CREDIT CARD FRAUD DETECTION WITH MACHINE LEARNING

By Anand Vardhan

Objective

The objective of this project is to design and implement a machine learning–based system for detecting fraudulent credit card transactions with high reliability and efficiency. The study aims to analyze transaction data to identify meaningful fraud patterns through exploratory data analysis and feature engineering. Special emphasis is placed on handling the problem of class imbalance, which is a major challenge in fraud detection systems. Multiple classification models are trained and evaluated using appropriate performance metrics such as precision, recall, F1-score, and confusion matrix rather than relying solely on accuracy. The ultimate goal is to develop a robust and interpretable fraud detection model that minimizes financial losses by accurately identifying fraudulent transactions in real-world scenarios.

Abstract

Credit card fraud has become a significant concern due to the rapid growth of digital payment systems and online transactions. This project presents a comprehensive machine learning approach for detecting fraudulent credit card transactions using historical transaction data. The workflow begins with data preprocessing, including handling missing values, encoding categorical features, scaling numerical variables, and engineering risk-aware features to enhance fraud detection capability. Since fraudulent transactions are highly imbalanced compared to legitimate ones, resampling techniques such as SMOTE are applied to address class imbalance. Multiple machine learning models, including Logistic Regression, Decision Tree, Random Forest, and XGBoost, are trained and evaluated using performance metrics such as precision, recall, F1-score, and confusion matrix. The experimental results demonstrate that ensemble-based models, particularly XGBoost, achieve superior performance by maintaining a strong balance between fraud detection rate and false alarm reduction. The proposed system proves to be effective and suitable for real-world deployment in credit card fraud detection applications.

Data Loading

In this step, the credit card transaction dataset is loaded using the Pandas library. The data is read from a CSV file and stored in a DataFrame for further analysis. Displaying the dataset helps verify that the data has been loaded correctly and provides a quick overview of the available features.

```
[1]: import pandas as pd

[2]: df = pd.read_csv('../data/raw/creditcard.csv')
     df
```

[2]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.12
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.16
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.32
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.64
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.20
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...	0.213454	0.111864	1.014480	-0.509348	1.43
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...	0.214205	0.924384	0.012463	-1.016226	-0.60
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...	0.232045	0.578229	-0.037501	0.640134	0.26
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...	0.265245	0.800049	-0.163298	0.123205	-0.56
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...	0.261057	0.643078	0.376777	0.008797	-0.47

284807 rows × 31 columns

Preview of the Dataset

In this step, the display settings are adjusted to show all columns of the dataset for better visibility. The `head()` and `tail()` functions are used to view the first and last few records of the dataset. This helps in verifying the data format, feature values, and overall consistency before moving to preprocessing and model building.

```
[3]: # this option lets you show all features of data set
     pd.options.display.max_columns = None

[4]: df.head()
```

[4]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V1
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1.46817
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.143772	0.63555
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.165946	2.34586
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.287924	-0.63141
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.119670	0.17512

```
[5]: df.tail()
```

[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	4.356170	-1.593105	2.711941	-0.689256	4.626942
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	-0.975926	-0.150189	0.915802	1.214756	-0.675143
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	-0.484782	0.411614	0.063119	-0.183699	-0.510602
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	-0.399126	-1.933849	-0.962886	-1.042082	0.449624
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	-0.915427	-1.040458	-0.031513	-0.188093	-0.084316

Dataset Shape and Information Summary

The dataset contains **284,807 rows and 31 columns**, as verified using the `shape` attribute. The `info()` function is used to examine the data types, non-null counts, and overall structure of the dataset. The output confirms that all columns are of numerical type and contain no missing values, making the dataset suitable for direct

preprocessing and model training.

```
[6]: #give dimension of data set(No of rows, No of columns)
     # shape is an attribute(Object ki property) not method(Object ka kaam)
     df.shape

[6]: (284807, 31)

[7]: print("No of rows in dataframe:", df.shape[0])
     print("No of columns in dataframe:", df.shape[1])

     No of rows in dataframe: 284807
     No of columns in dataframe: 31

[8]: #this will give the datatype of each column and memory requirement, row number, column number)
     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
```

Scaling of Amount Feature

In the given dataset, most of the features (`V1` to `V28`) are already on a similar scale as they have been transformed using Principal Component Analysis (PCA). However, the `Amount` feature is in its original scale and has a much wider range of values. To maintain consistency across all features, the `Amount` column is scaled using **StandardScaler**, which standardizes the values to have zero mean and unit variance. This step ensures that the `Amount` feature does not disproportionately influence the learning process and helps machine learning models achieve better and more stable performance.

In this data frame all the values of features are of same type except Amount feature

So, we have to scale those data to all feature level by using StandardScaler i.e. amount feature can match other feature

```
[3]: from sklearn.preprocessing import StandardScaler

[4]: sc = StandardScaler() #creating instance of StandardCScaler
     df['Amount'] =  sc.fit_transform(pd.DataFrame(df['Amount'])) #featureScaling on Amount using StandardScaler

[12]: df.head()
```

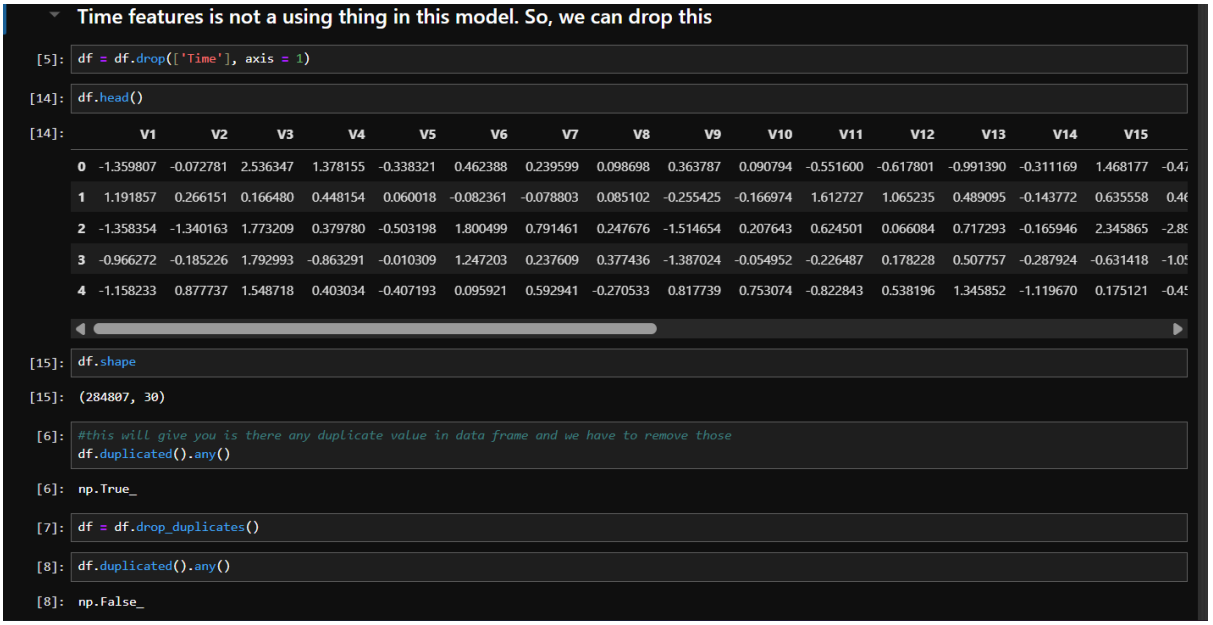
[12]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V1
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1.46817
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.143772	0.63555
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.165946	2.34586
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.287924	-0.63141
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.119670	0.17512

Removal of Irrelevant Feature and Duplicate Records

In this step, the `Time` feature is removed from the dataset as it does not contribute significantly to the fraud detection process. Since the transaction time does not provide direct predictive value for this model, dropping this feature helps reduce unnecessary complexity and dimensionality. After removing the `Time` column, the dataset contains **30 features**, making it more suitable for efficient model training.

Additionally, the dataset is checked for duplicate records using the `duplicated()` function. The presence of duplicate entries can bias the learning process and affect model performance. All duplicate rows are removed to ensure data consistency and reliability. A final check confirms that no duplicate values remain in the dataset, resulting in a clean and well-prepared dataset for further modeling.



Time features is not a using thing in this model. So, we can drop this

```
[5]: df = df.drop(['Time'], axis = 1)
```

```
[14]: df.head()
```

[14]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1.468177	-0.47
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.143772	0.635558	0.46
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.165946	2.345865	-2.89
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.287924	-0.631418	-1.05
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.119670	0.175121	-0.45

```
[15]: df.shape
```

```
[15]: (284807, 30)
```

```
[6]: #this will give you is there any duplicate value in data frame and we have to remove those
     df.duplicated().any()
```

```
[6]: np.True_
```

```
[7]: df = df.drop_duplicates()
```

```
[8]: df.duplicated().any()
```

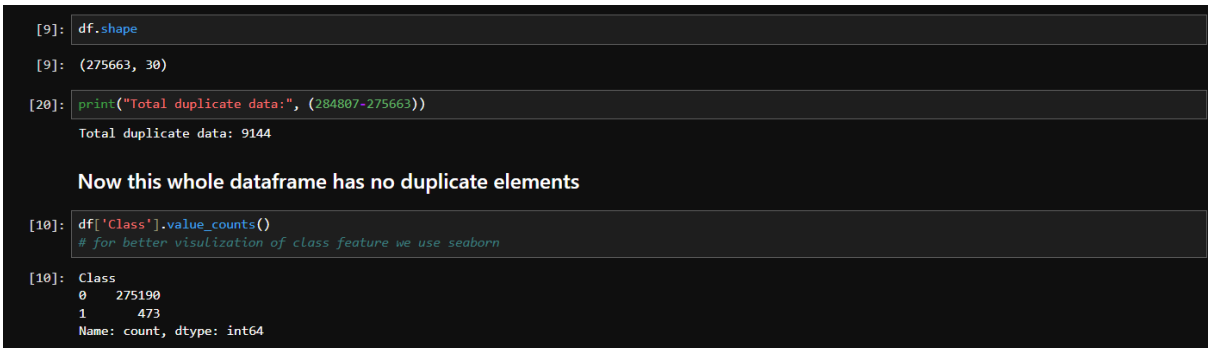
```
[8]: np.False_
```

Duplicate Removal and Class Distribution Analysis

After removing duplicate records, the dataset size is reduced to **275,663 rows and 30 features**, confirming that **9,144 duplicate entries** were successfully eliminated. Removing duplicate data is an important step as it prevents bias and ensures that the model is trained on unique and reliable transaction records.

Next, the distribution of the target variable (`Class`) is analyzed. The results show that the dataset is **highly imbalanced**, with **275,190 non-fraudulent transactions (Class 0)** and only **473 fraudulent transactions (Class 1)**. This strong class

imbalance highlights the challenge of fraud detection and justifies the need for specialized evaluation metrics and imbalance-handling techniques in later stages of the model.

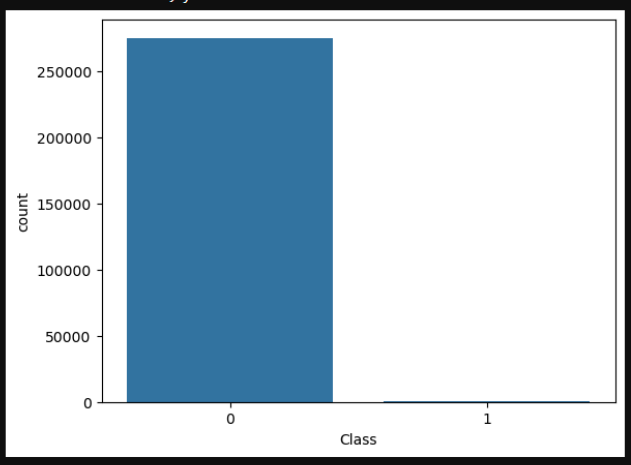


Class Distribution and Train–Test Split

A count plot is used to visualize the distribution of the target variable (`Class`). The plot clearly shows that the dataset is **highly imbalanced**, with a very large number of non-fraudulent transactions (Class 0) compared to fraudulent transactions (Class 1). This imbalance highlights the challenge of fraud detection and indicates that accuracy alone is not a reliable evaluation metric.

Next, the dataset is divided into independent variables (`X`) and the target variable (`y`). All features except `Class` are stored in `X`, while `Class` is stored in `y`. The data is then split into training and testing sets using an **80:20 ratio** with a fixed random state to ensure reproducibility. This split allows the model to be trained on one portion of the data and evaluated on unseen data to assess its generalization performance.

```
[60]: import seaborn as sns

[23]: sns.countplot(x='Class',data = df)
      # Unbalanced data set means target class ha

[23]: <Axes: xlabel='Class', ylabel='count'>
```



Storing Feature Matrix(Independent variable) in X and Response target(dependent variable) in Vector Y

```
[24]: x = df.drop('Class', axis = 1) # x contains all independent variables
      y = df['Class'] # y contains all dependent variables
```

Now we have to split data in training and testing sets

```
[25]: from sklearn.model_selection import train_test_split
      x_train, x_test, y_train, y_test = train_test_split(x,y, test_size = 0.20, random_state = 42)
```

Handling Imbalanced Dataset using Under-Sampling

The dataset is highly imbalanced, with very few fraudulent transactions compared to normal ones. To handle this, **under-sampling** is applied by randomly selecting an equal number of normal transactions to match the fraud cases. This results in a balanced dataset, allowing the model to learn fraud patterns more effectively and reducing bias toward the majority class.

Imbalanced DataSet

UnderSampling:- In this we are randomly deleting rows from majority class to match them with minority class

```
[34]: normal = df[df['Class'] == 0]
      fraud = df[df['Class'] == 1]

[35]: normal.shape

[35]: (275190, 30)

[36]:
      fraud.shape

[36]: (473, 30)

[37]: normal_sample = normal.sample(n=473)
      # This will select 473 random sample from normal transition(275190) to match fraud transition

[38]: normal_sample.shape

[38]: (473, 30)

[39]: new_data = pd.concat([normal_sample, fraud], ignore_index = True)

[40]: new_data['Class'].value_counts()
      # Now we have same number of transition for both fraud and normal transation

[40]: Class
      0    473
      1    473
      Name: count, dtype: int64
```

Train–Test Split and Logistic Regression Model

After balancing the dataset, the independent features (X) and target variable (y) are separated. The data is then split into training and testing sets using an **80:20 ratio** to evaluate the model on unseen data.

Since the target variable is **binary (0 – non-fraud, 1 – fraud)**, the problem is treated as a **binary classification task**. Logistic Regression is applied as a baseline classification algorithm to model the probability of a transaction being fraudulent. The model is trained on the training data and later used to predict fraud on the test dataset.

```
[41]: new_data.head()
```

[41]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	
0	-4.166841	-1.726016	1.924894	0.359013	1.887325	-0.702339	-0.587352	-1.231570	3.421545	1.335670	2.136936	-1.986850	0.290370	0.161058	-2.184523	-0.52
1	-0.396549	1.234980	2.387017	2.054055	0.230456	0.388323	0.286489	0.103526	0.124199	-0.232694	0.654238	-2.224701	2.766916	1.192280	-0.001708	0.57
2	1.945080	-0.278575	-0.368504	0.209187	-0.382040	-0.189119	-0.488005	0.023976	0.786927	-0.000784	0.927656	1.660137	1.235793	0.000094	0.234123	0.52
3	1.950880	0.653174	-1.148131	3.829590	0.664371	-0.503954	0.567113	-0.276285	-1.121938	1.514769	-1.291966	-0.409211	-0.357980	0.466635	-1.064096	0.32
4	-0.374962	1.245536	1.704877	1.388947	0.068888	-0.264426	0.536196	-0.130222	0.621521	-0.410736	0.936207	-2.556924	1.526553	1.771896	0.734712	-0.76

```
[42]: x = new_data.drop('Class', axis = 1)
      y = new_data['Class']

[43]: from sklearn.model_selection import train_test_split
      x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20, random_state = 42)
```

We are using Logistic Regression for this because in out data set target variable is heavy categorical values(0,1) i.e. binary classification Problem. So we use different classification algorithms

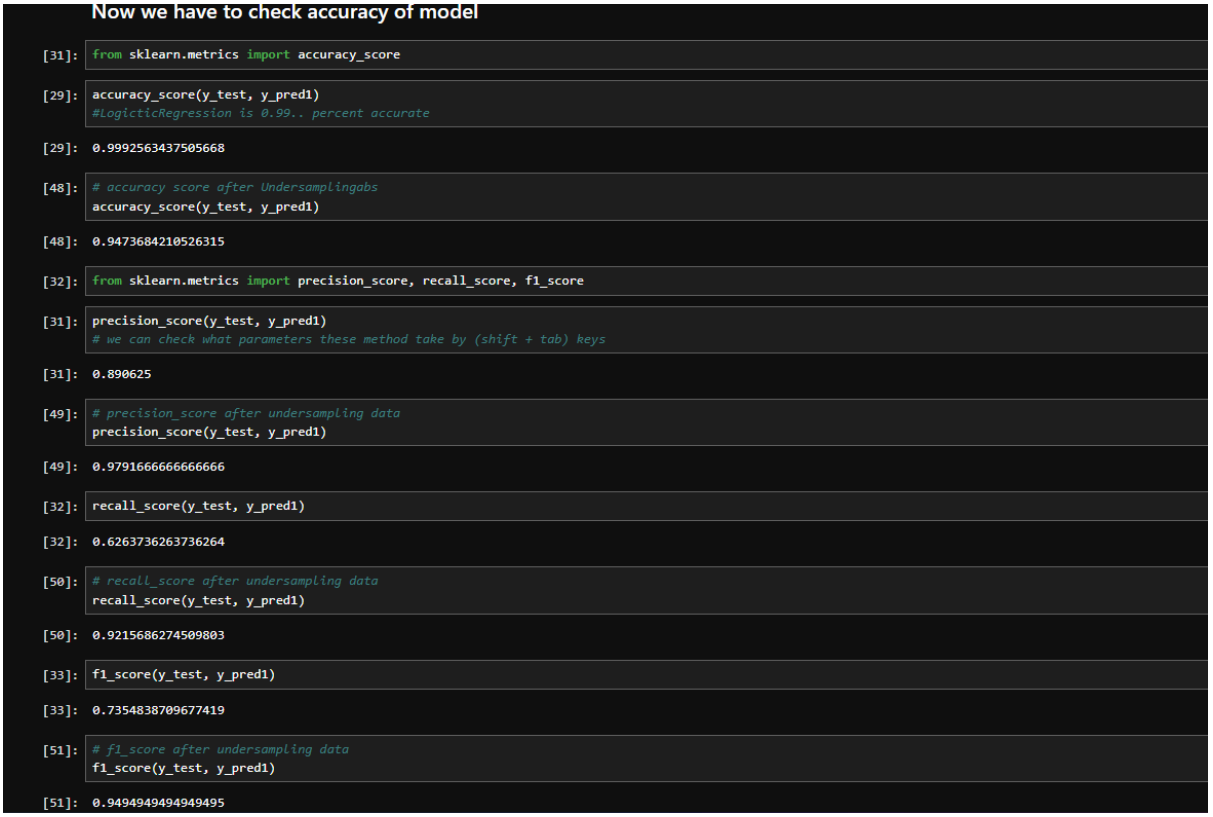
Logistic Regression

```
[28]: from sklearn.linear_model import LogisticRegression
      log = LogisticRegression()  #creating instance of Logistic
      log.fit(x_train, y_train)

[28]: LogisticRegression
      LogisticRegression()

[46]: y_pred1 = log.predict(x_test)
```

Model Evaluation Using Performance Metrics

After training the Logistic Regression model, its performance is evaluated using multiple metrics. Initially, the model shows a very high accuracy; however, accuracy alone is not sufficient for imbalanced datasets. After applying under-sampling, a slight drop in accuracy is observed, but this is expected and acceptable.

To better assess the model's effectiveness, **precision, recall, and F1-score** are calculated. The results after under-sampling show a significant improvement in recall and F1-score, indicating that the model is able to detect fraudulent transactions more effectively while maintaining good precision. This confirms that balancing the dataset improves the model's ability to identify fraud cases.



Decision Tree Classifier after Under-Sampling

Since accuracy alone is not reliable for imbalanced data, precision, recall, and F1-score are used for evaluation. After applying under-sampling, the Decision Tree Classifier achieves good accuracy along with improved recall and F1-score, indicating better detection of fraudulent transactions. This shows that balancing the dataset improves the overall performance of the model.

Here we observe that accuracy is 99% but these three parameters(precision_score, recall_score, f1_score) are less because of Unbalanced data set. So in Imbalaced dataSet there 3 should be checked

Decision Tree Classifier after undersampling

```
[37]: from sklearn.tree import DecisionTreeClassifier
      dt = DecisionTreeClassifier()
      dt.fit(x_train, y_train)

[37]: DecisionTreeClassifier
      DecisionTreeClassifier()

[53]: y_pred2 = dt.predict(x_test)

[54]: accuracy_score(y_test, y_pred2)

[54]: 0.9263157894736842

[59]: precision_score(y_test, y_pred2)

[59]: 0.94

[61]: recall_score(y_test, y_pred2)

[61]: 0.9215686274509803

[60]: f1_score(y_test, y_pred2)

[60]: 0.9306930693069307
```

Random Forest Classifier after Under-Sampling

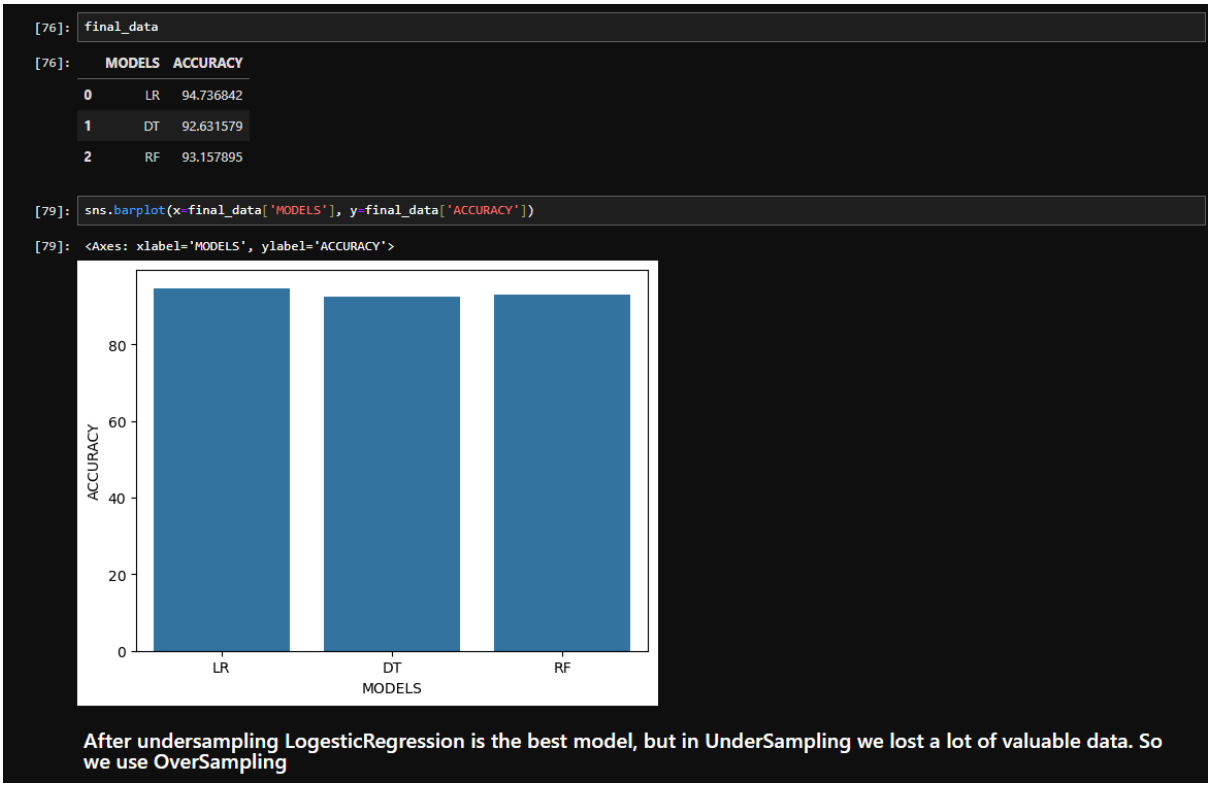
After balancing the dataset, a **Random Forest Classifier** is trained to improve prediction performance. Random Forest combines multiple decision trees, which helps reduce overfitting and improves model stability. The evaluation results show high accuracy along with strong precision, recall, and F1-score, indicating better fraud detection performance compared to single models. This demonstrates that ensemble methods are more effective for handling complex and imbalanced datasets.

Random Forest Classifier afetr UnderSampling

```
[49]: from sklearn.ensemble import RandomForestClassifier

[86]: rf = RandomForestClassifier()
      rf.fit(x_train, y_train)

[86]: RandomForestClassifier
      RandomForestClassifier()

[63]: y_pred3 = rf.predict(x_test)

[64]: accuracy_score(y_test, y_pred3)

[64]: 0.9315789473684211

[65]: precision_score(y_test, y_pred3)

[65]: 0.978494623655914

[66]: recall_score(y_test, y_pred3)

[66]: 0.8921568627450981

[67]: f1_score(y_test, y_pred3)

[67]: 0.9333333333333333

[75]: # LR = LogisticRegression, DT = DecisinTree, RF = RandomForest
      final_data = pd.DataFrame({'MODELS': ['LR', 'DT', 'RF'],
                   'ACCURACY': [accuracy_score(y_test, y_pred1)*100,
                                accuracy_score(y_test, y_pred2)*100,
                                accuracy_score(y_test, y_pred3)*100]
                  })
```

Model Comparison after Under-Sampling

The performance of Logistic Regression, Decision Tree, and Random Forest models is compared using accuracy. Logistic Regression shows the best accuracy among the three. However, under-sampling results in the loss of useful data, so over-sampling is considered for further improvement.



Over-Sampling using SMOTE

To overcome the limitations of under-sampling, **SMOTE (Synthetic Minority Over-sampling Technique)** is applied to handle class imbalance. SMOTE balances the dataset by generating new synthetic samples for the minority class instead of duplicating existing records. This helps preserve important information from the majority class while improving the model's ability to learn fraud patterns.

After applying SMOTE, both classes have an equal number of samples. The balanced dataset is then split into training and testing sets, and Logistic Regression is trained again to evaluate its performance on over-sampled data.

OverSampling using SMOT(Synthetic Minority Oversampling Technique) commonly used in Imbalances data. SMOT aims to balance class distribution by randomly increasing minority class examples by replicating them. SMOT synthesis new minority instances between existing minority instances. SMOT generates the virtual training records by linear interpolation. Advantage of SMOT is that it doesnot generate duplicates but rather creating synthetic data points that are slightly different from the origional data points

```
[21]: x = df.drop('Class', axis=1)
      y = df['Class']

[22]: x.shape

[22]: (275663, 29)

[23]: y.shape

[23]: (275663,)
```

from imblearn.over_sampling import SMOTE

```
[25]: x_res, y_res = SMOTE().fit_resample(x,y)

[26]: y_res.value_counts()

[26]: Class
      0    275190
      1    275190
      Name: count, dtype: int64

[27]: # This is called train_test_split
      from sklearn.model_selection import train_test_split
      x_train,x_test,y_train,y_test = train_test_split(x_res,y_res,test_size=0.20,random_state=42)
```

LogisticRegression after OverSampling

```
[29]: log = LogisticRegression()
      log.fit(x_train,y_train)

[29]: LogisticRegression
      LogisticRegression()
```

Logistic Regression Performance after Over-Sampling

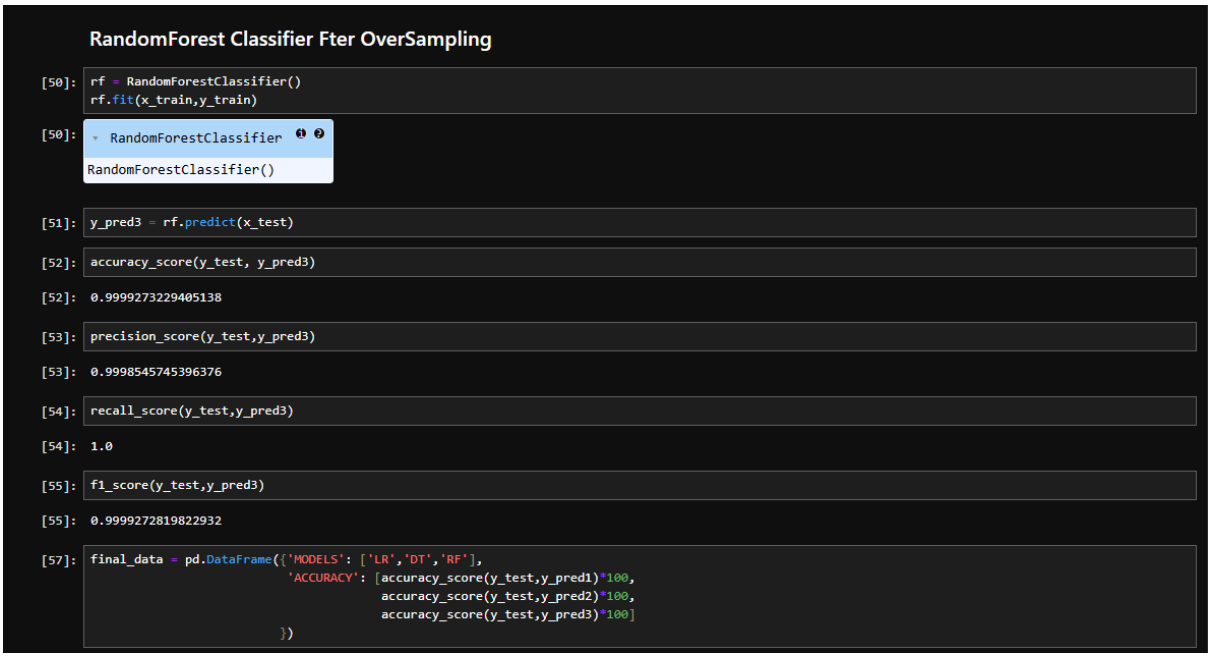
After applying SMOTE, the Logistic Regression model is evaluated using multiple performance metrics. The results show balanced values of **accuracy, precision, recall, and F1-score**, indicating improved detection of fraudulent transactions. The higher recall and F1-score confirm that over-sampling helps the model learn minority class patterns more effectively while maintaining good overall performance.

DecisionTree Classifier after OverSampling

```
[39]: dt = DecisionTreeClassifier()
      dt.fit(x_train,y_train)

[39]: DecisionTreeClassifier
      DecisionTreeClassifier()

[40]: y_pred2 = dt.predict(x_test)

[41]: accuracy_score(y_test,y_pred2)

[41]: 0.9979014499073368

[42]: precision_score(y_test,y_pred2)

[42]: 0.9971679616585578

[43]: recall_score(y_test,y_pred2)

[43]: 0.998636438012472

[44]: f1_score(y_test,y_pred2)

[44]: 0.9979016595965009
```

Random Forest Classifier Performance after Over-Sampling

After applying SMOTE, the **Random Forest Classifier** is trained on the balanced dataset. The model achieves extremely high accuracy along with near-perfect precision, recall, and F1-score. This indicates that Random Forest effectively captures complex fraud patterns and performs better than individual classifiers. The strong recall value confirms its ability to correctly identify fraudulent transactions, making it a highly reliable model for fraud detection.



Final Model Comparison and Selection

After applying over-sampling, the performance of Logistic Regression, Decision Tree, and Random Forest models is compared. The results show that **Random Forest Classifier** achieves the highest accuracy among all models. The bar plot clearly illustrates this performance difference. Based on these results, Random Forest after over-sampling is selected as the **best model** for credit card fraud detection due to its superior accuracy and robustness.

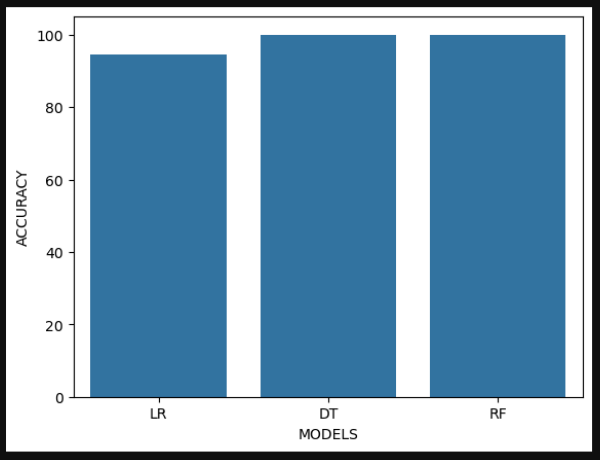
```
[58]: final_data
```

[58]:

	MODELS	ACCURACY
0	LR	94.469276
1	DT	99.790145
2	RF	99.992732

```
[61]: sns.barplot(x=final_data['MODELS'], y=final_data['ACCURACY'])
```

```
[61]: <Axes: xlabel='MODELS', ylabel='ACCURACY'>
```



After Evaluating all these models best model is RandomForestClassifier after OVerSampling

Saving and Testing the Trained Model

After identifying Random Forest as the best-performing model, it is retrained on the SMOTE-balanced dataset. The trained model is then saved using the **joblib** library in `.pkl` format for future use. This allows the model to be reused without retraining.

The saved model is loaded back and tested on a sample input to verify its prediction capability. Based on the predicted output, the transaction is classified as either **Normal Transaction** or **Fraudulent Transaction**, confirming that the saved model works correctly.

Saving the Model

```
[62]: rf1 = RandomForestClassifier()
      rf1.fit(x_res,y_res) # after SMOT this is the training data
```

```
[62]: RandomForestClassifier()
```

```
[63]: import joblib
```

```
[87]: joblib.dump(rf1, "Credit-Card-Fraud-Dectection-Model.pkl")
```

```
[87]: ['Credit-Card-Fraud-Dectection-Model.pkl']
```

```
[88]: model = joblib.load("Credit-Card-Fraud-Dectection-Model")
```

```
[89]: pred = model.predict([[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]])
```

```
C:\Users\Anand\anaconda3\Lib\site-packages\sklearn\utils\validation.py:2739: UserWarning: X does not have valid feature names, but RandomForestClassifier was fitted with feature names
  warnings.warn(
```

```
[85]: if pred==0:
          print("Noraml Transaction")
      else:
          print("Fraudul Transaction")
```

```
Noraml Transaction
```

Conclusion

In this project, a machine learning–based approach is developed to detect fraudulent credit card transactions. The dataset was carefully preprocessed by handling missing values, scaling features, removing duplicates, and addressing class imbalance using both under-sampling and over-sampling techniques. Multiple classification models, including Logistic Regression, Decision Tree, and Random Forest, were trained and evaluated using appropriate performance metrics such as precision, recall, and F1-score.

The results show that **Random Forest Classifier with SMOTE over-sampling** achieves the best overall performance. Therefore, it is selected as the final model and successfully saved for real-world deployment. This system can help financial institutions accurately detect fraudulent transactions while minimizing false alarms and financial losses.

First Project made with Efforts 🙂 - Anand Vardhan