f1': 0.020288150543957658}
nb_base={'precision': 0.7148252154546335, 'accuracy': 0.921695, 'recall':
0.3673002288784954, 'specificity': 0.9836299763204411, 'fpr': 0.016370023679558872, 'fnr':
0.6326997711215047, 'f1': 0.4852588331963845}

nb_opti={'precision': 0.7148252154546335, 'accuracy': 0.921695, 'recall':
0.3673002288784954, 'specificity': 0.9836299763204411, 'fpr': 0.016370023679558872, 'fnr':
0.6326997711215047, 'f1': 0.4852588331963845}
xgb_base={'precision': 1.0, 'accuracy': 0.89952, 'recall': 9.95123892924669e-05,
'specificity': 1.0, 'fpr': 0.0, 'fnr': 0.9999004876107075, 'f1': 0.0001990049751243781}
xgb_opti={'precision': 0.9406936871100118, 'accuracy': 0.977905, 'recall':
0.8326201612100707, 'specificity': 0.9941356961012107, 'fpr': 0.005864303898789341, 'fnr':
0.16737983878992935, 'f1': 0.8833636867527119}


# creating data frame from all model metrics
df =
pd.DataFrame([logistic_base,logistic_opti,dt_base,dt_opti,nb_base,nb_opti,xgb_base,xg
b_opti] ,index=['logit_base','logit_opti' ,'dt_base'
,'dt_opti','nb_base','nb_opti','xgb_base','xgb_opti'] ,columns=['precision' ,'accuracy'
,'recall' ,'specificity','fpr','fnr','f1'])

df.sort_values(by='f1',ascending =False).sort_values(by='fnr').head(10)

# predicting test data
ypred=xgb.predict(test)
sub_df = pd.DataFrame({"ID_code":test_Id_code_orignal.values})
sub_df["target"] = ypred
```

```
sub_df.to_csv("submission_logistic.csv", index=False)
```

References

1.https://towardsdatascience.com/feature-selection-techniques-in-machine-learning-with-python-f24e7da3f36e
2.https://towardsdatascience.com/ways-to-detect-and-remove-the-outliers-404d16608dba

3.https://towardsdatascience.com/analyze-the-data-through-data-visualization-using-seaborn-255e1cd3948e