

CREDIT CARD FRAUD DETECTION AND ENERGY EFFICIENCY MODEL USING MACHINE LEARNING ALGORITHMS

Anireddy Sujit Reddy (01987338)

I. INTRODUCTION

This project aims to evaluate different machine learning methods across two data sets, one for classification and the other for regression. The first part seeks to detect fraudulent credit card transactions over non-fraudulent transactions and predict Fraud fast and reliably using machine learning techniques. Our objective here is to see 100% of fraudulent transactions while minimizing the number of incorrect fraud classifications. We have covered several classification and regression models during this course. I stated three models for each of them in this project. Using a credit card fraud detection data set, I constructed classification models using random forest, decision trees, and logistic regression. I used the energy efficiency data set to develop regression models using polynomial, ridge, and linear regression for the project's other half. The accuracy scores for the random forest, decision trees, and logistic regression are 94%, 99.91%, and 99.90%, respectively. Similarly, the mean square errors for linear regression, polynomial regression, and ridge regression are 0.916, 0.998, and 0.911, respectively. Finally, I assessed each model's performance. I generated the confusion matrix to evaluate the credit card fraud model and the mean square error values for the energy efficiency model.

II. DATA SET 1: CREDIT CARD FRAUD DETECTION

a. The data set

The first data set that this project works with is Credit Card Fraud Detection. This is a binary classification problem with two classes: Fraud and not-fraud. The dataset was collected from

transactions made by European cardholders (credit cards) in September 2013. The dataset is not ideal and unbalanced; over 284,807 genuine transactions compared to just 492 (0.172%) fraudulent transactions. This dataset contains 28 features that have been computed from Principal Component Analysis (PCA). The only features which have not been transformed with PCA are 'Time' and 'Amount.' Fraud is indicated by a 1 for the feature 'Class' and a 0 otherwise. After preprocessing the data, three classification algorithms are used to train models for this dataset.

b. Examining the data set

I started by first examining the data set. I looked for possible duplicate and null values and then looked at the distribution of each dataset column. Then sci-kit learn's standard scaler was used to scale the data. As mentioned before, the data set is heavily imbalanced; there are only 492 fraudulent transactions and over 2,84,807 non-fraudulent transactions. Training any machine learning model directly on the dataset is not ideal.

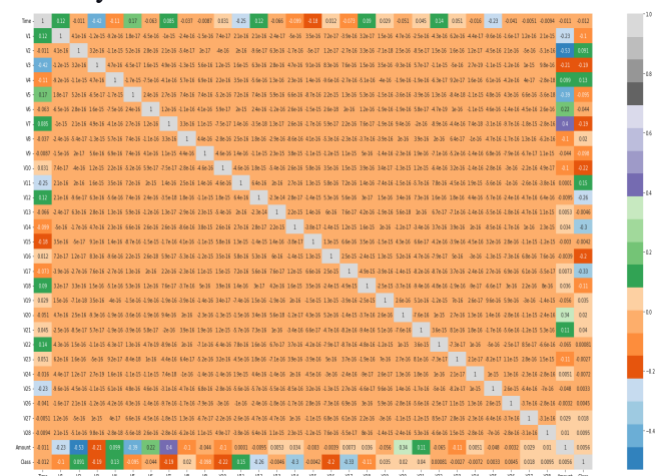


Fig.1 - Correlation Matrix (Heat Map)

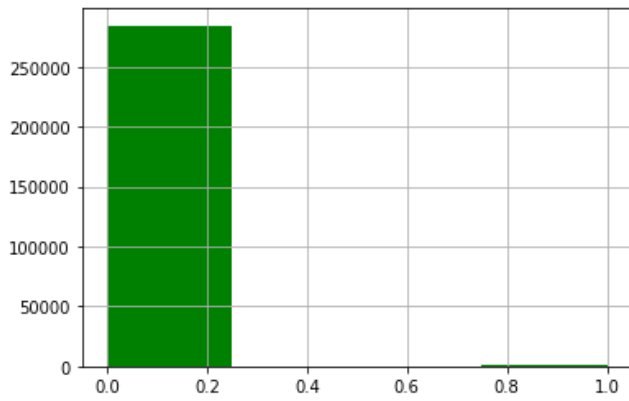


Fig.2 – Histogram of Colum “Class”

A balanced sampling technique was used to create a new dataset that can be used for training. This sampling approach separates the data into Fraud and non-fraud transactions, stored as valid and invalid variables. The incorrect variables include information from the dataset, which selects 492 non-fraudulent transactions at random. The non-fraudulent dataset is stored in the variables. The dataset was split into training, validation, and testing sets. A training dataset is a set of data used to train the model. A validation dataset is a subset of the original dataset used to evaluate model competence while tweaking the model's hyperparameters. The assessment becomes progressively skewed when a quality from the validation dataset is employed in the model configuration. The test dataset is a subset of data utilized to assess the final model fit on the training dataset objectively.

c. Logistic Regression

One of the three algorithms I used was logistic regressions. Logistic Regression in regression analysis is used to estimate the parameters of a logistic model. Logistic Regression does predictive analysis and is based on the idea of probability. A Logistic Regression model is like a Linear Regression model. However, the Logistic Regression utilizes a more sophisticated cost function, which may be characterized as the 'Sigmoid function' or

sometimes known as the 'logistic function' rather than a linear function. The logistic regression hypothesis restricts the cost function between 0 and 1. As a result, linear functions fail to describe it since they might have a value larger than one or less than 0, which is not feasible according to the logistic regression hypothesis.

The Logistic Regression model produced considerably better outcomes. Given Logistic Regression's simplicity, this model exceeded my expectations regarding the classification report (accuracy, recall, and f1score).

accuracy			1.00	85443
macro avg	0.94	0.79	0.85	85443
weighted avg	1.00	1.00	1.00	85443
Accuracy: 99.9098814414288%				

Fig.3 - Classification Report of Logistic Regression Classifier

d. Decision Tree Classifier

A decision tree incorporates a tree-like model of decisions and potential outcomes, such as resource costs, chance event outcomes and utility. It is one way of showing an algorithm made up entirely of conditional control statements. A decision tree is a flowchart-like structure containing a node representing a "test" on some attribute. The metrics used to evaluate this model were accuracy, precision, recall, F1-score, and confusion matrix.

accuracy			1.00	85443
macro avg	0.90	0.88	0.89	85443
weighted avg	1.00	1.00	1.00	85443
Accuracy: 99.92275552122467%				

Fig.4 – Evaluation of Decision Tree Model

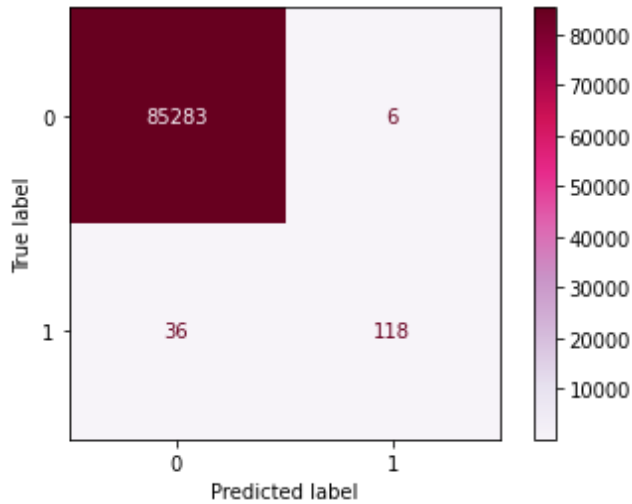


Fig.5 –Confusion Matrix for Decision Tree Model

e. Random Forest Algorithm

Random forests, also known as random decision forests, are a group learning approach for classification, Regression, and other problems that works by generating many decision trees during training. One of the essential characteristics of the Random Forest Algorithm is that it can handle both classification and regression datasets. Out of all three algorithms, random forest outperforms other algorithms.

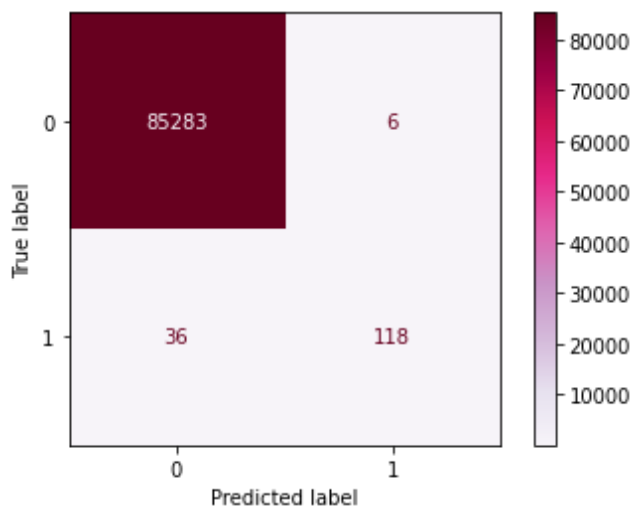


Fig.6 –Confusion Matrix for Random Forest Algorithm

f. Solving class imbalance

Oversampling the minority class is one method for dealing with unbalanced datasets. The most basic method includes copying instances from the minority class, even if these examples offer no new information to the model. Instead, new instances may be created by combining old ones. The Synthetic Minority Oversampling Technique (SMOTE) is a kind of data augmentation for the minority class. In simple terms: The number of fraud rows is very few compared to the number of non-fraud rows. This imbalance leads to a flawed model. To fix this, we can artificially generate new samples to compensate for the number difference. Using existing data, we can 'make' the latest data. This is called oversampling.

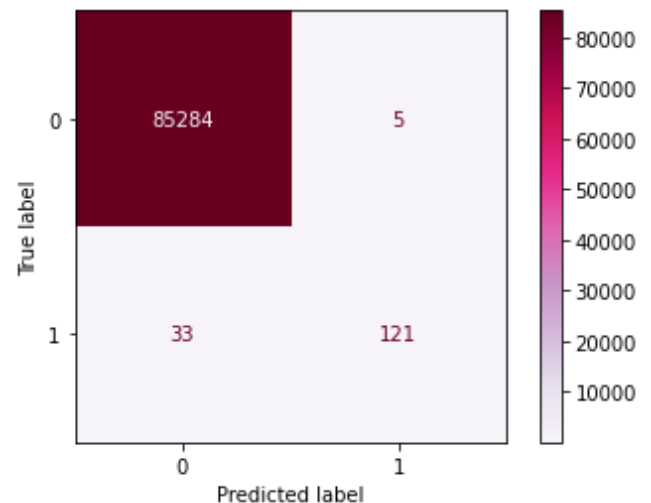


Fig.7 –Confusion Matrix for Random Forest Algorithm after SMOT

g. Metrics

Confusion matrices are matrices that represent counts from expected and actual values. The output "TN" stands for True Negative and represents the number of correctly identified negative cases. Similarly, "TP" stands for True

Positive, which denotes the number of correctly identified positive occurrences. The abbreviation "FP" stands for False Positive value, which is the number of actual negative cases categorized as positive; "FN" stands for False Negative value, which is the number of actual positive examples classified as negative. Accuracy is one of the most widely utilized criteria while doing categorization. The following formula is used to calculate the accuracy of a model (through a confusion matrix).

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

Three machine learning algorithms were used to detect credit card fraud from the given dataset. 70% of the dataset was used for training and 30% for testing and validation to evaluate the algorithms. The logistic regression, decision tree, and random forest classifiers have accuracy scores of 99 percent, 99 percent, and 94 percent, respectively. The results of the comparison show that the Random Forest technique outperforms the Logistic Regression and decision tree techniques. While this appears to be excessive in comparison to the findings obtained with the other models, it is important to recall that under-sampling was used. One possible explanation for this outcome is that the dataset's fraudulent and non-fraudulent occurrences must be balanced.

III. DATA SET 2 - REGRESSION DATASET: ENERGY EFFICIENCY DATASET

a. The data set

The second data set that was used was the Energy Efficiency data set. The dataset contains eight attributes or features, denoted by X1...X8 and two responses or outcomes, represented by y1 and y2. Our objective is to predict the energy.

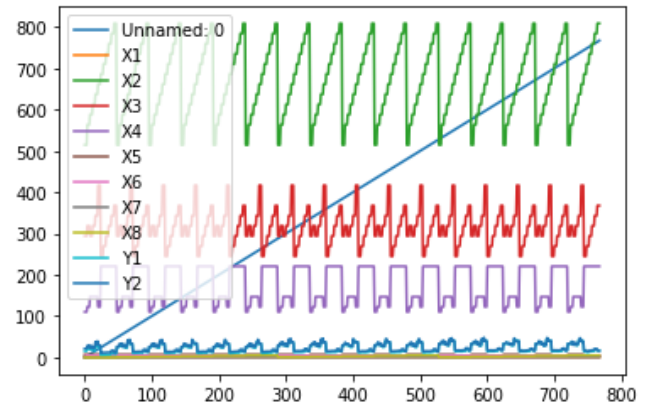


Fig 1. Plot of the whole data set

b. Data Exploration

The first thing I did when I started working on a new data collection was to investigate it. To better display and comprehend the data, I created a series of graphs. I began by visualizing the data set. The first is used to examine the correlation between each characteristic in the data set and to create a correlation coefficient matrix heat map, as shown in the picture below.



Fig 2. Correlation between each variable from the dataset

c. Polynomial Regression

Like many other machine learning concepts, polynomial regression is based on statistics. When there is a non-linear relationship between the value of xx and the related conditional mean of yy,

statistical analysis is performed. The general equation of a polynomial regression model.

$$y = b_0 + b_1x_1 + b_2\{x_1\}^2 + b_3\{x_1\}^3 + \dots + b_n\{x_1\}^n$$

d. Support Vector Regression

Support Vector Regression is a supervised learning approach for predicting discrete values. The same logic underpins Support Vector Regression as it does SVMs. SVR's primary concept is to locate the optimum fit line. The best fit line in SVR is the hyperplane with the greatest number of points.

e. Linear Regression

A linear technique to modeling the connection between a scalar response and one or more explanatory factors is known as linear regression. Linear regression is a statistical method for modeling the connection between a scalar response (or dependent variable) and one or more explanatory factors (or independent variables). Simple linear regression is used when there is only one explanatory variable. Linear regression is a popular and widely used regression approach. It is one of the most basic regression algorithms. One of its key benefits is the simplicity with which the results may be understood. Most of the time, when performing basic linear regression, you start with a predefined set of input-output (x-y) pairs. The following observations have been made by you. Multiple or multivariate linear regression is used when there are two or more independent variables.

IV. CONCLUSION

Overall, the models produced rather accurate findings. As we predicted, the accuracy of the models improved as they progressed. This may be shown by the fact that the Polynomial Regression is the highest of all. Continue examining the data and looking for items that don't have a high association to improve

these models. With more time, one can find a technique to reduce the dimensionality of the data, removing features that have minimal effect on the regression problem. We discovered that the polynomial regression model has a superior effect, with a score of 0.998, suggesting that our data set is more consistent with the polynomial regression model. This paper also explains in detail how machine learning may be used to improve fraud detection results, including the algorithm, pseudocode, description of its implementation, and experimentation results.

V. FUTURE WORKS

While we did not reach our goal of 100 percent accuracy in fraud detection, we did design a system that, given enough time and data, can get very close. As with any endeavor of this magnitude, there is room for improvement. Because of the nature of this project, numerous algorithms may be linked as modules, and their results may be pooled to optimize the accuracy of the final output. These models can be improved further by introducing other algorithms. However, the output of these algorithms must be in the same format as the others. Once that requirement is fulfilled, the modules may be readily added, as seen in the code. As a result, the project has a high degree of adaptability and versatility. The dataset includes further development opportunities. As previously stated, the precision of the algorithms increases as the size of the dataset increases. Therefore, more data will surely increase the model's effectiveness in detecting fraud while reducing the number of false positives. This, however, requires explicit approval from the banks themselves.

```

import sklearn
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import KFold, cross_validate
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import roc_curve, roc_auc_score
import itertools
from collections import Counter
from sklearn.manifold import TSNE
from sklearn import preprocessing
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
%matplotlib inline

```

```
df = pd.read_csv ("creditcard 2.csv")
```

```
df.head()
```

```
df.info()
```

```

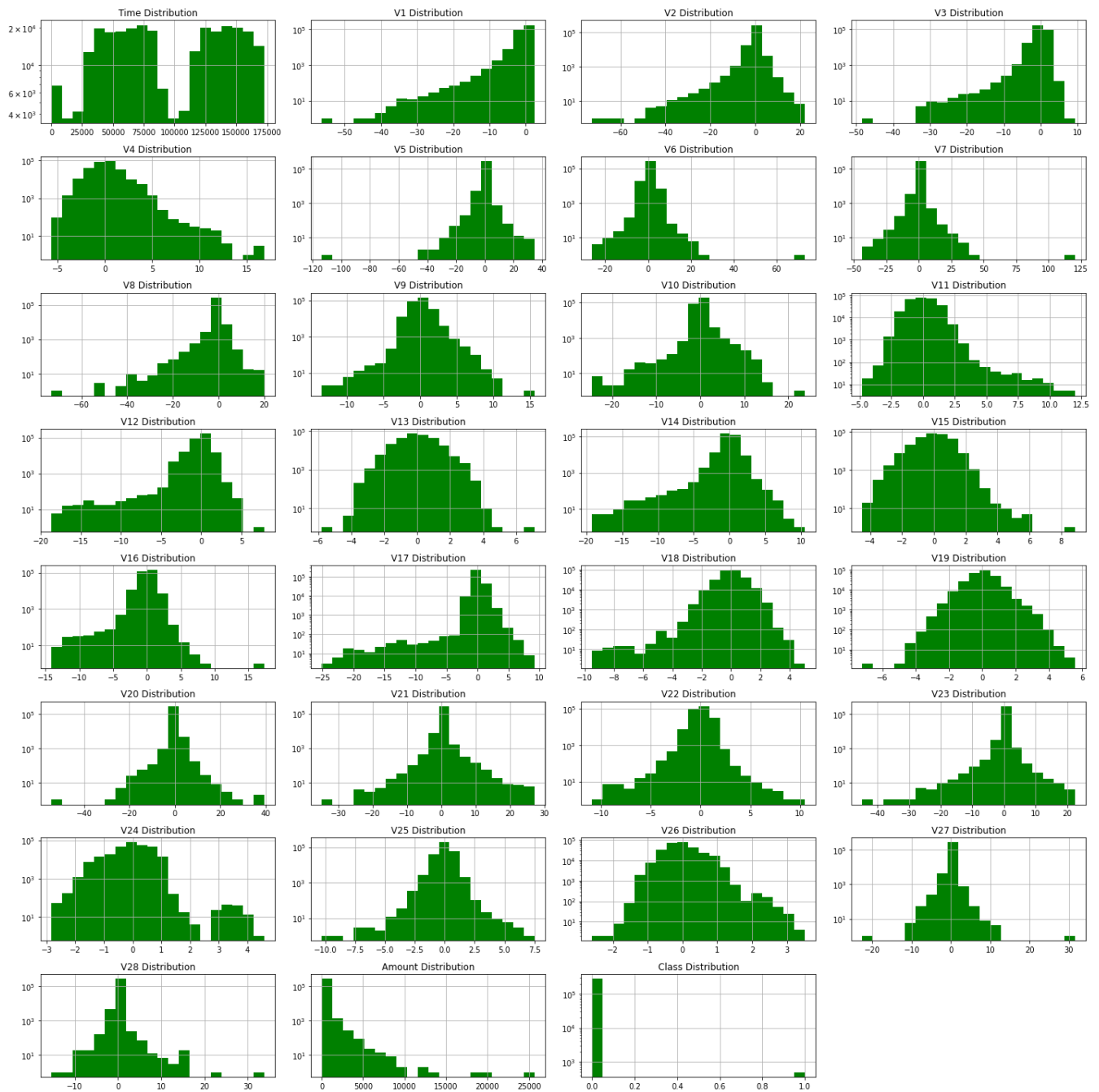
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0   Time        284807 non-null  float64
 1   V1          284807 non-null  float64
 2   V2          284807 non-null  float64
 3   V3          284807 non-null  float64
 4   V4          284807 non-null  float64
 5   V5          284807 non-null  float64
 6   V6          284807 non-null  float64
 7   V7          284807 non-null  float64
 8   V8          284807 non-null  float64
 9   V9          284807 non-null  float64
10  V10         284807 non-null  float64
11  V11         284807 non-null  float64
12  V12         284807 non-null  float64
13  V13         284807 non-null  float64
14  V14         284807 non-null  float64
15  V15         284807 non-null  float64
16  V16         284807 non-null  float64
17  V17         284807 non-null  float64
18  V18         284807 non-null  float64
19  V19         284807 non-null  float64

```

```
20  V20      284807 non-null  float64
21  V21      284807 non-null  float64
22  V22      284807 non-null  float64
23  V23      284807 non-null  float64
24  V24      284807 non-null  float64
25  V25      284807 non-null  float64
26  V26      284807 non-null  float64
27  V27      284807 non-null  float64
28  V28      284807 non-null  float64
29  Amount   284807 non-null  float64
30  Class    284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

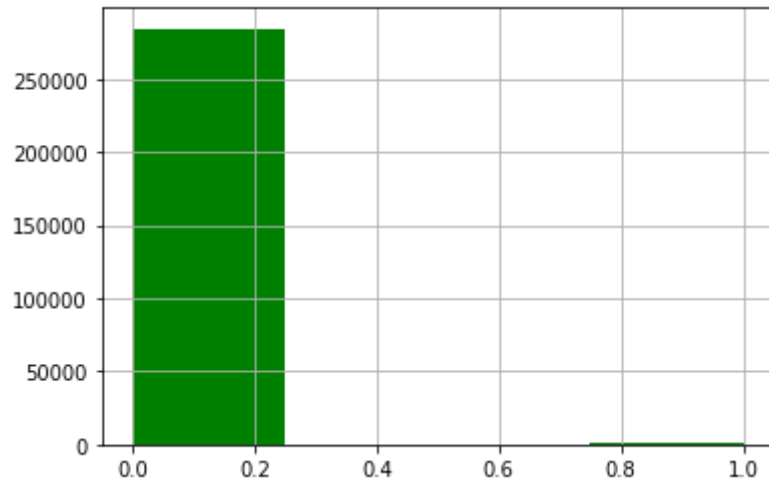
```
#round(100 * (df.isnull().sum()/len(card)),2).sort_values(ascending=False)
#round(100 * (df.isnull().sum(axis=1)/len(card)),2).sort_values(ascending=False)
#no null values in the dataset
```

```
fig=plt.figure(figsize=(20,20))
for i, feature in enumerate(df.columns):
    ax=fig.add_subplot(8,4,i+1)
    df[feature].hist(bins=20,ax=ax,facecolor='green')
    ax.set_title(feature+" Distribution",color='black')
    ax.set_yscale('log')
fig.tight_layout()
plt.show()
```



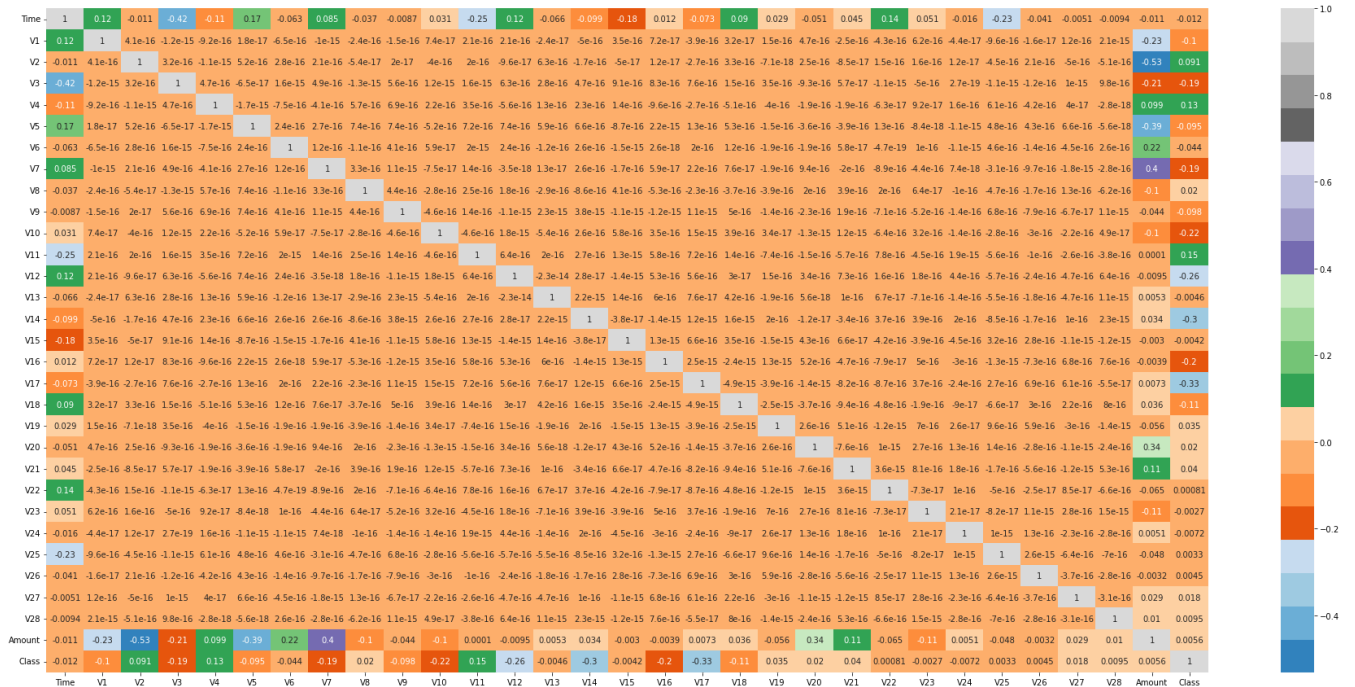

```
#visualising the class column  
plt.grid(False)  
df["Class"].hist(bins=4, facecolor='green')  
print(df["Class"].value_counts())
```

```
0    284315  
1      492  
Name: Class, dtype: int64
```



▼ Correlation Matrix

```
plt.figure(figsize = (32,15))  
sns.heatmap(df.corr(), annot = True, cmap="tab20c")  
plt.show()
```



Processing the DataSet

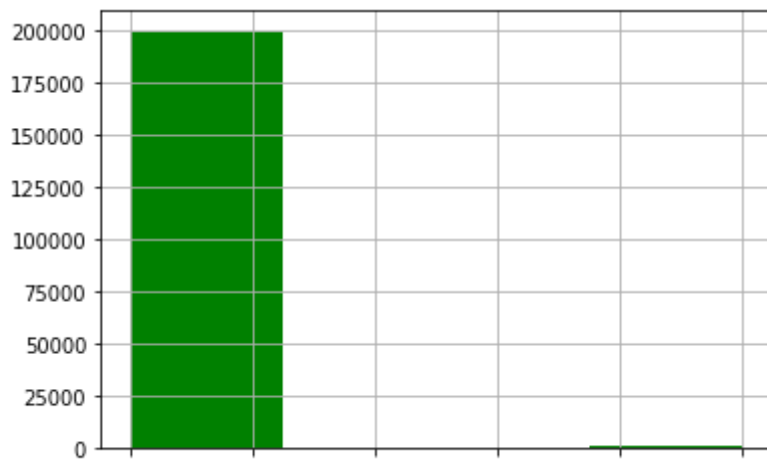
```
#Scaling
scaler = StandardScaler()
df["n_amount"] = scaler.fit_transform(df["Amount"].values.reshape(-1, 1))
#dropping amount and time columns, not needed
df.drop(["Amount", "Time"], inplace= True, axis= 1)

y = df["Class"]
X = df.drop(["Class"], axis= 1)

#train test splits
# Split the data
(X_train, X_test, y_train, y_test) = train_test_split(X, y, test_size= 0.3, random_st

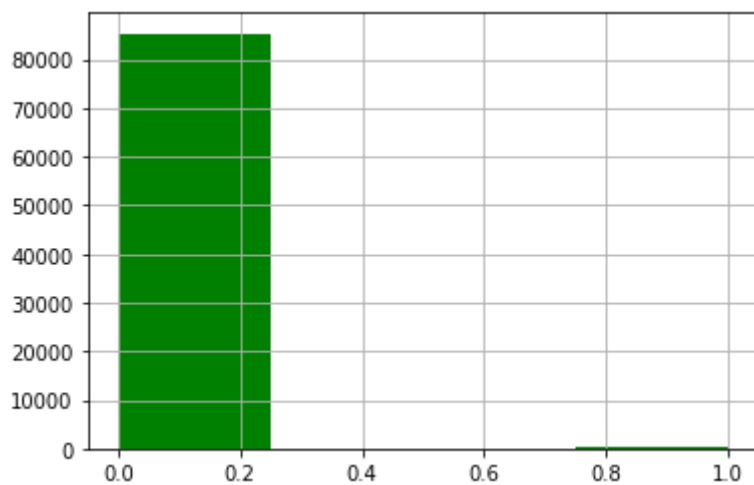
# checking the distribution of the split
print(y_train.value_counts())
plt.grid(False)
y_train.hist(bins=4,facecolor='green')
plt.show()
```

```
0    199026
1      338
Name: Class, dtype: int64
```



```
print(y_test.value_counts())
y_test.hist(bins=4, facecolor='green')
plt.show()
```

```
0    85289
1     154
Name: Class, dtype: int64
```



```
#model object
#running logistic regression
model = LogisticRegression()
model.fit(X_train, y_train)

#predicting and metrics
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=["Not-Fraud", "Fraud"]))
print(f'Accuracy: {str(accuracy_score(y_test, y_pred) * 100)}%')
```

	precision	recall	f1-score	support
Not-Fraud	1.00	1.00	1.00	85289
Fraud	0.88	0.58	0.70	154

accuracy			1.00	85443
macro avg	0.94	0.79	0.85	85443
weighted avg	1.00	1.00	1.00	85443

Accuracy: 99.9098814414288%

Decision Tree classifier

```
# Decision Tree Classifier
decision_tree = DecisionTreeClassifier()
decision_tree.fit(X_train, y_train)

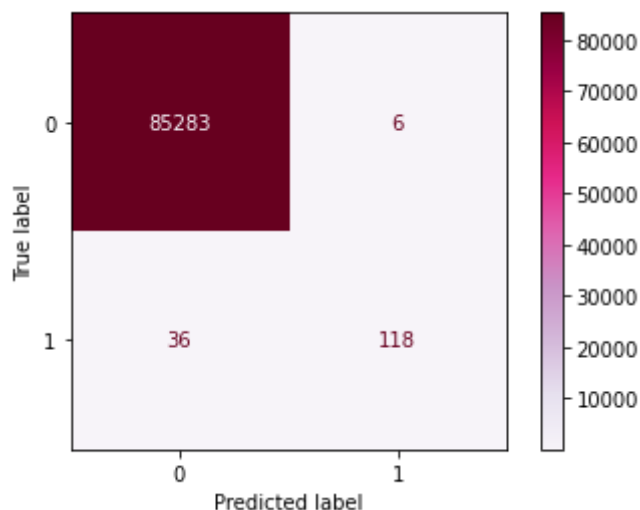
y_pred = decision_tree.predict(X_test)
print(classification_report(y_test, y_pred, target_names=["Not-Fraud", "Fraud"]))
print(f'Accuracy: {str(accuracy_score(y_test, y_pred) * 100)}%')
```

	precision	recall	f1-score	support
Not-Fraud	1.00	1.00	1.00	85289
Fraud	0.80	0.76	0.78	154
accuracy			1.00	85443
macro avg	0.90	0.88	0.89	85443
weighted avg	1.00	1.00	1.00	85443

Accuracy: 99.92275552122467%

```
# Confusion Matrix
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, cmap="PuRd")
```

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f707daee950



Random Forest Classifier

```
# Random Forest
random_forest = RandomForestClassifier(n_estimators= 100)
random_forest.fit(X_train, y_train)

y_pred = random_forest.predict(X_test)
print(classification_report(y_test, y_pred, target_names=["Not-Fraud", "Fraud"]))
print(f'Accuracy: {str(accuracy_score(y_test, y_pred) * 100)}%')
```

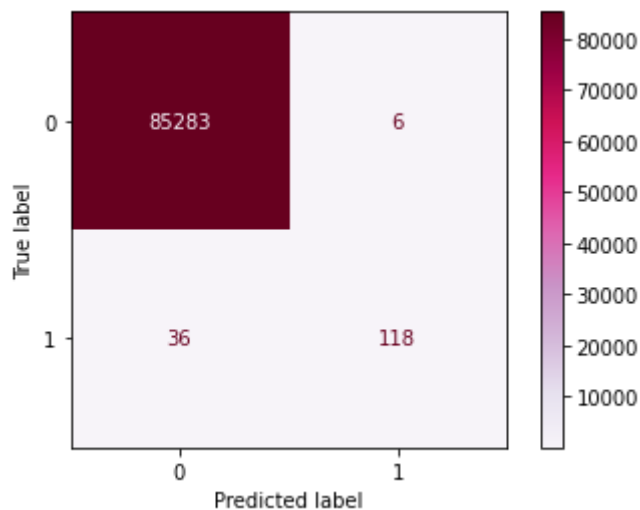
	precision	recall	f1-score	support
Not-Fraud	1.00	1.00	1.00	85289
Fraud	0.95	0.77	0.85	154
accuracy			1.00	85443
macro avg	0.98	0.88	0.92	85443
weighted avg	1.00	1.00	1.00	85443

Accuracy: 99.95084442259751%

```
# Plot confusion matrix for Random Forests
# Confusion Matrix
```

```
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, cmap="PuRd")
```

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f707d1d9ed0



Solving Class Imbalance:

The number of fraud rows are very few compared to the number of non-fraud rows. This imbalance leads to a bad model.

To fix this, we can artificially generate new samples to compensate for the number difference. Using existing data, we can 'make' new data. This is called over sampling.

This is known as the Synthetic Minority Oversampling Technique (SMOTE)

```
# Performing oversampling on RF and DT
```

```

from imblearn.over_sampling import SMOTE

X_new, y_new = SMOTE().fit_resample(X, y)

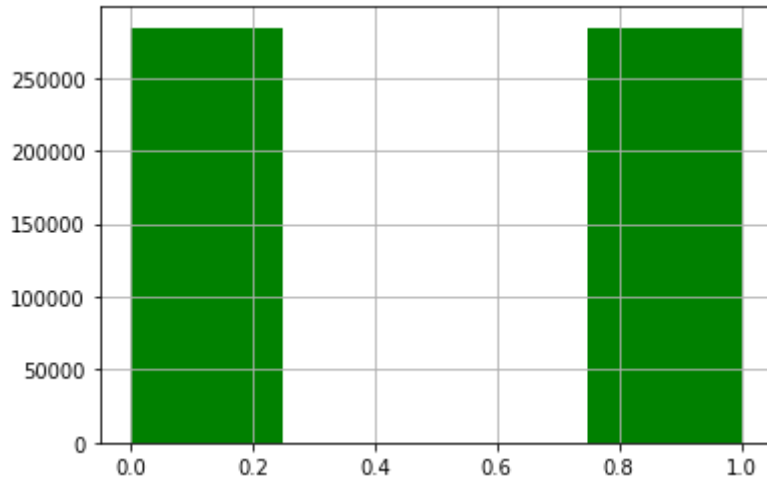
print(y_new.value_counts())
plt.grid(False)
y_new.hist(bins=4, facecolor='green')
plt.show()

```

```

0    284315
1    284315
Name: Class, dtype: int64

```



```

(X_train, X_test, y_train, y_test) = train_test_split(X, y, test_size= 0.3, random_st

# Build the Random Forest classifier on the new dataset
random_forest = RandomForestClassifier(n_estimators= 100)
random_forest.fit(X_train, y_train)

y_pred = random_forest.predict(X_test)
print(classification_report(y_test, y_pred, target_names=["Not-Fraud", "Fraud"]))
print(f'Accuracy: {str(accuracy_score(y_test, y_pred) * 100)}%')

```

	precision	recall	f1-score	support
Not-Fraud	1.00	1.00	1.00	85289
Fraud	0.96	0.79	0.86	154
accuracy			1.00	85443
macro avg	0.98	0.89	0.93	85443
weighted avg	1.00	1.00	1.00	85443

Accuracy: 99.95552590615966%

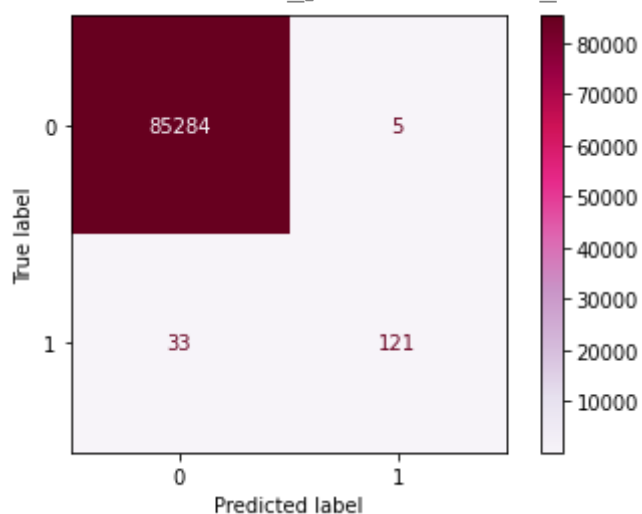
```

# Plot confusion matrix for Random Forests
# Confusion Matrix

```

```
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, cmap="PuRd")
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f707d110050
```



Now it is evident that after addressing the class imbalance problem, our Random forest classifier with SMOTE performs far better than the Random forest classifier without SMOTE

Now it is evident that after addressing the class imbalance problem, our Random forest classifier with SMOTE performs far better than the Random forest classifier without SMOTE

```
url="https://archive.ics.uci.edu/ml/machine-learning-databases/00242/ENB2012_data.xls"
```

```
import pandas as pd
dataset=pd.read_excel(url)
```

```
dataset
```



1 to 25 of 768 entries

Filter



index	X1	X2	X3	X4	X5	X6	X7	X8	Y1	Y2
0	0.98	514.5	294.0	110.25	7.0	2	0.0	0	15.55	21.33
1	0.98	514.5	294.0	110.25	7.0	3	0.0	0	15.55	21.33
2	0.98	514.5	294.0	110.25	7.0	4	0.0	0	15.55	21.33
3	0.98	514.5	294.0	110.25	7.0	5	0.0	0	15.55	21.33
4	0.9	563.5	318.5	122.5	7.0	2	0.0	0	20.84	28.28
5	0.9	563.5	318.5	122.5	7.0	3	0.0	0	21.46	25.38
6	0.9	563.5	318.5	122.5	7.0	4	0.0	0	20.71	25.16
7	0.9	563.5	318.5	122.5	7.0	5	0.0	0	19.68	29.6
8	0.86	588.0	294.0	147.0	7.0	2	0.0	0	19.5	27.3
9	0.86	588.0	294.0	147.0	7.0	3	0.0	0	19.95	21.97
10	0.86	588.0	294.0	147.0	7.0	4	0.0	0	19.34	23.49
11	0.86	588.0	294.0	147.0	7.0	5	0.0	0	18.31	27.87
12	0.82	612.5	318.5	147.0	7.0	2	0.0	0	17.05	23.77
13	0.82	612.5	318.5	147.0	7.0	3	0.0	0	17.41	21.46
14	0.82	612.5	318.5	147.0	7.0	4	0.0	0	16.95	21.16
15	0.82	612.5	318.5	147.0	7.0	5	0.0	0	15.98	24.93
16	0.79	637.0	343.0	147.0	7.0	2	0.0	0	28.52	37.73
17	0.79	637.0	343.0	147.0	7.0	3	0.0	0	29.9	31.27
18	0.79	637.0	343.0	147.0	7.0	4	0.0	0	29.63	30.93
19	0.79	637.0	343.0	147.0	7.0	5	0.0	0	28.75	39.44
20	0.76	661.5	416.5	122.5	7.0	2	0.0	0	24.77	29.79
21	0.76	661.5	416.5	122.5	7.0	3	0.0	0	23.93	29.68
22	0.76	661.5	416.5	122.5	7.0	4	0.0	0	24.77	29.79
23	0.76	661.5	416.5	122.5	7.0	5	0.0	0	23.93	29.4
24	0.74	686.0	245.0	220.5	3.5	2	0.0	0	6.07	10.9

Show 25 per page

1

2

10

30

31

Like what you see? Visit the [data table notebook](#) to learn more about interactive tables.

```
dataset.to_csv("data.csv",encoding='utf-8')
```

```
df=pd.read_csv("./data.csv")
```

```
df
```


	Unnamed: 0	x1	x2	x3	x4	x5	x6	x7	x8	y1	y2
0	0	0.98	514.5	294.0	110.25	7.0	2	0.0	0	15.55	21.33
1	1	0.98	514.5	294.0	110.25	7.0	3	0.0	0	15.55	21.33
2	2	0.98	514.5	294.0	110.25	7.0	4	0.0	0	15.55	21.33
3	3	0.98	514.5	294.0	110.25	7.0	5	0.0	0	15.55	21.33
4	4	0.90	563.5	318.5	122.50	7.0	2	0.0	0	20.84	28.28
...
763	763	0.64	784.0	343.0	220.50	3.5	5	0.4	5	17.88	21.40
764	764	0.62	808.5	367.5	220.50	3.5	2	0.4	5	16.54	16.88
765	765	0.62	808.5	367.5	220.50	3.5	3	0.4	5	16.44	17.11
766	766	0.62	808.5	367.5	220.50	3.5	4	0.4	5	16.48	16.61
767	767	0.62	808.5	367.5	220.50	3.5	5	0.4	5	16.64	16.03

768 rows x 11 columns

```
df.isnull().sum()
```

```

Unnamed: 0    0
x1            0
x2            0
x3            0
x4            0
x5            0
x6            0
x7            0
x8            0
y1            0
y2            0
dtype: int64

```

```
df.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 11 columns):
#   Column          Non-Null Count  Dtype
---  -
0   Unnamed: 0      768 non-null   int64
1   x1              768 non-null   float64
2   x2              768 non-null   float64
3   x3              768 non-null   float64
4   x4              768 non-null   float64
5   x5              768 non-null   float64
6   x6              768 non-null   int64

```

```

7    X7          768 non-null    float64
8    X8          768 non-null    int64
9    Y1          768 non-null    float64
10   Y2          768 non-null    float64
dtypes: float64(8), int64(3)
memory usage: 66.1 KB

```

```
df.describe()
```

	Unnamed: 0	x1	x2	x3	x4	x5	x6	
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	7
mean	383.500000	0.764167	671.708333	318.500000	176.604167	5.250000	3.500000	
std	221.846794	0.105777	88.086116	43.626481	45.165950	1.75114	1.118763	
min	0.000000	0.620000	514.500000	245.000000	110.250000	3.500000	2.000000	
25%	191.750000	0.682500	606.375000	294.000000	140.875000	3.500000	2.750000	
50%	383.500000	0.750000	673.750000	318.500000	183.750000	5.250000	3.500000	
75%	575.250000	0.830000	741.125000	343.000000	220.500000	7.000000	4.250000	

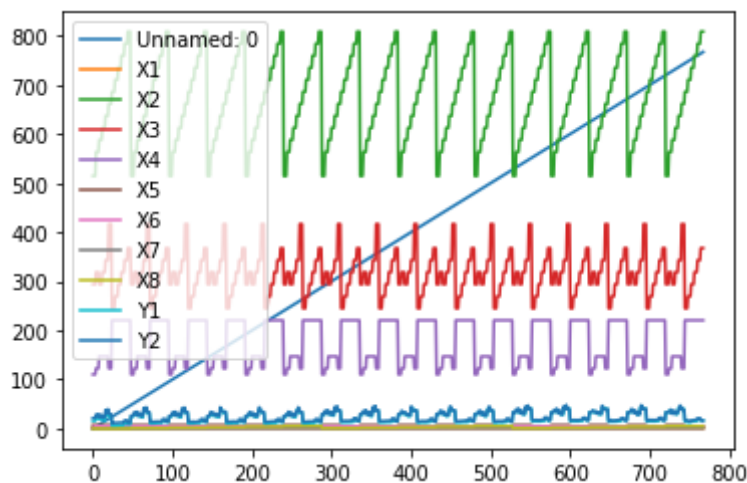
```

tar1=df['Y1']
tar2=df['Y2']
data=df.drop(columns=['Y1','Y2'])

```

```
df.plot()
```

<matplotlib.axes._subplots.AxesSubplot at 0x7fc0157de090>



```
print(tar1,tar2,data)
```

```
0      15.55
```

```
1      15.55
2      15.55
3      15.55
4      20.84
```

```
...
```

```
763    17.88
764    16.54
765    16.44
766    16.48
767    16.64
```

```
Name: Y1, Length: 768, dtype: float64 0      21.33
```

```
1      21.33
2      21.33
3      21.33
4      28.28
```

```
...
```

```
763    21.40
764    16.88
765    17.11
766    16.61
767    16.03
```

```
Name: Y2, Length: 768, dtype: float64      Unnamed: 0      X1      X2      X3      X4
0          0  0.98  514.5  294.0  110.25  7.0  2  0.0  0
1          1  0.98  514.5  294.0  110.25  7.0  3  0.0  0
2          2  0.98  514.5  294.0  110.25  7.0  4  0.0  0
3          3  0.98  514.5  294.0  110.25  7.0  5  0.0  0
4          4  0.90  563.5  318.5  122.50  7.0  2  0.0  0
..      ...      ...      ...      ...      ...      ..      ...      ..
763      763  0.64  784.0  343.0  220.50  3.5  5  0.4  5
764      764  0.62  808.5  367.5  220.50  3.5  2  0.4  5
765      765  0.62  808.5  367.5  220.50  3.5  3  0.4  5
766      766  0.62  808.5  367.5  220.50  3.5  4  0.4  5
767      767  0.62  808.5  367.5  220.50  3.5  5  0.4  5
```

```
[768 rows x 9 columns]
```

```
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
```

```
data=sc.fit_transform(data)
```

```
pd.DataFrame(data)
```

	0	1	2	3	4	5	6	7	
0	-1.729797	2.041777	-1.785875	-0.561951	-1.470077	1.0	-1.341641	-1.760447	-1.8145
1	-1.725286	2.041777	-1.785875	-0.561951	-1.470077	1.0	-0.447214	-1.760447	-1.8145
2	-1.720776	2.041777	-1.785875	-0.561951	-1.470077	1.0	0.447214	-1.760447	-1.8145
3	-1.716265	2.041777	-1.785875	-0.561951	-1.470077	1.0	1.341641	-1.760447	-1.8145
4	-1.711755	1.284979	-1.229239	0.000000	-1.198678	1.0	-1.341641	-1.760447	-1.8145
...
763	1.711755	-1.174613	1.275625	0.561951	0.972512	-1.0	1.341641	1.244049	1.4113

```
#train test split
```

```
from sklearn.model_selection import train_test_split
```

```
x_train1,x_test1,y_train1,y_test1=train_test_split(data,tar1,test_size=0.3)
```

```
x_train2,x_test2,y_train2,y_test2=train_test_split(data,tar2,test_size=0.3)
```

767	1.729797	-1.363812	1.553943	1.123903	0.972512	-1.0	1.341641	1.244049	1.4113
-----	----------	-----------	----------	----------	----------	------	----------	----------	--------

```
print(x_train1,x_train2)
```

```
[[-0.57509548  0.24438254 -0.39428407 ... -1.34164079 -1.00932293
  1.41133622]
 [ 1.43661107 -0.51241501  0.44067043 ...  0.4472136  1.2440492
  0.76615395]
 [-0.7645387  -0.03941654 -0.1159659  ...  0.4472136  -1.00932293
  0.76615395]
 ...
 [ 0.25033568  1.28497917 -1.22923856 ...  1.34164079  0.11736313
  0.76615395]
 [ 0.48037387  0.90658039 -0.95092039 ...  0.4472136  0.11736313
  1.41133622]
 [ 0.66530654  2.04177671 -1.78587489 ...  1.34164079  1.2440492
 -1.16939287]] [[-1.50877991  2.04177671 -1.78587489 ... -0.4472136  -1.00932293
 -1.16939287]
 [ 0.87279197  2.04177671 -1.78587489 ... -0.4472136  1.2440492
 -0.5242106 ]
 [ 1.69371258 -0.98541347  0.99730676 ...  1.34164079  1.2440492
  1.41133622]
 ...
 [ 0.85474976 -1.36381225  1.55394308 ... -0.4472136  1.2440492
 -1.16939287]
 [ 0.21425126 -1.36381225  1.55394308 ...  1.34164079  0.11736313
  0.12097168]
 [-0.77355981 -0.03941654 -0.1159659  ... -1.34164079 -1.00932293
  0.76615395]]
```

▼ Linear Regression

```
from sklearn.linear_model import LinearRegression
```

```
from sklearn.linear_model import LinearRegression
lr=LinearRegression()

#for Y1
lr.fit(x_train1,y_train1)

LinearRegression()

from sklearn.metrics import r2_score,mean_squared_error

print(r2_score(lr.predict(x_train1),y_train1))
print(mean_squared_error(lr.predict(x_train1),y_train1))

0.9131137939105459
8.410099944641606

#for y2
lr.fit(x_train2,y_train2)

LinearRegression()

print(r2_score(lr.predict(x_train2),y_train2))
print(mean_squared_error(lr.predict(x_train2),y_train2))

0.8780522610079334
9.688472451337635

#test for Y1
print(r2_score(lr.predict(x_test1),y_test1))
print(mean_squared_error(lr.predict(x_test1),y_test1))

0.8230420106459628
13.443488602858773

#test for Y2
print(r2_score(lr.predict(x_test2),y_test2))
print(mean_squared_error(lr.predict(x_test2),y_test2))

0.8716554581216548
10.681939978575457

import numpy as np

print("Y1 prediction test:",lr.predict(x_test1))
```

```
Y1 prediction test: [32.26165889 32.65154714  9.73311014 15.81549429 11.95562672
19.72714571 17.82789954 15.7537019  36.41555342 19.62460075 18.09968026
```

```

34.24354132 36.97940793 16.13640897 19.827025 15.66083589 20.03646252
12.98135736 17.33606718 15.5035118 14.37587574 31.69780439 12.51246796
15.91803926 17.99980097 32.18336921 26.87687084 15.5367003 17.31882415
15.80091694 31.87832806 17.54017335 34.01059324 38.27373605 14.19962671
30.50112464 14.60882281 19.31528394 18.25433867 17.51858273 31.68841763
17.06428646 18.08510291 38.11907764 16.72178115 15.63657959 19.06074624
30.17051042 19.3222972 17.78279937 32.46141747 34.04378275 15.80848102
29.60665592 10.71576675 16.80014277 36.6462706 16.04844136 12.5270611
17.23085654 16.49082595 18.6037843 14.89455713 14.99443642 15.09698138
30.57941432 30.51080359 14.06655893 19.57248731 19.67236659 15.69402438
35.99675132 17.33340151 29.5518768 12.41750286 29.45199751 36.49161219
28.98449782 17.62112769 35.92775769 17.79737672 30.40124536 16.37233553
34.88796726 32.847103 30.34646624 16.1005548 40.41027688 15.70860173
37.86453995 15.28216261 15.25163979 31.85275499 13.96667965 12.66979204
12.31762357 16.01525287 19.17030448 33.34324426 15.45841163 31.84278384
34.52934757 33.96549307 32.59256531 17.52124841 14.9396573 15.84868279
28.18862799 14.01879308 16.92153973 36.13719521 31.38877977 30.06703371
39.69176397 37.90964012 30.14670766 16.00067551 18.50390501 17.67590681
35.23313825 14.22121734 32.08776558 15.76338085 18.3396406 17.13364293
33.06074324 19.27018377 19.16062553 30.30136607 17.28563566 14.7087021
28.0887487 17.53582576 15.64625853 33.37006005 30.42475592 16.10322048
12.67204055 30.96509986 29.50677663 39.94630167 32.22846938 16.62190187
18.7584427 14.35428512 40.21051831 35.84209292 15.43682101 31.99744225
30.76534129 30.71056217 15.33694172 12.72681967 38.01919835 27.56219424
19.26751809 34.42946828 33.80543137 17.84514257 32.10700049 28.68485996
14.25440583 15.89378296 15.60605677 35.64233434 16.20309977 19.06775951
15.18228332 19.52472146 28.28850728 33.11552236 31.85246279 30.04682838
16.97898453 31.99773444 34.06537235 17.52826168 36.54639131 17.59060488
18.86098767 15.65382262 17.13097725 17.9304445 15.35151908 27.87931118
19.88180412 15.9458964 19.42217649 36.62499095 29.91568053 31.33400065
36.31567414 31.92342823 35.25206319 17.69749743 38.17385676 17.36392432
15.86059446 17.59761815 29.9610729 15.40629819 13.75457644 19.31794961
38.47349462 40.0558599 16.35775817 30.07063113 35.8278784 36.05153044
30.665462 31.77844877 36.33695379 17.28830134 31.76876982 27.07662941
12.25526458 17.42136912 32.86098466 30.07033894 31.97820735 34.12015147
30.47953504 30.27009751 38.52827374 34.27480988 36.15140973 19.21807033
16.1579996 32.49268602 19.01298039]

```

```
Y1_pred=lr.predict(x_test1)
```

```
pd.DataFrame(Y1_pred).describe()
```

0 **count** 231.000000**mean** 24.008396

#Y2 prediction

Y2_pred=lr.predict(x_test2)

pd.DataFrame(Y2_pred).describe()

0 **count** 231.000000**mean** 25.381448**std** 9.142784**min** 10.516008**25%** 16.404191**50%** 27.562194**75%** 33.895080**max** 40.410277

pd.DataFrame(Y2_pred).to_csv('Y2_lr_test.csv')

pd.DataFrame(Y1_pred).to_csv('Y1_lr_test.csv')

▼ Polynomial Regression

from sklearn.preprocessing import PolynomialFeatures

poly=PolynomialFeatures()

p_train1=poly.fit_transform(x_train1)

p_train2=poly.fit_transform(x_train2)

p_test1=poly.fit_transform(x_test1)

p_test2=poly.fit_transform(x_test2)

pl=LinearRegression()

#Y1

pl.fit(p_train1,y_train1)

LinearRegression()

```
print(mean_squared_error(pl.predict(p_train1),y_train1))
print(r2_score(pl.predict(p_train1),y_train1))
```

```
0.2049080907468625
```

```
0.9980484869949252
```

```
#test Y1
```

```
print(mean_squared_error(pl.predict(p_test1),y_test1))
print(r2_score(pl.predict(p_test1),y_test1))
```

```
0.2260660348312774
```

```
0.997520632841922
```

```
#for Y2
```

```
pl.fit(p_train2,y_train2)
```

```
LinearRegression()
```

```
print(mean_squared_error(pl.predict(p_train2),y_train2))
print(r2_score(pl.predict(p_train2),y_train2))
```

```
2.235408259356072
```

```
0.9742763228854109
```

```
#test
```

```
print(mean_squared_error(pl.predict(p_test2),y_test2))
print(r2_score(pl.predict(p_test2),y_test2))
```

```
3.120367099067735
```

```
0.9669879319567856
```

```
pred_y1_poly=pl.predict(p_test1)
```

```
pred_y2_poly=pl.predict(p_test2)
```

```
pd.DataFrame(pred_y1_poly).to_csv('Y1_poly_test.csv')
```

```
pd.DataFrame(pred_y2_poly).to_csv('Y2_poly_test.csv')
```

```
print(pd.DataFrame(pred_y1_poly))
```

```


      0
0    31.190259
1    38.146196
2    11.303890
3    19.158650
4    12.639415
...
226  31.964788
227  15.957342
```




```
228 15.952713
229 34.129987
230 17.391479
```

```
[231 rows x 1 columns]
```

```
pd.DataFrame(pred_y1_poly).describe()
```

	0 
count	231.000000
mean	23.929511
std	9.008340
min	11.291511
25%	15.724074
50%	20.416214
75%	31.972668
max	43.352808

```
pd.DataFrame(pred_y2_poly).describe()
```

	0 
count	231.000000
mean	25.325791
std	9.743356
min	11.501056
25%	15.922862
50%	25.799237
75%	34.038997
max	44.127160

▼ Decision tree

```
from sklearn.tree import DecisionTreeRegressor
dt=DecisionTreeRegressor(max_depth=6,min_samples_leaf=10)
```

```
#for Y1
dt.fit(x_train1,y_train1)

DecisionTreeRegressor(max_depth=6, min_samples_leaf=10)

print(mean_squared_error(dt.predict(x_train1),y_train1))
print(r2_score(dt.predict(x_train1),y_train1))

0.9596512762402747
0.9907942588795297

#test Y1
print(mean_squared_error(dt.predict(x_test1),y_test1))
print(r2_score(dt.predict(x_test1),y_test1))

1.1434648603929145
0.987334695117118

pd.DataFrame(dt.predict(x_test1)).to_csv("Y1_tree_test.csv")

#for y2
dt.fit(x_train2,y_train2)

DecisionTreeRegressor(max_depth=6, min_samples_leaf=10)

print(mean_squared_error(dt.predict(x_train2),y_train2))
print(r2_score(dt.predict(x_train2),y_train2))

2.6026756609094237
0.969922925717513

#test
print(mean_squared_error(dt.predict(x_test2),y_test2))
print(r2_score(dt.predict(x_test2),y_test2))

4.14055960855955
0.9560717197670364

pd.DataFrame(dt.predict(x_test2)).to_csv("Y2_tree_test.csv")

pd.DataFrame(dt.predict(x_test2)).describe()
```

0



count	231.000000
mean	25.257005
std	9.729701
min	11.670833
25%	15.418333

▼ Random Forest

```

from sklearn.ensemble import RandomForestRegressor
rf=RandomForestRegressor()

#y1
rf.fit(x_train1,y_train1)

RandomForestRegressor()

print(mean_squared_error(rf.predict(x_train1),y_train1))
print(r2_score(rf.predict(x_train1),y_train1))

0.0405533506331478
0.9996139890006981

#test y1
print(mean_squared_error(rf.predict(x_test1),y_test1))
print(r2_score(rf.predict(x_test1),y_test1))

0.23501819467532545
0.997420014221337

pd.DataFrame(rf.predict(x_test1)).to_csv("Y1_rf_test.csv")

#for y2
rf.fit(x_train2,y_train2)

RandomForestRegressor()

print(mean_squared_error(rf.predict(x_train2),y_train2))
print(r2_score(rf.predict(x_train2),y_train2))

0.48404653441340595
0.9944539383072616


```

```
#test
print(mean_squared_error(rf.predict(x_test2),y_test2))
print(r2_score(rf.predict(x_test2),y_test2))

3.7197474461038844
0.9607877830338429

pd.DataFrame(rf.predict(x_test2)).to_csv("Y2_rf_test.csv")

pd.DataFrame(rf.predict(x_test2)).describe()
```

	0 
count	231.000000
mean	25.294546
std	9.760860
min	11.433200
25%	15.571000
50%	25.726400
75%	33.670000
max	44.121600

▼ SVR

```
from sklearn.svm import SVR
svr=SVR(kernel='linear')

#for Y1
svr.fit(x_train1,y_train1)

SVR(kernel='linear')

print(mean_squared_error(svr.predict(x_train1),y_train1))
print(r2_score(svr.predict(x_train1),y_train1))

9.207474596274805
0.8990760384149301

#test Y1
```

```
print(mean_squared_error(svr.predict(x_test1),y_test1))  
print(r2_score(svr.predict(x_test1),y_test1))
```

```
6.47290455890255
```

```
0.9223650801780476
```

✓ 1s completed at 17:30

