CREDIT CARD FRAUD DETECTION SYSTEM

INTRODUCTION

Credit card fraud is any dishonest act or behaviour to obtain information without proper authorisation from the account holder for financial gain. Among different ways of committing frauds, skimming is the most common one, which is a way of duplicating information that is located on the magnetic strip of the card. Apart from this, following are the other ways:

- Manipulation/alteration of genuine cards
- Creation of counterfeit cards
- Stealing/loss of credit cards
- Fraudulent telemarketing

BUSINESS PROBLEM OVERVIEW

- For many banks, retaining high profitable customers is the number one business goal. Banking fraud, however, poses a significant threat to this goal for different banks. In terms of substantial financial losses, trust and credibility, this is a concerning issue to both banks and customers alike.
- It has been estimated by Nilson Report that by 2020, banking frauds would account for $30 billion worldwide. With the rise in digital payment channels, the number of fraudulent transactions is also increasing in new and different ways.
- In the banking industry, credit card fraud detection using machine learning is not only a trend but a necessity for them to put proactive monitoring and fraud prevention mechanisms in place. Machine learning is helping these institutions to reduce time-consuming manual reviews, costly chargebacks and fees as well as denials of legitimate transactions.

About the Dataset

1

- The data set is taken from the Kaggle website and has a total of 2,84,807 transactions; out of these, 492 are fraudulent. Since the data set is highly imbalanced, it needs to be handled before model building.
- It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase.
- 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.
- It contains only numerical input variables which are the result of a PCA transformation. Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

2 Exploratory Data Analysis

Exploratory Data Analysis

Initially we performed some Exploratory Data Analysis, to understand the data in a brief manner. We observed some columns and observed different features of the data.

```
column_list = (list(df.columns))
print(column_list)
```

```
['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11', 'V12', 'V13',
'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21', 'V22', 'V23', 'V24', 'V25', 'V26',
'V27', 'V28', 'Amount', 'Class']
```

```
# Drop unnecessary columns
df = df.drop("Time", axis = 1)
df.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
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
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
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
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
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

```
 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
Information about the data frame: None
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8
0	0.000000	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698
1	0.000000	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102
2	1.000000	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676
3	1.000000	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436
4	2.000000	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533

Observing the Distribution of Classes.

The first line states that non-fraudulent transactions account for 99.83% of the total transactions. This means that the overwhelming majority of transactions in the dataset are legitimate and not associated with fraudulent activities.

On the other hand, the second line states that fraudulent transactions represent only 0.17% of the total transactions. This is a relatively small percentage, indicating that fraudulent transactions are rare occurrences within the dataset.

The next section provides the shares of normal and fraud transactions as decimal values. The "Normal_share" represents the proportion of non-fraudulent transactions, which is approximately 99.83%. Similarly, the "Fraud_share" represents the proportion of fraudulent transactions, which is approximately 0.17%.

Finally, the "Imbalance Percentage" is calculated as the difference between the two shares, which is 0.173%. This value indicates the degree of imbalance between the two classes in the dataset. In this case, the dataset is highly imbalanced, with the minority class (fraudulent transactions) comprising only a small fraction of the total transactions.

```
Non-Fraudulent : 99.83 %

    Fraudulent : 0.17 %

==============================================

 Normal_share= 99.82725143693798

 Fraud_share= 0.1727485630620034

==============================================

 Imbalance Percentage = 0.173047500131896
```

Splitting the Dataset

We split the dataset into 80:20 and hence based on that we can analyse that the difference between the Full and Test data.

Fraudulent Count for Full Data : 492

Fraudulent Count for Train Data : 394

Fraudulent Count for Test Data : 98

```
# Spltting the into 80:20 train test size
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 42,stratify=y)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((227845, 29), (56962, 29), (227845,), (56962,))
```

```
# Checking the split of the class label
print(" Fraudulent Count for Full data : ",np.sum(y))
print("Fraudulent Count for Train data : ",np.sum(y_train))
print(" Fraudulent Count for Test data : ",np.sum(y_test))
```

```
 Fraudulent Count for Full data :  492

Fraudulent Count for Train data :  394

 Fraudulent Count for Test data :  98
```

Checking for Skewness and Treating It

In data analysis, it is important to check for skewness in variables as it can have implications for statistical modeling, interpretation of results, and the choice of appropriate analysis techniques.

As we can see a lot of our principal components are skewed and hence we'll need to treat it.

```
var = X_train.columns
skew_list = []
for i in var:
    skew_list.append(X_train[i].skew())

tmp = pd.concat([pd.DataFrame(var, columns=["Features"]), pd.DataFrame(skew_list, columns=["Skewness"])], axis=1)
tmp.set_index("Features", inplace=True)
tmp
```

	Skewness
Features	
V1	-3.306334
V2	-4.779484
V3	-2.247962
V4	0.687574
V5	-2.786851
V6	1.937381
V7	3.152665
V8	-8.639485
V9	0.541869
V10	1.132688

Checking for Skewness and Treating It

The preprocessing.PowerTransformer() function is used to transform the data to have a more Gaussian-like distribution.

Both methods (yeo-johnson and box-cox) are commonly used to transform data to make it more normally distributed, which can improve the performance of some machine learning algorithms.

The yeo-johnson method is an extension of the box-cox method that allows for transformation of variables with negative values, while the box-cox method only works with variables that are strictly positive. Therefore, if you have variables with negative values, yeo-johnson would be the appropriate choice.

```
pt= preprocessing.PowerTransformer(method='yeo-johnson', copy=True)  # creates an instance of the PowerTransformer class.
pt.fit(X_train)

X_train_pt = pt.transform(X_train)
X_test_pt = pt.transform(X_test)

y_train_pt = y_train
y_test_pt = y_test
```

```
print(X_train_pt.shape)
print(y_train_pt.shape)
```

```
(227845, 29)

(227845,)
```

Methods Used 3

We will construct models using the algorithms mentioned below and assess them to find the most optimal one. The process of building models with these algorithms is extremely resource-intensive, particularly when dealing with large datasets.

1. Logistic Regression
2. KNN
3. SVM
4. Decision Tree

Why can't we use accuracy for imbalanced dataset?

- **Accuracy is not a good metric** for imbalanced datasets.
- This model would receive a very good accuracy score as it predicted correctly for the majority of observations, but this **hides the true performance** of the model which is objectively not good as it only predicts for one class
- Don't use accuracy score as a metric with imbalanced datasets (will be usually high and misleading), **instead use f1-score, precision/recall score or confusion matrix**

How are we working with Imbalance Data ?

- In **undersampling**, you select fewer data points from the majority class for your model building process to balance both classes.
- In **oversampling**, you assign weights to randomly chosen data points from the minority class. This is done so that the algorithm can focus on this class while optimising the loss function.
- **SMOTE** is a process using which you can generate new data points that lie vectorially between two data points that belong to the minority class.
- **ADASYN** is similar to SMOTE, with a minor change in the sense that the number of synthetic samples that it will add will have a density distribution. The aim here is to create synthetic data for minority examples that are harder to learn rather than the easier ones.

SVM CLASSIFIER

1. A. Finidng a suitable range for a Single Hyperparameters for narrowing the range of parameters using visualization
2. Multiple Hyperparameter tuning (GridSearchCV) + Best Model ROC_AUC Score

Perform class balancing with :

I. Random Oversampling

II. SMOTE (Synthetic Minority Over-sampling Technique)

III. ADASYN (ADASYN (Adaptive Synthetic)

Type of OverSampling	Model	Parameter	ROC-AUC Score	F1-Score	Precision	Recall
None	svm.SVC	{'C': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'probability': True}	0.9701114654	0.8121827411	0.8080808081	0.8163265306
ROS	svm.SVC	{'C': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'probability': True}	NA	NA	NA	NA
SMOTE	svm.SVC	{'C': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'probability': True}	NA	NA	NA	NA
ADASYN	svm.SVC	{'C': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'probability': True}	NA	NA	NA	NA

R
A
N
D
O
M

F
O
R
E
S
T

- **ROC-AUC Curve :**

It ranges from 0 to 1, where a score of 0.5 represents a random classifier, and a score of 1 represents a perfect classifier.

Our Model Average Value : 0.96 : A high ROC-AUC score suggests that the model has good predictive power and can effectively separate the classes

F1 SCORE

Formula used : F1 score = 2 * (precision * recall) / (precision + recall)

Precision is the ratio of true positive predictions to the total number of positive predictions. It measures how well the model correctly identifies positive instances.

Recall, also known as sensitivity or true positive rate, is the ratio of true positive predictions to the total number of actual positive instances in the dataset. It measures how well the model captures all positive instances.

Random Forest

Type of OverSampling	Model	Parameter	ROC-AUC Score	F1-Score	Precision	Recall
None	RandomForestClassifier	{'min_samples_split': 5, 'n_estimators': 500}	0.9623530686894904	0.8282828283	0.82	0.83673469
ROS	RandomForestClassifier	{'min_samples_split': 5, 'n_estimators': 500}	0.9634288542372493	0.8258706468	0.8058252427	0.84693877
SMOTE	RandomForestClassifier	{'min_samples_split': 5, 'n_estimators': 500}	0.9788061953689164	0.8258706468	0.8058252427	0.84693877
ADASYN	RandomForestClassifier	{'min_samples_split': 5, 'n_estimators': 500}	0.9634288542372493	0.8258706468	0.8058252427	0.84693877

Logistic Regression

Finding a suitable range for a Single Hyperparameters for narrowing the range of parameters using visualization.

We're seeing a nan value for the score when using the GridSearchCV function in scikit-learn, it typically indicates that the model did not converge or encountered some numerical instability during the training process.

This can happen for a variety of reasons, such as

- ❏ an insufficient number of iterations,
- ❏ a learning rate that's too high or too low,
- ❏ a dataset that has features with a wide range of values.

1. LogisticRegression

Type of OverSampling	Model	Parameter	ROC-AUC Score	F1-Score	Precision	Recall
None	LogisticRegression	{'C': 0.01, 'penalty': 'l2'}	0.9752271442	0.5977011494	0.4785276074	0.7959183673
ROS	LogisticRegression	{'C': 4, 'penalty': 'l2'}	0.9714047245	0.9320171419	0.925193644	0.9389420371
SMOTE	LogisticRegression	{'C': 4, 'penalty': 'l2'}	0.9698314202	0.9210604137	0.9111856823	0.9311515194
ADASYN	LogisticRegression	{'C': 4, 'penalty': 'l2'}	0.9173743861	0.8375936925	0.842044465	0.8331897234

LogisticRegression {'C': 0.01, 'penalty': 'l2'} =

Best Mean ROC-AUC score for val data: 0.9797969874466093
Mean precision val score for best C: 0.885478588591554
Mean recall val score for best C: 0.6295975017349064
Mean f1 val score for best C: 0.7341406860856002

Hence, we can achieve ROC-AUC score of 97% which is a good accuracy for this model

KNN Classifier

- Finding a suitable range for a Single Hyperparameters for narrowing the range of parameters using visualization.
- Multiple Hyperparameter tuning (GridSearchCV) + Best Model ROC_AUC Score
- Euclidean distance is a good choice for problems where the variables have similar importance and are measured in the same scale. On the other hand, Manhattan distance is a good choice when variables have different scales or when you want to penalize differences in some variables more heavily than others.

2. KNeighborsClassifier

Type of OverSampling	Model	Parameter	ROC-AUC Score	F1-Score	Precision	Recall
None	KNeighborsClassifier	{'metric': 'manhattan', 'n_neighbors': 9}	0.9385655571	0.8248587571	0.9240506329	0.744897959
ROS	KNeighborsClassifier	{'n_neighbors': 9}	0.9398546706	0.9241250283	0.9986317595	0.859964124
SMOTE	KNeighborsClassifier	{'metric': 'manhattan', 'n_neighbors': 9}	0.9520626163	0.9379643837	0.9952832981	0.886888013
ADASYN	KNeighborsClassifier	{'n_neighbors': 9}	0.8685620475	0.82343481	0.9950914437	0.702287794

KNeighborsClassifier {'metric': 'manhattan', 'n_neighbors': 9} =

0.9274613536399045

Result

Based on above observations we found following result:

- Logistic Regression:
  - ROC-AUC Score: 0.9174
  - F1-Score: 0.8376
  - Precision: 0.8420
  - Recall: 0.8332
- K-Nearest Neighbors Classifier:
  - ROC-AUC Score: 0.8686
  - F1-Score: 0.8234
  - Precision: 0.9951
  - Recall: 0.7023
- Support Vector Machine (SVM) Classifier:
  - Not computed due to very large training time

Result(cont...)

- Decision Tree Classifier:
  ROC-AUC Score: 0.8812
  F1-Score: 0.8004
  Precision: 0.8987
  Recall: 0.7215

- Random Forest Classifier:
  ROC-AUC Score: 0.9634
  F1-Score: 0.8259
  Precision: 0.8058
  Recall: 0.8469
  It is important to note that the SVM classifier's performance could not be computed due to the excessive training time.

Result(cont....)

Among the evaluated models, the Random Forest Classifier achieved the highest ROC-AUC score, indicating its strong ability to discriminate between the classes. The K-Nearest Neighbors Classifier demonstrated high precision but relatively lower recall, suggesting that it may be effective in identifying positive instances but could miss some relevant cases. The Logistic Regression and Decision Tree classifiers showed balanced performance across multiple metrics.

Considering these results, further analysis and fine-tuning of the models can be conducted to enhance their performance.

Conclusion

- F1 Score gives threshold of "0.9880" and gives a Precision of 99.42%
- While there are areas that require improvement, we have a clear understanding of the steps needed to enhance our work.
- By addressing these shortcomings and focusing on data exploration, skewness mitigation, stratified train-test split, class imbalance handling, model hyperparameter tuning, appropriate model evaluation, and code readability, we can achieve more comprehensive and impactful results.
- With diligence and a positive mindset, we will elevate the quality of our work and move closer to our desired outcomes.
- Considering the limited information provided, we are unable to provide a comprehensive result and conclusion at this time.

THANKS!