Credit Card Fraud Detection

by Pratiek Sonare

1. Overview

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. With the rise of online transactions, the volume of data generated in the payment systems has skyrocketed, providing both opportunities and challenges for detecting fraudulent activity. The primary goal of a credit card fraud detection piepline is accurately identify fraudulent transactions in real-time while minimizing false positives (legitimate transactions flagged as fraud).

2. Dataset

I've used the Kaggle Dataset - Credit Card Fraud Detection for this project.

- 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. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data.

- 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.

---

3. Tools and Libraries

The following tools and Python libraries were used:

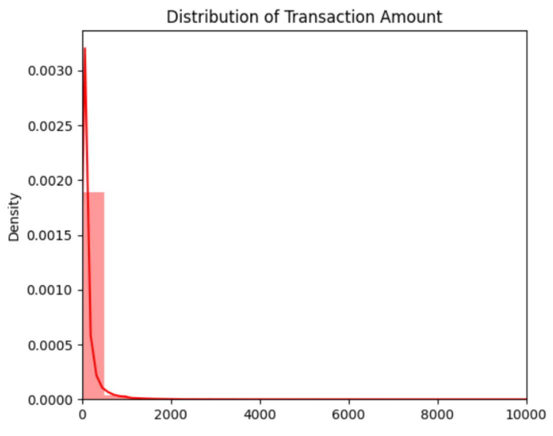
- **Python**: Programming language for data processing and implementation.
- **Pandas**: For data manipulation and preprocessing.
- **Numpy**: For numerical computations.
- **Scikit-learn**: For implementing count vectorization and cosine similarity.

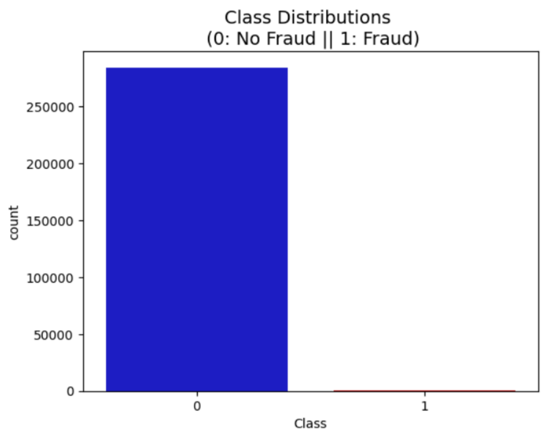
---

4. Methodology

Step 1: Data Analysis

- Analyzing the dataset
  - Conclusion: the given dataset is highly **skewed**, there is a need to normalize the dataset before feeding it to the pipeline.





- Handling Unbalanced Datasets - Outlier removal and sampling of data
    - Scaling data -

```
from sklearn.preprocessing import RobustScaler

rob_scaler = RobustScaler()
scaled_values = rob_scaler.fit_transform(df[['Amount',

df[['scaled_amount', 'scaled_time']] = scaled_values
df.drop(['Amount', 'Time'], axis=1, inplace=True)

df.head()
```

- Extreme Outlier Removal -

```
import numpy as np

def remove_outliers(df, feature_column, label_column='C
    """
    Remove outliers for a given feature based on IQR (I
    Parameters:
    - df: The dataframe containing the data.
    - feature_column: The column name for which to remo
    - label_column: The column representing the labels

    Returns:
    - df: The dataframe with outliers removed for the g
    """

    feature_fraud = df[feature_column].loc[df[label_col

    q25, q75 = np.percentile(feature_fraud, 25), np.per
    iqr = q75 - q25

    cut_off = iqr * 1.5
    lower, upper = q25 - cut_off, q75 + cut_off

    print(f'{feature_column} Lower: {lower}')
    print(f'{feature_column} Upper: {upper}')

    outliers = [x for x in feature_fraud if x < lower o
    print(f'{feature_column} outliers: {outliers}')
    print(f'Feature {feature_column} Outliers for Fraud

    df = df.drop(df[(df[feature_column] > upper) | (df[

    print(f'Number of Instances after outliers removal:
    print('----' * 30)

    return df
```

```
# Removing outliers for V14
new_df = remove_outliers(new_df, 'V14')

# Removing outliers for V12
new_df = remove_outliers(new_df, 'V12')

# Removing outliers for V10
new_df = remove_outliers(new_df, 'V10')
```

Step 2: Under-Sampling Data

Reduce the size of dataset, by eliminating the majority class to the same count as minority class.

>> Balance the dataset

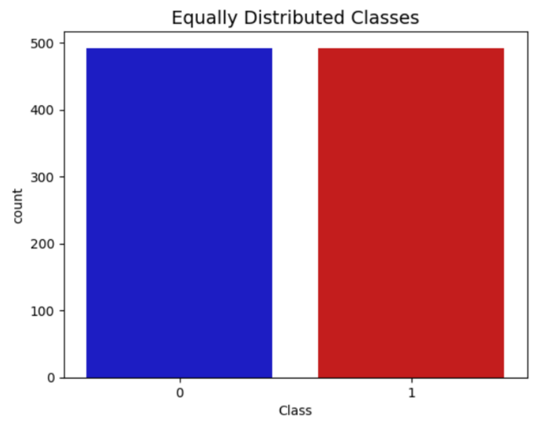
>> Prevent overfitting

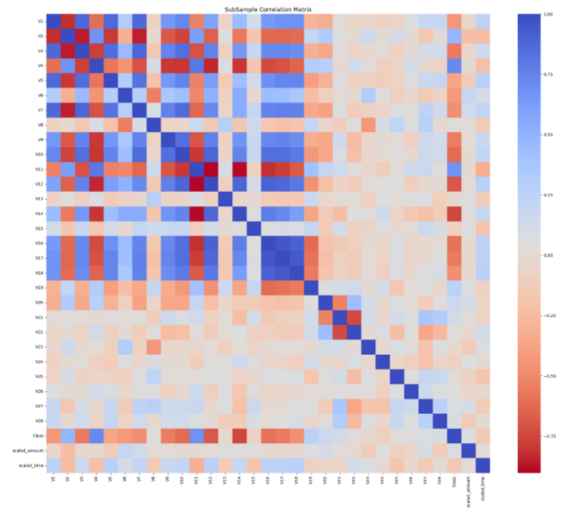
>> "Loss" of data

1. Random Undersampling

- Shuffle data randomly
- Reduce the dataset to equal number of Fraud / Non-Fraud instances

```
#shuffle
df = df.sample(frac=1)

fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0][:492]

#first fraud rows, next non-fraud rows
normal_distributed_df = pd.concat([fraud_df, non_fraud_
#shuffle again
new_df = normal_distributed_df.sample(frac=1, random_st
new_df.head()
```





- Correlation Matrix showcases:
  - **Negative Correlations:** V17, V14, V12 and V10 are negatively correlated. Notice how the lower these values are, the more likely the end result will be a fraud transaction.
  - **Positive Correlations:** V2, V4, V11, and V19 are positively correlated. Notice how the higher these values are, the more likely the end result will be a fraud transaction.

2. NearMiss Technique

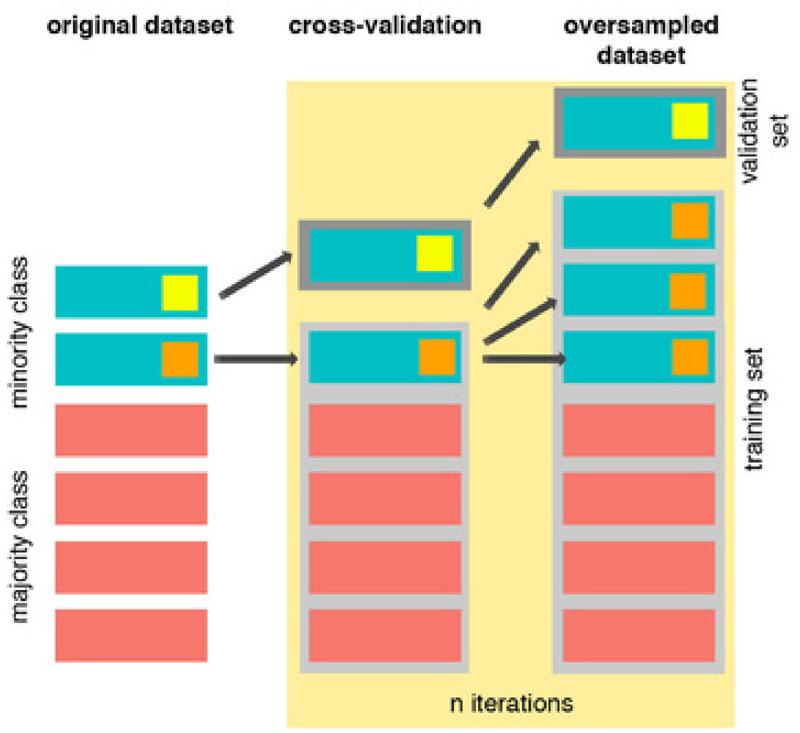
Using imblearn.under_sampling.NearMiss technique to undersample data.

How does NearMiss technique work?

NearMiss selects a subset of majority class samples that are closest to the minority class, aiming to retain as much information as possible while achieving a balance between the classes

1. **Distance Calculation**:
    - For each instance in the minority class (fraudulent transactions), calculate the distance to every instance in the majority class (legitimate transactions).
2. **Selection Criteria**:
   NearMiss has different variants, each defining a specific rule for selecting majority class instances based on their proximity to the minority class:
    - **NearMiss-1**: Selects majority class instances that are closest (in terms of distance) to the minority class instances.
    - **NearMiss-2**: In this variant, the majority class instances that are farthest from the minority class are removed.
    - **NearMiss-3**: Combines aspects of both the previous variants by selecting a subset of majority class instances that are close to the minority class, but also considering the distance relationships among the majority class samples themselves.

Step 3: SMOTE Over-Sampling Data

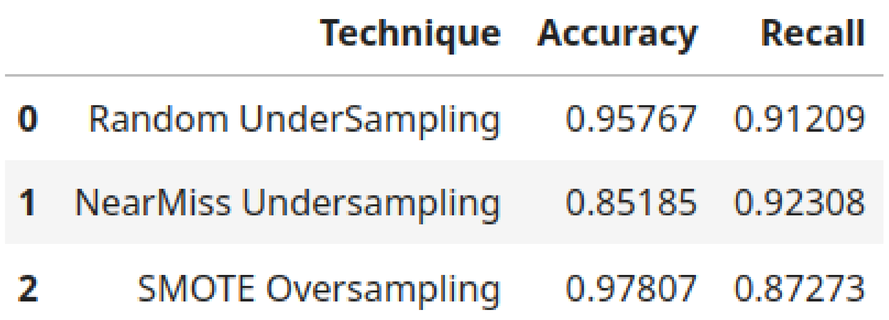


- Implementing pipeline with SMOTE and logistic regression fitting

```
from sklearn.metrics import accuracy_score, precision_scor
from sklearn.model_selection import StratifiedShuffleSplit
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import make_pipeline

# Set up your cross-validation strategy
sss = StratifiedShuffleSplit(n_splits=5, test_size=0.3, ra

# Initialize lists to store metrics
accuracy_lst = []
precision_lst = []
recall_lst = []
f1_lst = []
auc_lst = []
```

```
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.
log_reg = LogisticRegression()

log_reg_sm = GridSearchCV(log_reg, log_reg_params)

# Loop through cross-validation splits
for train, test in sss.split(X_train, y_train):
    # Create pipeline with SMOTE oversampling
    pipeline = make_pipeline(SMOTE(sampling_strategy='mino

    # Fit the model on training data (SMOTE will oversampl
    model = pipeline.fit(X_train[train], y_train[train])

    # Get the best model after grid search
    log_reg_smbest = log_reg_sm.best_estimator_

    # Predict on the test data (no resampling applied to t
    prediction = log_reg_smbest.predict(X_train[test])

    # Store evaluation metrics
    accuracy_lst.append(accuracy_score(y_train[test], pred
    precision_lst.append(precision_score(y_train[test], pr
    recall_lst.append(recall_score(y_train[test], predicti
    f1_lst.append(f1_score(y_train[test], prediction))
    auc_lst.append(roc_auc_score(y_train[test], prediction

# Print cross-validation results
print('Cross Val Results: ')
print('---' * 45)
print("accuracy: {}".format(np.mean(accuracy_lst)))
print("precision: {}".format(np.mean(precision_lst)))
print("recall: {}".format(np.mean(recall_lst)))
print("f1: {}".format(np.mean(f1_lst)))
print('---' * 45)
```

Step 4: Evaluating Pipelines



	Technique	Accuracy	Recall
0	Random UnderSampling	0.95767	0.91209
1	NearMiss Undersampling	0.85185	0.92308
2	SMOTE Oversampling	0.97807	0.87273