Project Report

Credit Card Fraud Detection

Aditya Dhaduk (B21AI014)

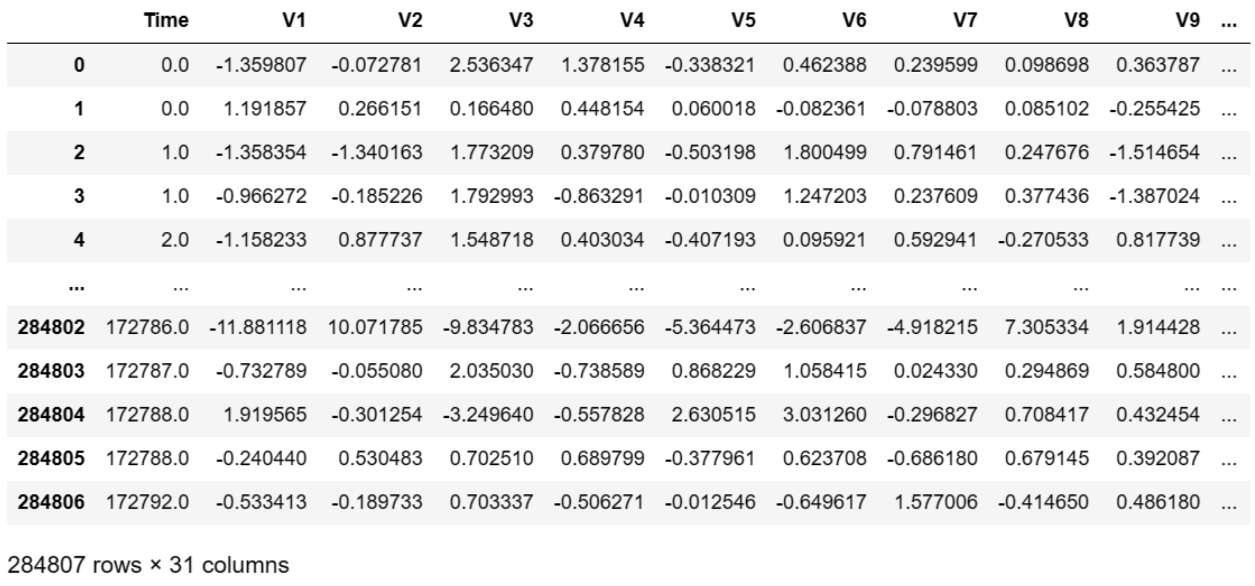
Harsh Soni (B21AI016)

Introduction

The dataset contains transactions made by credit cards in September 2013 by European cardholders.

This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

The dataset has 284807 rows and 31 columns.

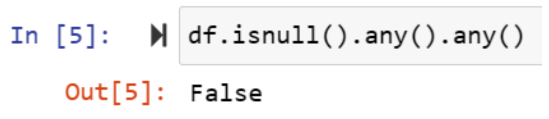


	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...
...	...	...	...	...	...	...	...	...	...	...	...
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...

284807 rows × 31 columns

Exploratory Data Analysis

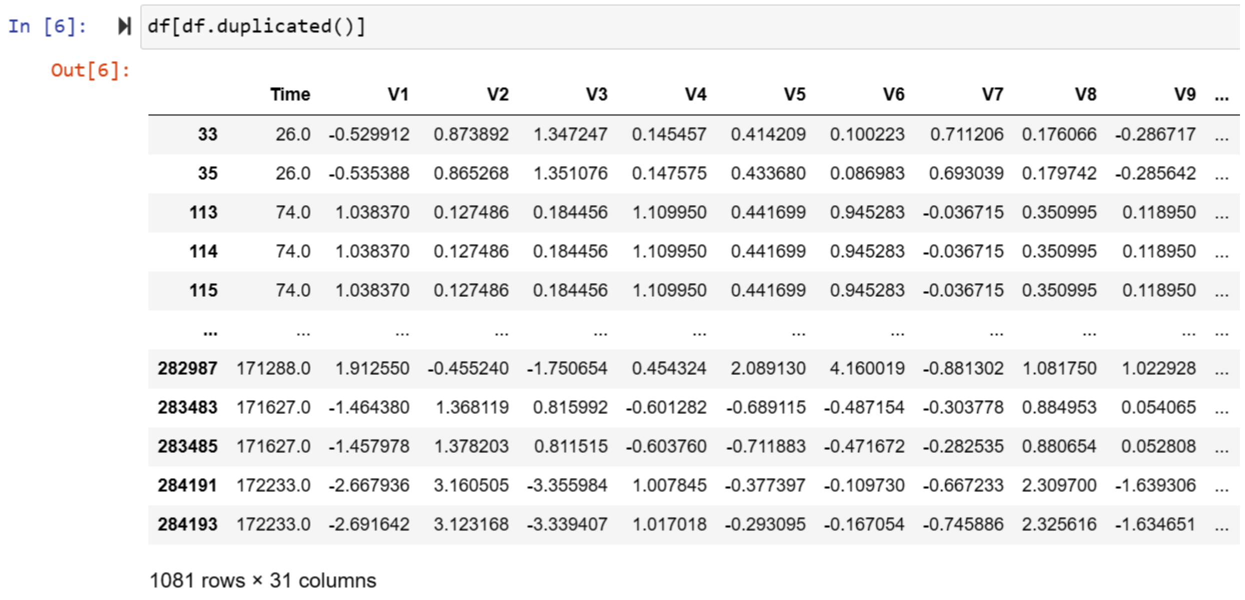
To check for any null values in the dataset:



```
In [5]:  df.isnull().any().any()

   Out[5]: False
```

To check for any duplicate rows in the dataset:



```
In [6]:  df[df.duplicated()]
```

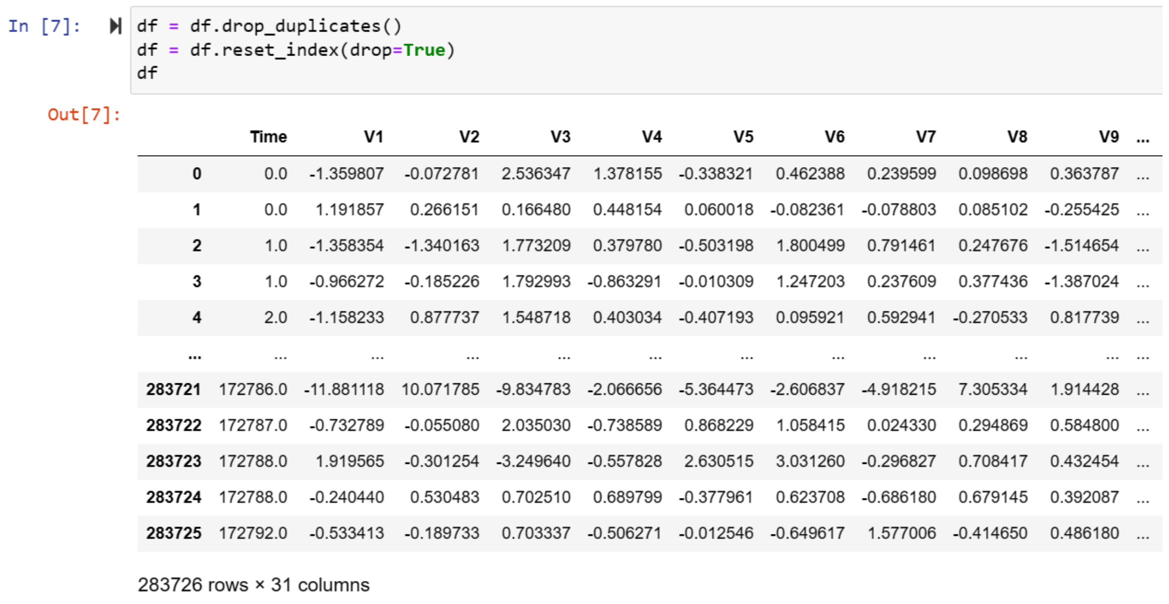
Out[6]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...
33	26.0	-0.529912	0.873892	1.347247	0.145457	0.414209	0.100223	0.711206	0.176066	-0.286717	...
35	26.0	-0.535388	0.865268	1.351076	0.147575	0.433680	0.086983	0.693039	0.179742	-0.285642	...
113	74.0	1.038370	0.127486	0.184456	1.109950	0.441699	0.945283	-0.036715	0.350995	0.118950	...
114	74.0	1.038370	0.127486	0.184456	1.109950	0.441699	0.945283	-0.036715	0.350995	0.118950	...
115	74.0	1.038370	0.127486	0.184456	1.109950	0.441699	0.945283	-0.036715	0.350995	0.118950	...
...	...	...	...	...	...	...	...	...	...	...	...
282987	171288.0	1.912550	-0.455240	-1.750654	0.454324	2.089130	4.160019	-0.881302	1.081750	1.022928	...
283483	171627.0	-1.464380	1.368119	0.815992	-0.601282	-0.689115	-0.487154	-0.303778	0.884953	0.054065	...
283485	171627.0	-1.457978	1.378203	0.811515	-0.603760	-0.711883	-0.471672	-0.282535	0.880654	0.052808	...
284191	172233.0	-2.667936	3.160505	-3.355984	1.007845	-0.377397	-0.109730	-0.667233	2.309700	-1.639306	...
284193	172233.0	-2.691642	3.123168	-3.339407	1.017018	-0.293095	-0.167054	-0.745886	2.325616	-1.634651	...

1081 rows × 31 columns

There are 1081 rows which are duplicates (already existing in the dataset).

To remove all these duplicate rows from the dataset:



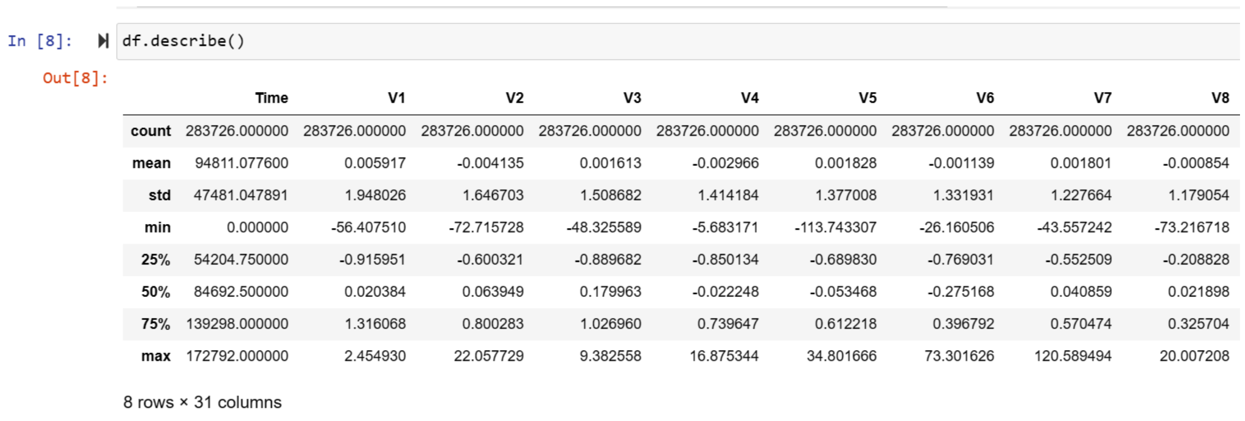
```
In [7]:  df = df.drop_duplicates()
         df = df.reset_index(drop=True)
         df
```

Out[7]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...
...	...	...	...	...	...	...	...	...	...	...	...
283721	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...
283722	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...
283723	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...
283724	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...
283725	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...

283726 rows × 31 columns

To get some general information about the dataset and its features:



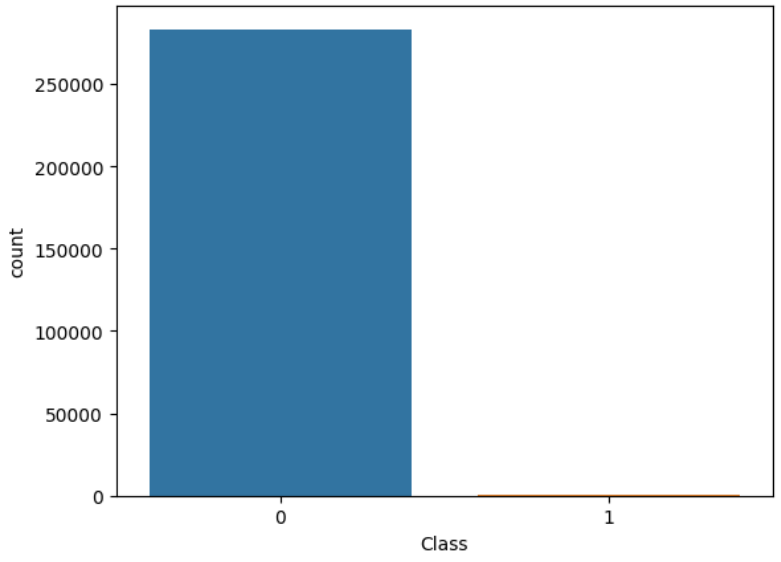
```
In [8]:  df.describe()
```

Out[8]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8
count	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000
mean	94811.077600	0.005917	-0.004135	0.001613	-0.002966	0.001828	-0.001139	0.001801	-0.000854
std	47481.047891	1.948026	1.646703	1.508682	1.414184	1.377008	1.331931	1.227664	1.179054
min	0.000000	-56.407510	-72.715728	-48.325589	-5.683171	-113.743307	-26.160506	-43.557242	-73.216718
25%	54204.750000	-0.915951	-0.600321	-0.889682	-0.850134	-0.689830	-0.769031	-0.552509	-0.208828
50%	84692.500000	0.020384	0.063949	0.179963	-0.022248	-0.053468	-0.275168	0.040859	0.021898
75%	139298.000000	1.316068	0.800283	1.026960	0.739647	0.612218	0.396792	0.570474	0.325704
max	172792.000000	2.454930	22.057729	9.382558	16.875344	34.801666	73.301626	120.589494	20.007208

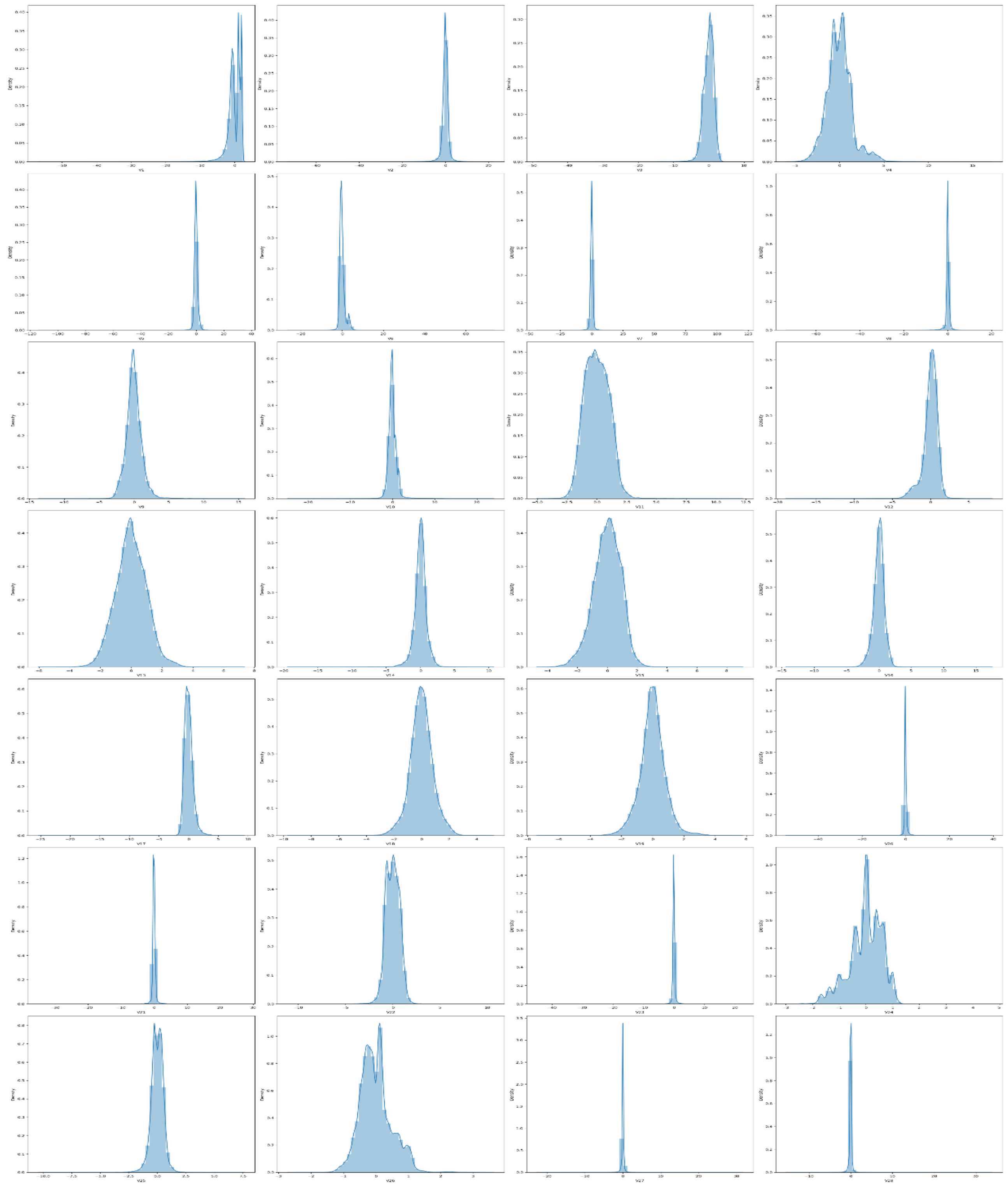
8 rows × 31 columns

To see the number of instances of each class in the dataset using **seaborn.countplot()**.

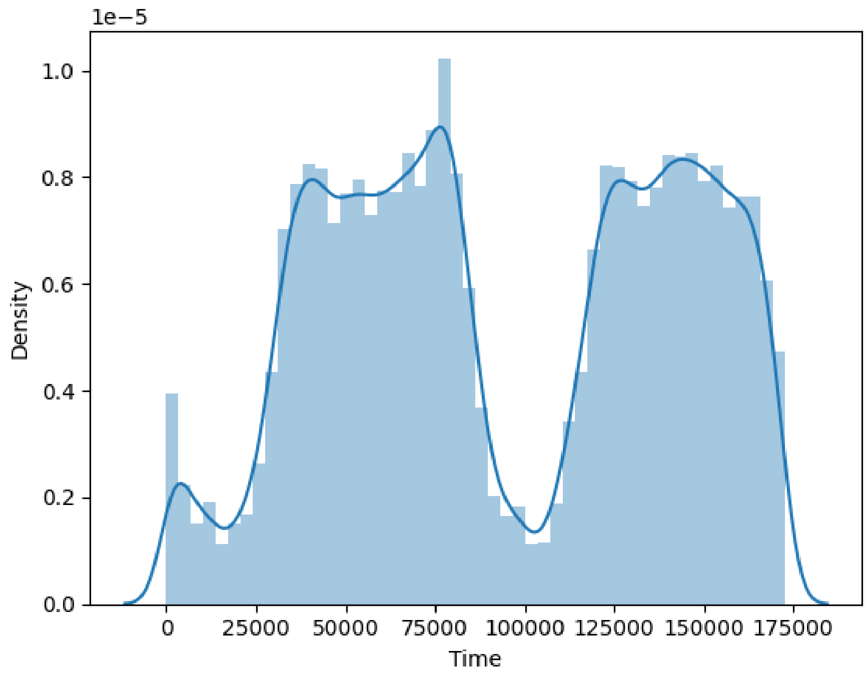


As shown above, class 1 has miniscule number of instances compared to class 0.

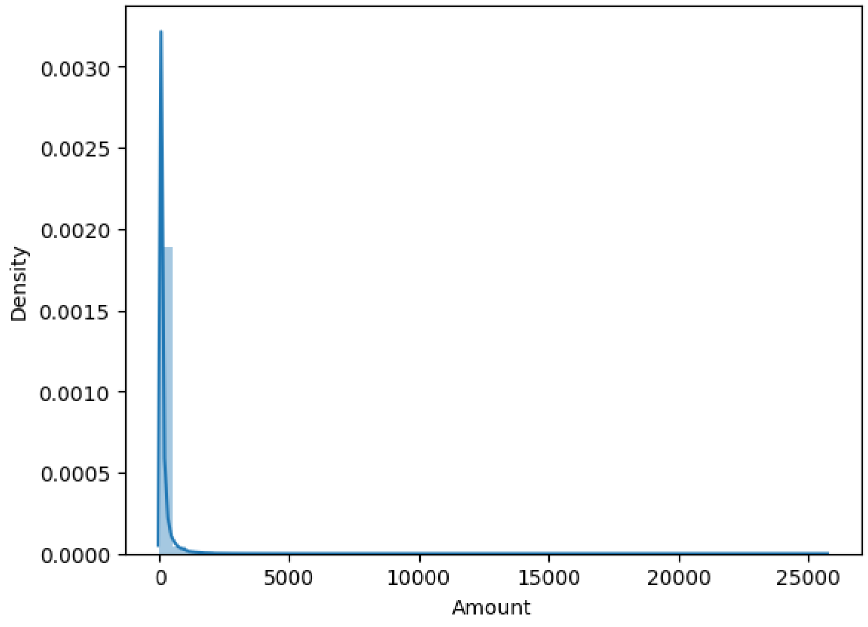
Plotting all the features from V1 to V28:



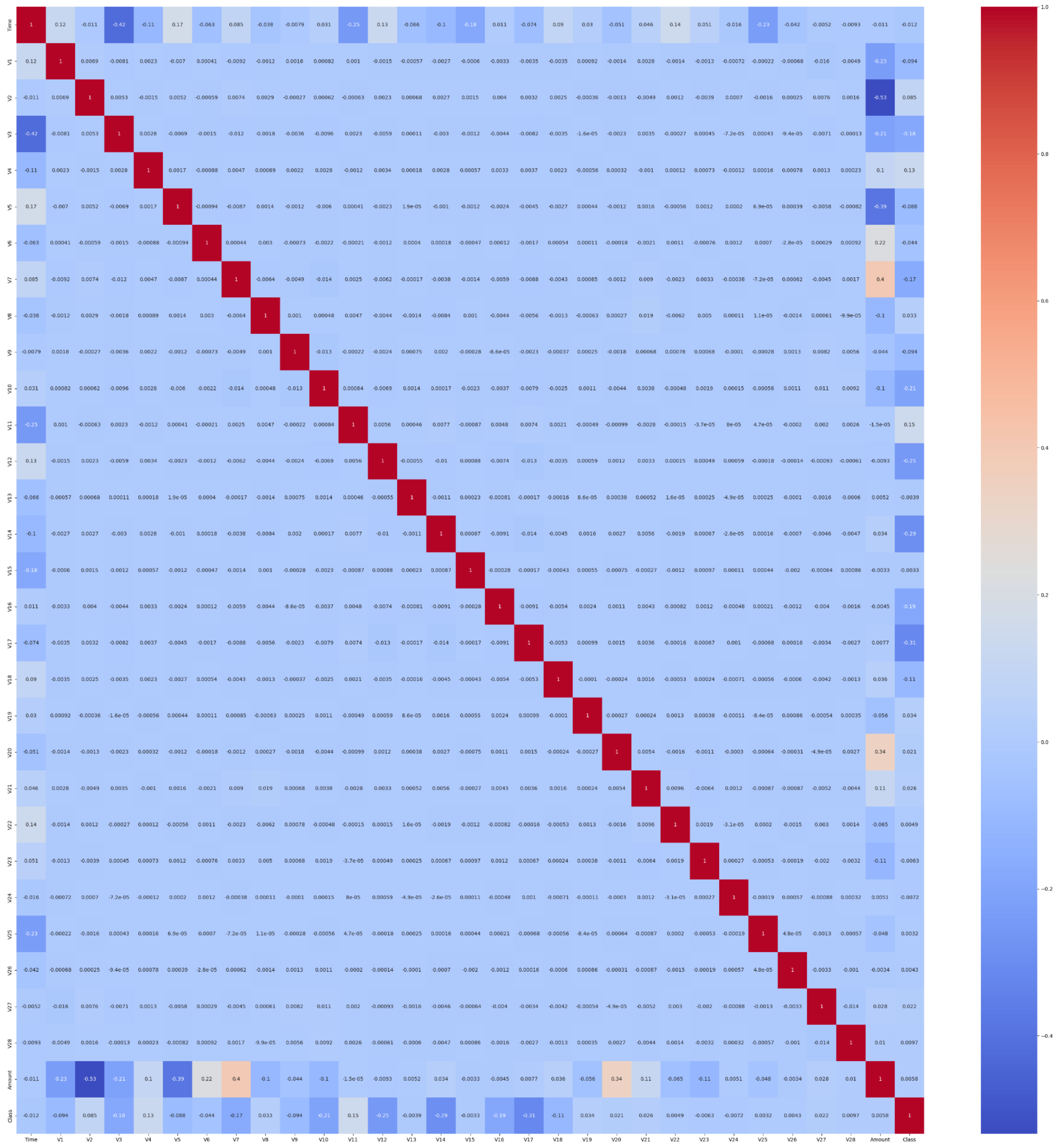
Plotting "Time" feature:



Plotting "Amount" feature:



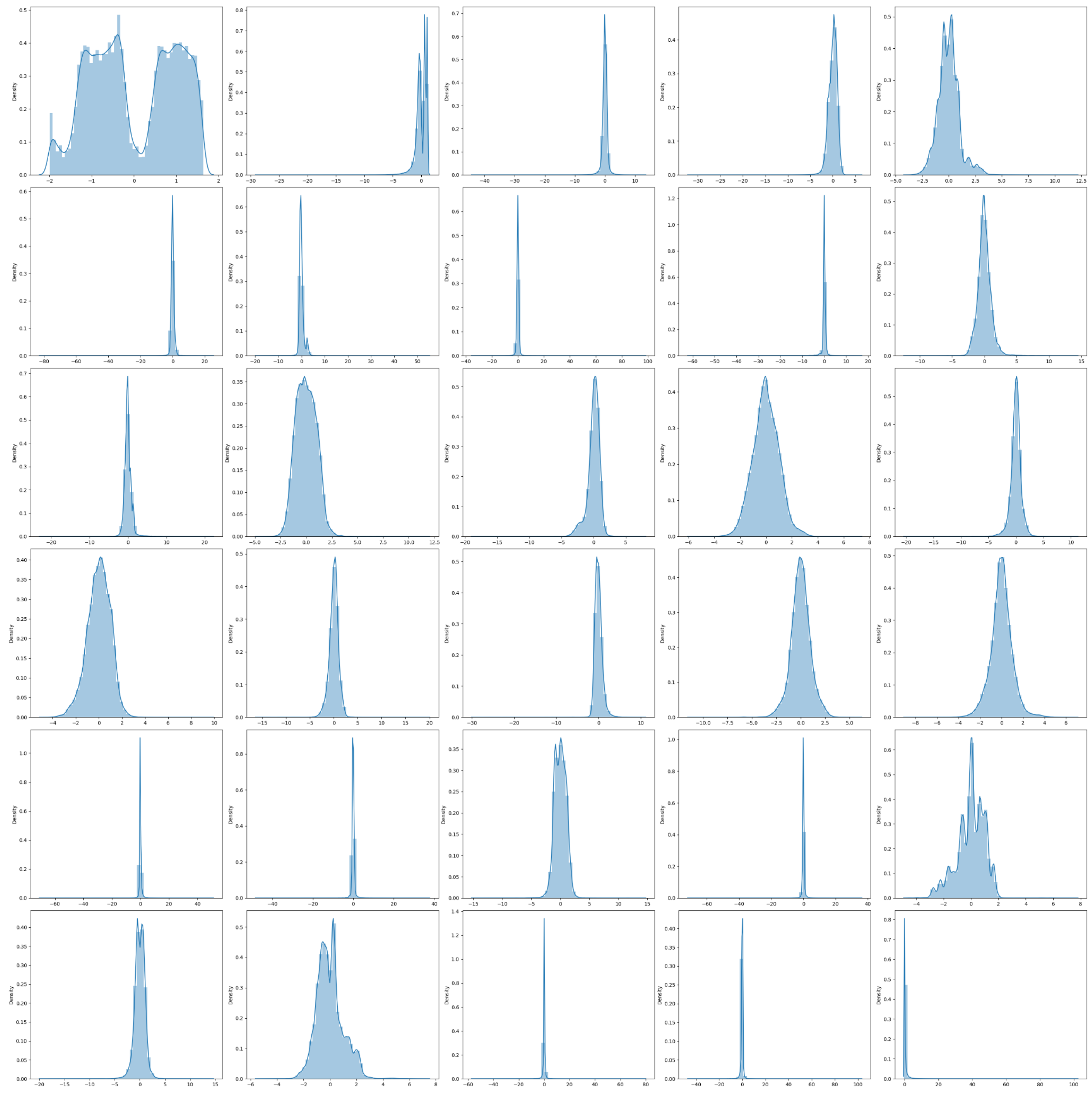
The **Correlation matrix** which shows the relationship between all the features. The darker the color, the higher the relationship.



Scaling the features

Scaling all the features using **StandardScaler()**.

Plotting all the features after scaling.



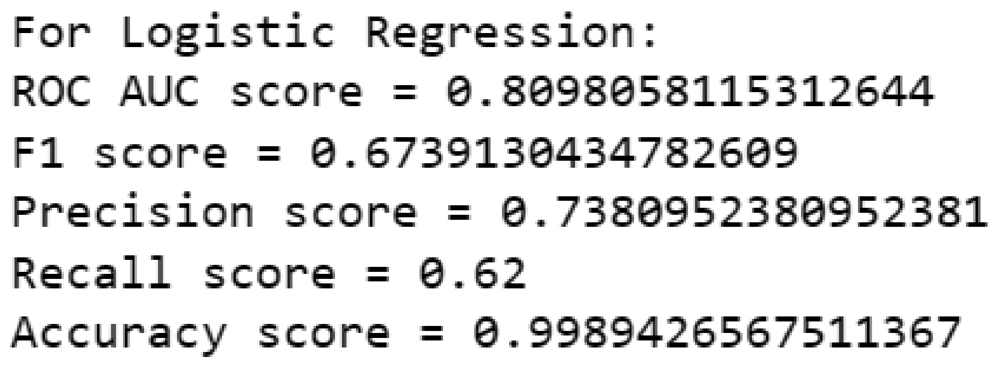
Splitting the dataset

Using **train_test_split()** to split the dataset into train and test set.

Keeping the **train set as 80% and test set as 20%.**

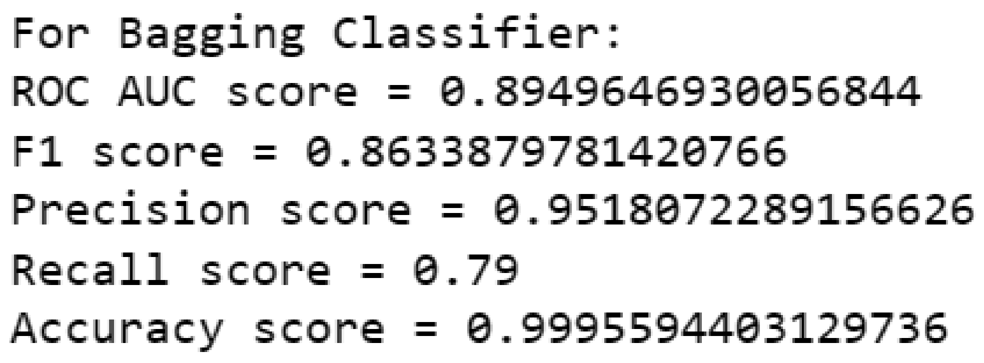
Using different models to train

1. **Logistic Regression:**



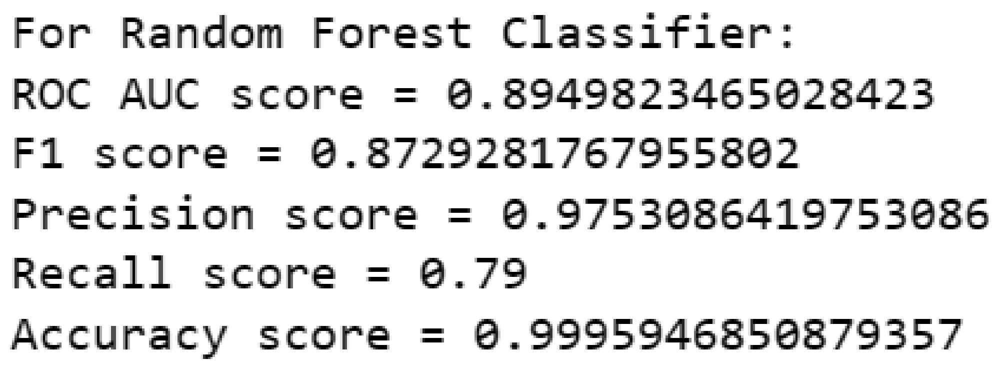
```
For Logistic Regression:
ROC AUC score = 0.8098058115312644
F1 score = 0.6739130434782609
Precision score = 0.7380952380952381
Recall score = 0.62
Accuracy score = 0.9989426567511367
```

2. **Bagging Classifier:**



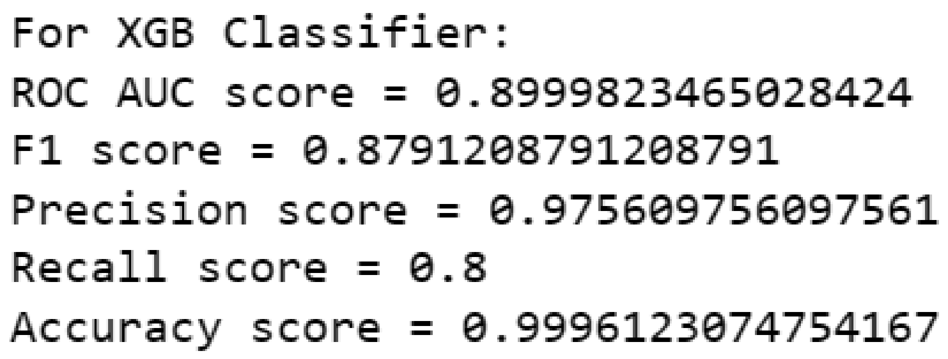
```
For Bagging Classifier:
ROC AUC score = 0.8949646930056844
F1 score = 0.8633879781420766
Precision score = 0.9518072289156626
Recall score = 0.79
Accuracy score = 0.9995594403129736
```

3. **Random Forest Classifier:**



```
For Random Forest Classifier:
ROC AUC score = 0.894982346502842З
F1 score = 0.872928176795580Z
Precision score = 0.9753086419753086
Recall score = 0.79
Accuracy score = 0.9995946850879357
```

4. **XGB Classifier:**



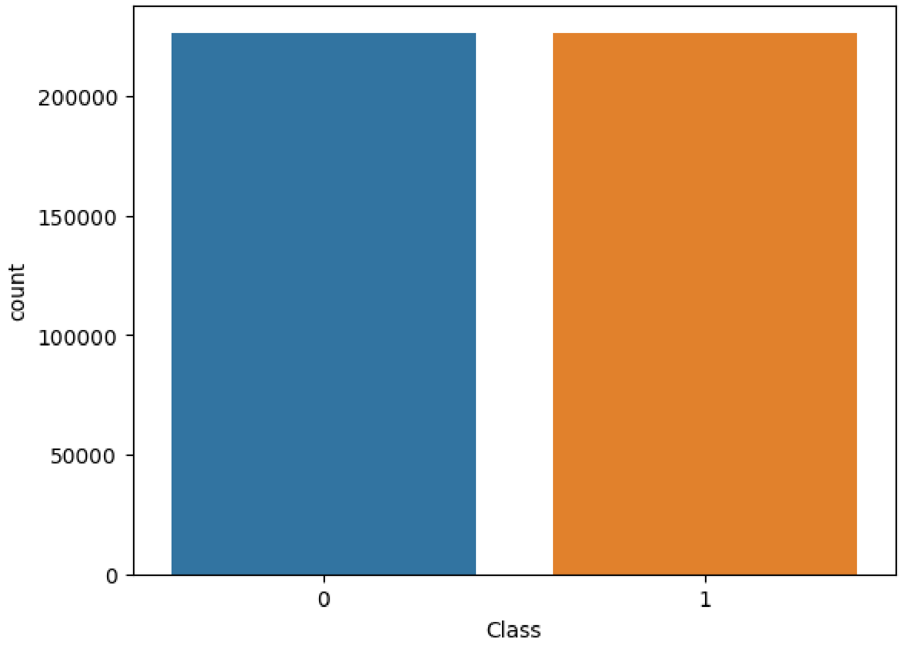
```
For XGB Classifier:
ROC AUC score = 0.8999823465028424
F1 score = 0.8791208791208791
Precision score = 0.975609756097561
Recall score = 0.8
Accuracy score = 0.9996123074754167
```

Training different models after using SMOTE

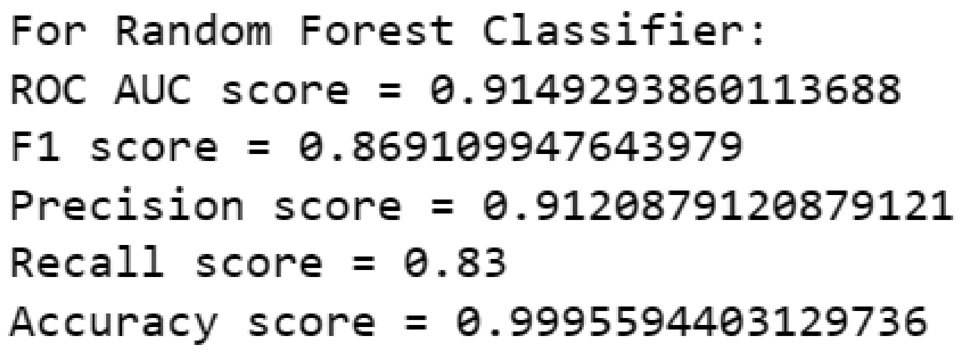
SMOTE - Synthetic Minority Over-sampling Technique.

Synthetic Minority Oversampling Technique (SMOTE) is a **statistical** technique for **increasing** the **number of cases** in your dataset in a **balanced** way. The component works by **generating new instances** from existing **minority cases** that you supply as input.

Using **imblearn** to import SMOTE and after performing over-sampling below are the results which shows that number of instances of class 1 has increased artificially:

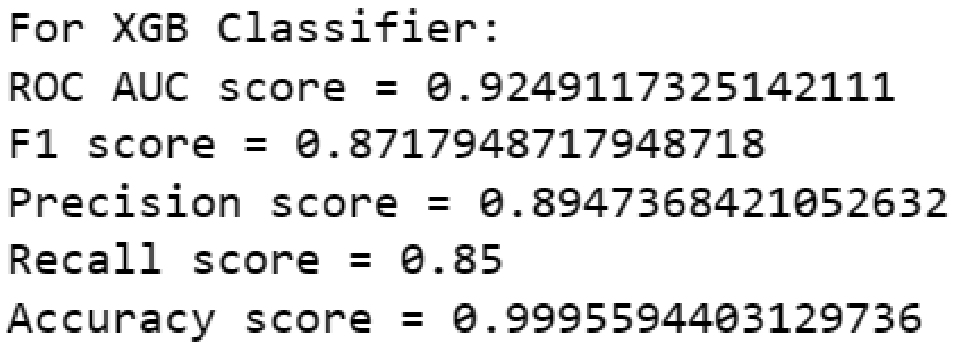


1. **Random Forest Classifier:**



```
For Random Forest Classifier:
ROC AUC score = 0.9149293860113688
F1 score = 0.869109947643979
Precision score = 0.9120879120879121
Recall score = 0.83
Accuracy score = 0.9995594403129736
```

2. **XGB Classifier:**



```
For XGB Classifier:
ROC AUC score = 0.9249117325142111
F1 score = 0.8717948717948718
Precision score = 0.8947368421052632
Recall score = 0.85
Accuracy score = 0.9995594403129736
```

Conclusion

As evident from the above two results of **Random Forest Classifier** and **XGB Classifier**, **XGB classifier** is working **better** for this imbalanced dataset. XGB classifier is better than random forest classifier in terms of **ROC-AUC score** and **F1 score** both. **Accuracy** should not be used in this type of imbalanced dataset as an evaluation metric because even if the model predicts 0 for any input, the accuracy would still be greater than 99% (because class 1 consists of only 0.176% of all the dataset).

Another thing to note is that **all the ensemble learning techniques work better for imbalanced datasets** whether it be **bagging classifier, random forest classifier or extreme gradient boosting classifier.**