VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

LAM NGOC HAI – 518H0347

PHAN AN DUY - 518H0616

CREDIT CARD FRAUD DETECTION WITH IMBALANCED DATA

INFORMATION TECHNOLOGY PROJECT

2

COMPUTER SCIENCE

HO CHI MINH , YEAR 2025

VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

LAM NGOC HAI – 518H0347

PHAN AN DUY - 518H0616

CREDIT CARD FRAUD DETECTION WITH IMBALANCED DATA

INFORMATION TECHNOLOGY PROJECT 2

COMPUTER SCIENCE

Advised by

Mr., Trinh Hung Cuong

HO CHI MINH , YEAR 2025

ACKNOWLEDGMENT

I would like to express my heartfelt gratitude to Mr. Trinh Hung Cuong for his invaluable guidance and support throughout this project. His insights and encouragement have been instrumental in shaping this work, and I am deeply thankful for his contributions..

Finally, I would want to express my gratitude to all the other people and organizations that have supported the project in various ways even though they aren't listed here.

This project would not have been possible without the cooperation, hard work, and dedication of those involved. I sincerely hope that we can continue to collaborate and accomplish similar feats in the future.

Ho Chi Minh City, day 08 month 02 year 2025

Author

(Signature and full name)

Hai, Duy

LAM NGOC HAI

PHAN AN DUY

DECLARATION OF AUTHORSHIP

I hereby declare that this project report is my own work and that all sources of information and data have been duly acknowledged. I affirm that I have not received unauthorized assistance in preparing this report and that all content and ideas presented are original, except where explicitly stated otherwise.

By signing this declaration, I take full responsibility for the authenticity and integrity of this work.

I will take full responsibility for any fraud detected in my thesis. Ton Duc Thang University is unrelated to any copyright infringement caused on my work (if any).

Ho Chi Minh City, day 08 month 02 year 2025

Author

Hai , Duy

Lam Ngoc Hai

Phan An Duy

(signature and full name)

CREDIT CARD FRAUD DETECTION WITH IMBALANCED DATA

Introduction

Credit card fraud detection with imbalanced data refers to challenge of identifying fraudulent transaction when the dataset contains a significantly higher number of legitimate transactions compared to fraudulent ones. This imbalance make it difficult for machine learning models to accurately detect fraud, as they tent to be biased towards the majority class(Legitimate transactions)

Why we need to balanced data?

Balancing data is crucial in credit card fraud detection for several reasons:

1. **Improved Model Performance:** Machine learning models can struggle with imbalanced data because they tend to predict the majority class (Legitimate transactions) more often. Balancing the data helps the model learn to identify both classes more accurately.
2. **Reduced Bias:** When the data is imbalanced, the model can become biased towards the majority class. Balancing the data mitigates this bias, ensuring that the model pays attention to the minority class (Fraudulent transactions).
3. **Enhanced Detection of Fraud:** Since fraudulent transactions are rare, they can be easily overlooked by the model. Balancing the data increase the model's ability to detect these rare events, improving the overall effectiveness of fraud detection.
4. **Better Evaluation Metrics:** Standard evaluation metrics like accuracy can be misleading with imbalanced data. Balancing the data allows for more meaningful metrics like Precision, Recall, and F-1 Score, which provide a clearer picture of the model's performance.

Challenges of Imbalanced Data

Imbalanced data can lead to several issues when training machine learning models:

1. Bias towards Majority Class: Models trained on imbalanced data tend to be biased towards the majority class, resulting in poor detection of the minority class.
2. Metric Misleading: Traditional evaluation metrics like accuracy can be misleading. A model that classifies all transactions as legitimate may have high accuracy but will fail to detect fraud.

Techniques to Address Imbalanced Data

Various techniques are employed to handle imbalanced datasets:

1. **Resampling Methods:**

- **Oversampling:** This involves increasing the number of instances in the minority class. Techniques like Synthetic Minority Over-sampling Technique (SMOTE) generate synthetic examples.
- **Undersampling:** This reduces the number of instances in the majority class. Random undersampling selects a subset of the majority class to balance the dataset.

2. **Cost-Sensitive Learning:**

- This approach assigns a higher cost to misclassifying the minority class (fraudulent transactions), encouraging the model to focus more on detecting fraud.

3. **Ensemble Models:**

- Combining multiple models, such as Random Forest or Gradient Boosting, can improve detection performance by leveraging the strengths of different algorithms.

4. **Anomaly Detection:**

- Since fraud is rare, treating it as an anomaly detection problem can be effective. Techniques like Isolation Forest and One-Class SVM can help identify outliers that may represent fraudulent transactions.

5. **Feature Engineering:**

- Creating new features that capture patterns of fraudulent behavior can improve model performance. For example, features like transaction frequency, location patterns, and purchase amounts can provide valuable insights.

Evaluation Metrics

When dealing with imbalanced data, it's crucial to use appropriate evaluation metrics:

1. **Precision:** The ratio of true positives to the sum of true positives and false positives.
2. **Recall:** The ratio of true positives to the sum of true positives and false negatives.
3. **F1-Score:** The harmonic mean of precision and recall.
4. **Area Under the ROC Curve (AUC-ROC):** Measures the model's ability to distinguish between classes.

Conclusion

Credit card fraud detection with imbalanced data is a complex but critical task. By employing techniques like resampling, cost-sensitive learning, ensemble models, anomaly detection, and careful feature engineering, it's possible to build robust models that effectively identify fraudulent transactions. Appropriate evaluation metrics are essential to ensure the model's performance is accurately assessed.

CONTENTS

LIST OF FIGURES

Chapter 1: Read And Understanding Data

1 Import Libraries:

```
[42]  1  import numpy as np # linear algebra
      2  import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
      3  import matplotlib.pyplot as plt
      4  import seaborn as sns
      5
      6  import math
      7
      8  # Classifier Libraries
      9  from sklearn.linear_model import LogisticRegression
     10  from sklearn.svm import SVC
     11  from sklearn.neighbors import KNeighborsClassifier
     12  from sklearn.tree import DecisionTreeClassifier
     13  from sklearn.ensemble import RandomForestClassifier
     14
     15  import warnings
     16  warnings.filterwarnings("ignore") # Suppress all the warking of the dataset
```

Importing libraries in Python is essential for several reasons:

1. **Code Reusability:** Libraries contain pre-written code that you can use in your projects. Instead of writing code from scratch, you can import libraries that offer the functionalities you need, saving time and effort.
2. **Access to Advanced Features:** Libraries provide access to advanced features and tools that would be difficult to implement on your own. For example, libraries like NumPy and pandas offer powerful data manipulation capabilities, while libraries like TensorFlow and PyTorch provide tools for machine learning and deep learning.
3. **Standardization:** Using well-established libraries ensures that your code follows industry standards and best practices. This can make your code more reliable and easier to understand for others who may work on your project.
4. **Community Support:** Popular libraries often have strong community support, including extensive documentation, tutorials, and forums. This makes it easier to learn how to use the library and get help when you encounter issues.

5. **Efficiency:** Libraries are often optimized for performance. By using them, you can leverage optimized algorithms and data structures that can significantly improve the efficiency of your code.

2 Understanding our data:

Reading out the dataset, fully formatted. For this is to having a glimpse of what is coming about the dataset, about the imbalance column, value, data, and numbers. From there we will have a look on what we are working on the dataset.

Also on this dataset, we'll try to cover it all the rows and columns, to have a better sense on what we are working on.

- id: Unique identifier for each transaction
- V1-V28: Anonymized features representing various transaction attributes (e.g., time, location, etc.)
- Amount: The transaction amount
- Class: Binary label indicating whether the transaction is fraudulent (1) or not (0)

To read dataset we use this command

```
In [3]: dataset = pd.read_csv("creditcard.csv")
```

3 Check The Data:

To begin with the dataset, let's see how to dataset went.

```
In [4]:
# The classes are heavily skewed we need to solve this issue later.
print('We have detected no Frauds Credit', round(dataset['Class'].value_counts()[0]/len(dataset) * 100,2), '% of the data
print('We have detected Frauds Credit', round(dataset['Class'].value_counts()[1]/len(dataset) * 100,2), '% of the dataset

We have detected no Frauds Credit 99.83 % of the dataset
We have detected Frauds Credit 0.17 % of the dataset
```

Printing Non-Fraudulent Transactions:

- `dataset['Class'].value_counts():` This function counts the occurrences of each class in the 'Class' column of the dataset. It returns a Series with the count of each class (0 for non-fraudulent and 1 for fraudulent).
- `value_counts()[0]:` This retrieves the count of non-fraudulent transactions (class 0).
- `len(dataset):` This returns the total number of transactions in the dataset.
- `dataset['Class'].value_counts()[0]/len(dataset) * 100`: This calculates the percentage of non-fraudulent transactions in the dataset.
- `round(..., 2):` This rounds the percentage to two decimal places.
- `print('We have detected no Frauds Credit', ..., '% of the dataset'):` This prints the percentage of non-fraudulent transactions along with a message.

From the dataset, there are 99.83 % of non-fraudulent and 0.17% of the fraudulent credit cards. The data are imbalance due to the lack of non-secure transaction

Printing Fraudulent Transactions:

- Similar to the first print statement, but this one is for fraudulent transactions.
- `value_counts()[1]:` This retrieves the count of fraudulent transactions (class 1).

- The rest of the calculation follows the same steps to determine the percentage of fraudulent transactions.

Example Output

Assuming the dataset has 10,000 transactions, with 9,950 non-fraudulent and 50 fraudulent transactions, the output would be:

```
We have detected no Frauds Credit 99.83 % of the dataset
We have detected Frauds Credit 0.17 % of the dataset
```

4 Glimpse the data:

As at the time we detection, with non format all the dataset we can find the data value seem to be off set to no Frauds Credit, with the majority of 50% of the value and only 50% are the Frauds one

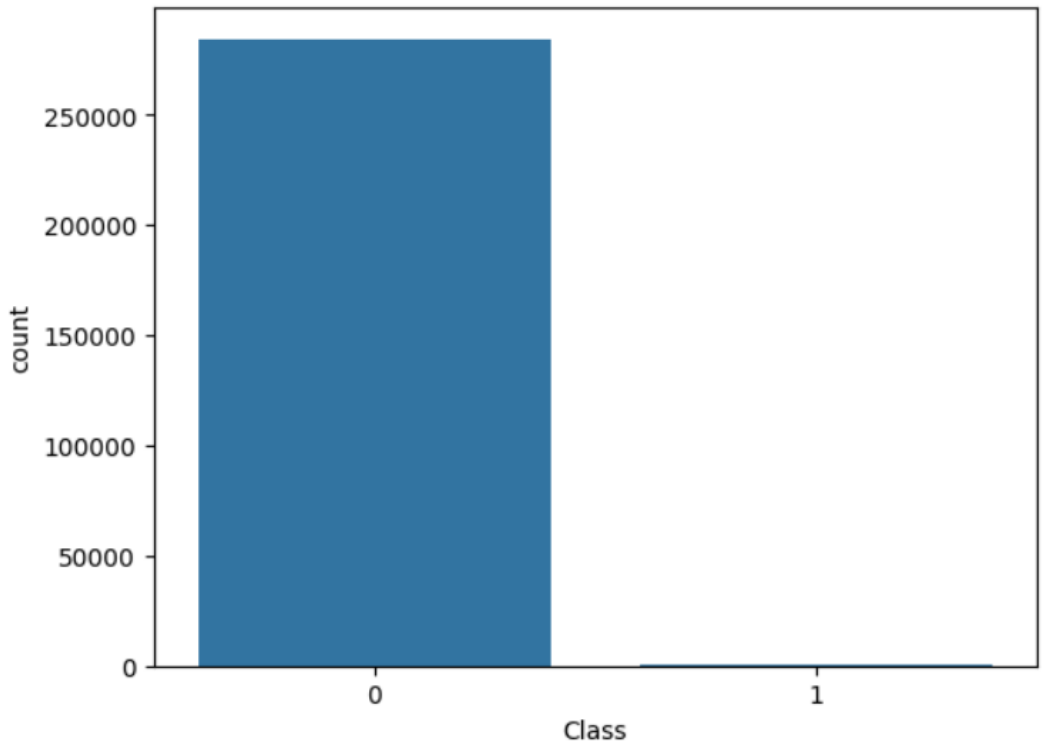
Definition: Because most of the transaction need to be secure during all the transaction, with many transaction need to be secure before can other transaction can be made. So the value contain the fraud only 50% of the all the transaction made it. The dataset are validate to use as a research purposeAutomotive Industry

In [5]:

```
sns.countplot(x='Class', data=df)
```

Out[5]:

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



As the label are 0 and 1 are Non-Fraudulent and Fraudulent respectively, and the dataset are in the bias of the Non-fraud side, with since today the transaction are in the secure side, the dataset for fraud is minimal

```
dataset.describe()
```

	Time	V1	V2	V3	V4	V5	V6	V7	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	1.2
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.1
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.3
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.0
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.2
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.2
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.0

8 rows × 31 columns

5 Result Meaning

```
plt.figure(figsize=(15,15))
t = 1
for i in dataset.columns:
  plt.subplot(7,5,t)
  sns.histplot(dataset[i], kde= True)
  plt.title(i+' Distribution')
  t+= 1
plt.tight_layout()
plt.show()
```

Set Figure Size: This sets the size of the entire figure to 15x15 inches.

Initialize Counter: A counter t is initialized to 1. This counter is used to specify the position of the subplot within the figure.

Loop Through Columns:

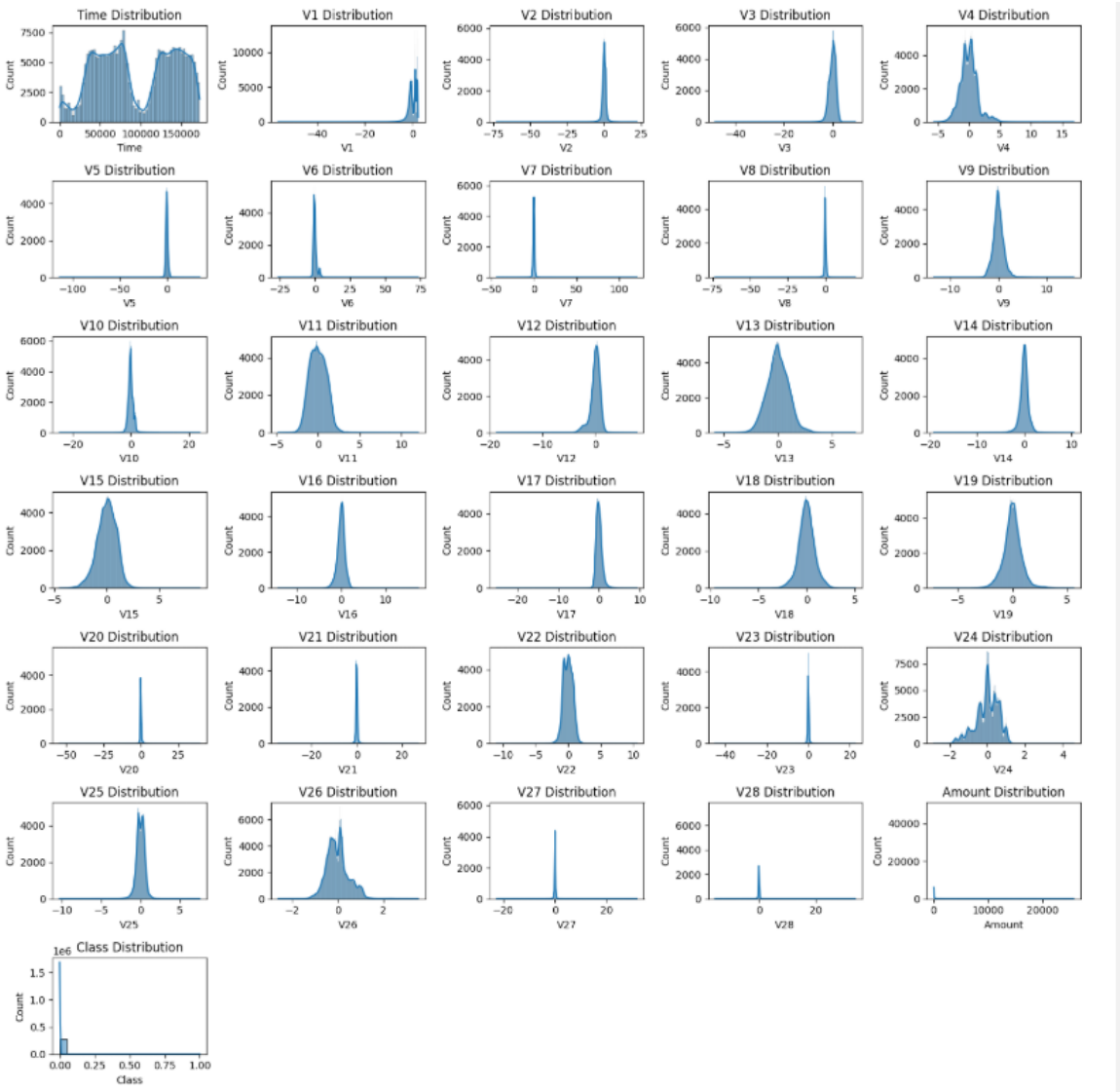
- The loop iterates through each column in the dataset.
- plt.subplot(7,5,t):

This creates a subplot in a 7x5 grid at the position specified by t.

- sns.histplot(dataset[i], kde=True):
  - This creates a histogram for the column i using Seaborn's histplot function.
  - kde=True adds a Kernel Density Estimate (KDE) curve to the histogram, which represents the data's probability density function.
- plt.title(i+' Distribution'):
  - This sets the title of the subplot to indicate the column name followed by "Distribution".
- t+= 1 increments the counter t by 1 for the next subplot position.

Adjust Layout: This adjusts the layout of the subplots to ensure there is minimal overlap and they fit neatly within the figure.

Show the Plot: This displays the entire figure with all the subplots.



6 Data Preprocessing

In [8]:

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
df['Amount'] = sc.fit_transform(pd.DataFrame(df['Amount']))
```

This code helps standardize the 'Amount' column in the DataFrame so that its values have a mean of 0 and a standard deviation of 1.

In [9]:

```
from sklearn.preprocessing import RobustScaler

# RobustScaler is less prone to outliers.

std_scaler = StandardScaler()
rob_scaler = RobustScaler()

df['scaled_amount'] = rob_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['scaled_time'] = rob_scaler.fit_transform(df['Time'].values.reshape(-1,1))

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

This code imports the necessary scaler, creates instances of the scalers, scales the 'Amount' and 'Time' columns using the RobustScaler, and then removes the original columns from the DataFrame.

These will scale down to match the label ready for the dataset sampling and training

[52]

```
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import StandardScaler

# RobustScaler is less prone to outliers.

std_scaler = StandardScaler()
rob_scaler = RobustScaler()

df['Amount'] = std_scaler.fit_transform(pd.DataFrame(df['Amount']))

df['scaled_amount'] = rob_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['scaled_time'] = rob_scaler.fit_transform(df['Time'].values.reshape(-1,1))

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

[53]

```
scaled_amount = df['scaled_amount']
scaled_time = df['scaled_time']

df.drop(['scaled_amount', 'scaled_time'], axis=1, inplace=True)
df.insert(0, 'scaled_amount', scaled_amount)
df.insert(1, 'scaled_time', scaled_time)


df.head()
```

Code sample of scaling the data set, using fit and transform.

Chapter 2: Undersampling The Dataset

Undersampling the dataset is a technique used to address class imbalance, which is a common issue in machine learning. This approach involves reducing the number of instances in the majority class to create a more balanced dataset.

Usage: Undersampling is applied when the dataset contains a significantly larger number of instances for one class compared to another. By randomly selecting

and removing instances from the majority class, we can create a more balanced distribution of classes. This is particularly useful for algorithms that are sensitive to class imbalance, as it helps them to better detect patterns and avoid being biased towards the majority class.

Based on the dataset, we can see the non fraudulent are massive, so for the first undersampling method, we will be using the Random Undersampling Method, Later we will use the NearMiss Method to test the differentials.

a. Using Random Under Sampling

This code rearranges the DataFrame df by first extracting the scaled columns, dropping them from their original positions, and then reinserting them at the beginning of the DataFrame. This ensures that the DataFrame has the *'scaled_amount'* and *'scaled_time'* columns in the first two positions.

In [11]:

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

# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0][:492]

normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

# Shuffle dataframe rows
new_df = normal_distributed_df.sample(frac=1, random_state=42)

new_df.head()
```

Out[11]:

	scaled_amount	scaled_time	V1	V2	V3	V4	V5	V
35686	-0.104800	-0.545942	-0.421659	0.517644	1.558363	0.063718	-0.683353	0.7715
252124	-0.296653	0.833774	-1.928613	4.601506	-7.124053	5.716088	1.026579	-3.1890
29973	-0.041780	-0.575312	1.249660	-1.058342	0.309025	-2.108688	-1.029756	0.1329
10484	-0.254454	-0.793066	1.088375	0.898474	0.394684	3.170258	0.175739	-0.2219
223366	-0.293440	0.689176	1.118331	2.074439	-3.837518	5.448060	0.071816	-1.0205

5 rows × 31 columns

Code sample using Random Under Sampling

This code creates a balanced dataset with an equal number of fraud and non-fraud samples, concatenates them into a single DataFrame, shuffles the rows to

ensure randomness, and then displays the first few rows of the shuffled DataFrame, after under sampling the dataset by dropping the non-fraud scale down to the same size as the fraud.

After scale down, the dataset are now equal to the train test

This code will generate a bar plot showing the count of each class (fraud and non-fraud) in the new_df DataFrame. The sns.countplot function creates the plot, and plt.show() displays it.

In [13]:

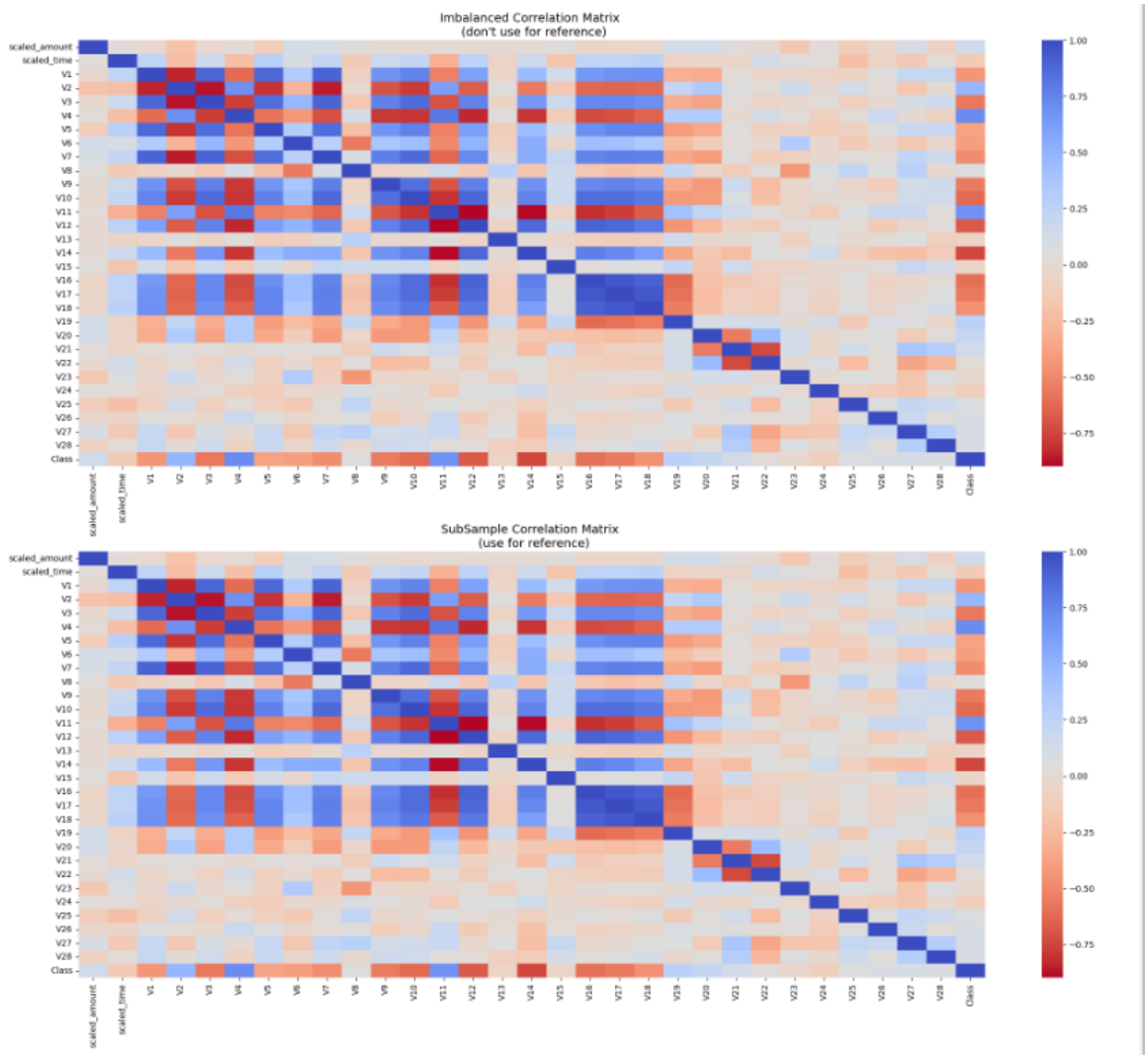
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
for i in new_df.columns:
  if i != 'Class':
    new_df[i] = scaler.fit_transform(new_df[[i]])
```

This code standardizes all columns in the DataFrame new_df except for the 'Class' column, ensuring that the feature values have a mean of 0 and a standard deviation of 1.

In [16]:

```
# Make sure we use the subsample in our correlation

f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

# Entire DataFrame
corr = new_df.corr()
sns.heatmap(corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title("Imbalanced Correlation Matrix \n (don't use for reference)", fontsize=14)


sub_sample_corr = new_df.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax2)
ax2.set_title('SubSample Correlation Matrix \n (use for reference)', fontsize=14)
plt.show()
```

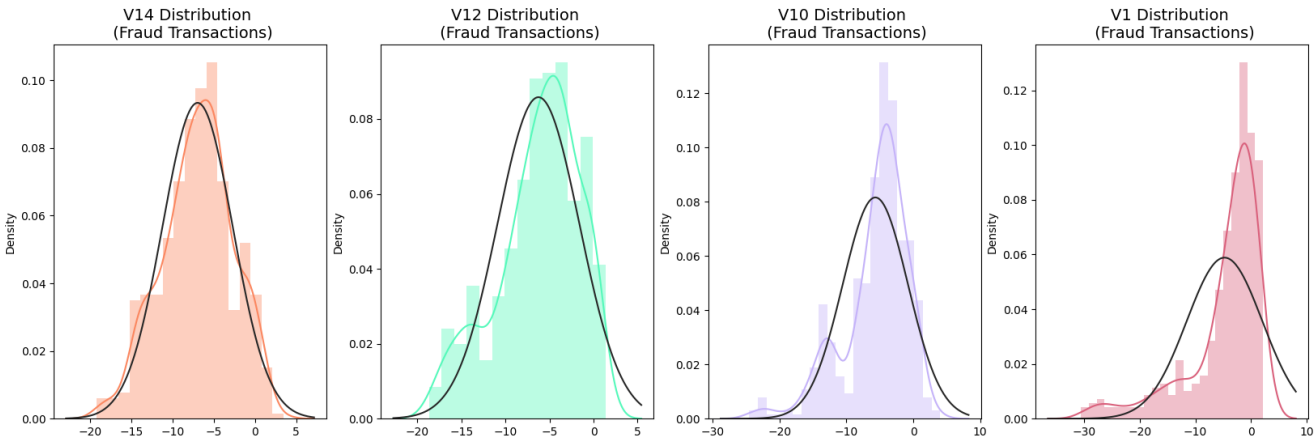
This code creates a figure with two subplots: one showing the correlation matrix for the entire DataFrame and the other showing the correlation matrix for the subsample. The heatmaps visualize the correlation values between different features, and the titles indicate which plot should be used for reference.



In [17]:

```
from scipy.stats import norm

f, (ax1, ax2, ax3, ax4) = plt.subplots(1,4, figsize=(20, 6))

v14_fraud_dist = new_df['V14'].loc[new_df['Class'] == 1].values
sns.distplot(v14_fraud_dist,ax=ax1, fit=norm, color='#FB8861')
ax1.set_title('V14 Distribution \n (Fraud Transactions)', fontsize=14)

v12_fraud_dist = new_df['V12'].loc[new_df['Class'] == 1].values
sns.distplot(v12_fraud_dist,ax=ax2, fit=norm, color='#56F9BB')
ax2.set_title('V12 Distribution \n (Fraud Transactions)', fontsize=14)


v10_fraud_dist = new_df['V10'].loc[new_df['Class'] == 1].values
sns.distplot(v10_fraud_dist,ax=ax3, fit=norm, color='#C5B3F9')
ax3.set_title('V10 Distribution \n (Fraud Transactions)', fontsize=14)

v1_fraud_dist = new_df['V1'].loc[new_df['Class'] == 1].values
sns.distplot(v1_fraud_dist,ax=ax4, fit=norm, color='#DA627D')
ax4.set_title('V1 Distribution \n (Fraud Transactions)', fontsize=14)

plt.show()
```

This code creates a figure with four subplots, each showing the distribution of a different feature ('V14', 'V12', 'V10', 'V1') for fraud transactions. Each distribution is fitted with a normal distribution, and the subplots are arranged horizontally.



b. Using NearMiss UnderSampling

Near Miss UnderSampling is another method to scale down the dataset as it shown in the code below

```
[117]
1  from imblearn.under_sampling import NearMiss
2
3  # amount of fraud classes 492 rows.
4  nm_fraud_df = df.drop(columns=['Class'])
5  nm_non_fraud_df = df['Class']
6
7  nm_non_fraud_df = nm_non_fraud_df.dropna()
8  nm_fraud_df = nm_fraud_df.dropna()
9
10 nm = NearMiss(version=1)  # Version 1 selects majority samples with smallest avg distance to minority class
11 X_resampled, y_resampled = nm.fit_resample(nm_fraud_df, nm_non_fraud_df)
12
13 print (X_resampled.shape, y_resampled.shape)
```

```
(984, 30) (984,)
```

Using the library to test with the fill dataset, create a new dataset after scaled down, balanced out. But it may cost the fact that some of the field have prone to accommodate the NaN, so on line 7-8 will dropna() all from the dataset.

II. TRAIN-TEST-SPLIT

1. Using for Random Under Sampling

a. Train, test, split

These method for training, processing the dataset, to match the usage need. This first segment for train test split will apply for the RUS method

```
1  # Splitting the dataset.
2
3  from sklearn.model_selection import train_test_split
4  X = new_df.drop('Class', axis = 1)
5  y = new_df['Class']
6
7  df2 = new_df.copy()
8  df3 = new_df.copy()
9  X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=10)
```

Data will be trained and store as X_train, X_test, Y_train, Y_test

The code performs the following steps:

1. Imports the necessary library.
2. Defines the features (X) and target (y) variables.
3. Creates copies of the dataset.
4. Splits the dataset into training and testing sets , with different test sizes and random seeds.

This is useful for preparing the data for model training and evaluation while ensuring that the class distribution is preserved.

This code is helpful for understanding the dimensions of your datasets, especially when you're working with train-test splits and need to ensure they are correctly partitioned.

In [20]:

```
# Using the Isolation Forest to recording the anomaly of the dataset

from sklearn.ensemble import IsolationForest
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report

iso = IsolationForest(n_estimators= 100, contamination= 0.0016, random_state= 28)
iso.fit(X)

df2['anomaly'] = iso.predict(X)
df2['anomaly'] = df2['anomaly'].map({1:0, -1:1})
```

In [22]:

```
pd.DataFrame(df2)
```

Out[22]:

This code uses the Isolation Forest algorithm to detect anomalies in the dataset X, predicting anomalies and mapping the results to a new column 'anomaly' in the DataFrame df2.

```
pd.DataFrame(df2)
```

Out[22]:

	scaled_amount	scaled_time	V1	V2	V3	V4	V5	V
35686	-0.376702	-1.023168	0.350163	-0.385271	0.809305	-0.676018	0.211380	0.8571
252124	-0.442177	1.421247	0.075503	0.752384	-0.584225	1.083005	0.619494	-1.4323
29973	-0.355195	-1.075203	0.654780	-0.824300	0.608786	-1.352073	0.128703	0.4880
10484	-0.427775	-1.460994	0.625384	-0.279182	0.622534	0.290740	0.416422	0.2828
223366	-0.441080	1.165066	0.630844	0.048410	-0.056736	0.999594	0.391618	-0.1787
...	...	...	...	...	...	...	...	
16863	3.039433	-1.230893	-0.080993	-0.549347	0.317588	0.350341	-0.426849	0.7382
15204	0.030978	-1.265986	-3.068726	2.762687	-2.958846	1.198195	-3.136129	-2.0797
218952	-0.422052	1.127247	0.280115	-0.460027	0.665968	-1.155795	1.008118	2.7233
42756	-0.445849	-0.960496	-1.513303	1.119210	-1.314018	2.045015	-1.529060	-1.6622
45203	-0.441080	-0.939391	-0.033015	0.055016	-0.234659	1.313449	1.132007	-1.3570

984 rows × 32 columns

b. Building LogisticRegression with RUS

```
[61]  1  from sklearn.model_selection import cross_val_score
0s
```

```
0s
 1  # Using the LogisticRegression as the methodology of training model
 2
 3  model = LogisticRegression()
 4  model2 = LogisticRegression(solver='saga', random_state=10)
 5
 6  model.fit(X_train, Y_train)
 7  model2.fit(X_train, Y_train)
 8  # accuracy on training data
 9  X_train_prediction = model.predict(X_train)
10  training_data_accuracy = accuracy_score(X_train_prediction, Y_train)
11
12  X_train_prediction_2 = model2.predict(X_train)
13  training_data_accuracy_2 = accuracy_score(X_train_prediction_2, Y_train)
14
15  y_pred_log_reg = model.predict(X_test)
16
17  print("Logistic Regression Report")
18  print(classification_report(Y_test, y_pred_log_reg))
19
20  # accuracy on test data
21  X_test_prediction = model.predict(X_test)
22  test_data_accuracy = accuracy_score(X_test_prediction, Y_test)
23
24  print("Logistic Regression Testing Data Accuracy", round(test_data_accuracy, 2) * 100, "%")
```

Coding method for Logistic Regression

This method will fit the LogisticRegression for the RUS sampling, as the result will log as the classification report here

```
Logistic Regression Report
              precision    recall  f1-score   support

           0       0.89      0.98      0.93        98
           1       0.98      0.88      0.93        99

    accuracy                           0.93       197
   macro avg       0.93      0.93      0.93       197
weighted avg       0.93      0.93      0.93       197

Logistic Regression Testing Data Accuracy 93.0 %
```

As the Logistic Regression method training out, the accuracy are 93% of the efficiency

c. Using Decision Tree Method

Decision Tree is a supervised learning algorithm that can solve both regression and classification problems. As with this method, the code below are representing it

```
decision_model = DecisionTreeClassifier()
decision_model.fit(X_train, Y_train)
prediction = decision_model.predict(X_test)

print("Decision Tree Report")
print(classification_report(Y_test, prediction))

test_data = accuracy_score(prediction, Y_test)
print("Prediction using the DecisionTree Classifier",round(test_data, 2) * 100, "%")
```

The prediction testing out for this method are

```
Decision Tree Report
              precision    recall  f1-score   support

           0       0.87      0.86      0.86        98
           1       0.86      0.87      0.86        99

    accuracy                           0.86       197
   macro avg       0.86      0.86      0.86       197
weighted avg       0.86      0.86      0.86       197

Prediction using the DecisionTree Classifier 86.0 %
```

d. Using K-Nearest Neighbors Classifier

K-nearest neighbor is one of the simplest (and in some cases effective) supervised-learning algorithms in Machine Learning. During training, the algorithm does not learn anything from the training data (which is why it is classified as lazy learning)

For this the code below representing this method

```
K_model = KNeighborsClassifier()
K_model.fit(X_train, Y_train)
K_prediction = K_model.predict(X_test)

print("K-Nearest Neighbor Classifier Report")
print(classification_report(Y_test, K_prediction))

K_test_data = accuracy_score(K_prediction, Y_test)
print("Prediction using the K-Nearest Neighbor: ", round(K_test_data, 2)* 100, "%")
```

And the Result is:

```
K-Nearest Neighbor Classifier Report
              precision    recall  f1-score   support

           0       0.90      0.96      0.93        98
           1       0.96      0.89      0.92        99

    accuracy                           0.92       197
   macro avg       0.93      0.92      0.92       197
weighted avg       0.93      0.92      0.92       197

Prediction using the K-Nearest Neighbor:  92.0 %
```

e. Using Support Vector Machine

```
[96]  1  from sklearn.svm import SVC
      2
      3  svm = SVC(kernel="linear")
      4  svm.fit(X_train, Y_train)
      5
      6  y_pred_svm = svm.predict(X_test)
      7
      8  print("SVM Report")
      9  print(classification_report(Y_test, y_pred_svm))
```

And the result for using SVM are

```
SVM Report
                precision    recall  f1-score   support

             0       0.88      0.96      0.92        98
             1       0.96      0.87      0.91        99

      accuracy                           0.91       197
     macro avg       0.92      0.91      0.91       197
  weighted avg       0.92      0.91      0.91       197
```

The test for the SVM are in the relatively in 92%. After using the RUS Sample

2. Using NearMiss Under Sampling Method

a. Train Test Split

Beside of using the RUS, we have the method of using the Near Miss as well, the code below are the sampling of using the NearMiss

```
+ Code  + Text
1  from imblearn.under_sampling import NearMiss
2
3  # amount of fraud classes 492 rows.
4  nm_fraud_df = df.drop(columns=['Class'])
5  nm_non_fraud_df = df['Class']
6
7  nm_non_fraud_df = nm_non_fraud_df.dropna()
8  nm_fraud_df = nm_fraud_df.dropna()
9
10 nm = NearMiss(version=1)
11 X_resampled, y_resampled = nm.fit_resample(nm_fraud_df, nm_non_fraud_df)
12
```

We will process as well with the droping NaN section of the dataset

After that, we'll train-test-split with this new dataset

```
1  X_train_NM, X_test_NM, Y_train_NM, Y_test_NM = train_test_split(X_resampled, y_resampled,
2                                                                  test_size=0.2,
3                                                                  stratify=y_resampled, random_state=10)
```

b. Using LogisticRegression Model

```python
# Using the LogisticRegression as the methodology of training model

model = LogisticRegression()
model2 = LogisticRegression(solver='saga', random_state=10)

model.fit(X_train_NM, Y_train_NM)
model2.fit(X_train_NM, Y_train_NM)
# accuracy on training data
X_train_prediction = model.predict(X_train_NM)
training_data_accuracy = accuracy_score(X_train_prediction, Y_train_NM)

X_train_prediction_2 = model2.predict(X_train_NM)
training_data_accuracy_2 = accuracy_score(X_train_prediction_2, Y_train_NM)
print("Logistic Regression Training Data Accuracy: ", round(training_data_accuracy, 2) * 100, "%")
print("Logistic Regression Training Data Accuracy (Model2): ", round(training_data_accuracy_2, 2) * 100, "%")

# accuracy on test data
X_test_prediction = model.predict(X_test_NM)
test_data_accuracy = accuracy_score(X_test_prediction, Y_test_NM)


print(classification_report(Y_test_NM, X_test_prediction))
print("Logistic Regression Testing Data Accuracy", round(test_data_accuracy, 2) * 100, "%")
```

After running the model, we can see the report below

```
Logistic Regression Training Data Accuracy:  97.0 %
Logistic Regression Training Data Accuracy (Model2):  96.0 %
              precision    recall  f1-score   support

         0.0       0.97      0.97      0.97        34
         1.0       0.97      0.97      0.97        34

    accuracy                           0.97        68
   macro avg       0.97      0.97      0.97        68
weighted avg       0.97      0.97      0.97        68

Logistic Regression Testing Data Accuracy 97.0 %
```

c. Using Decision Tree Classifier

```python
decision_model = DecisionTreeClassifier()
decision_model.fit(X_train_NM, Y_train_NM)
prediction = decision_model.predict(X_test_NM)
test_data = accuracy_score(prediction, Y_test_NM)

print("Decision Tree Report")
print(classification_report(Y_test_NM, prediction))

print("Prediction using the DecisionTree Classifier",round(test_data, 2) * 100, "%")
```

And the result are

```
Decision Tree Report
              precision    recall  f1-score   support

         0.0       0.97      0.97      0.97        34
         1.0       0.97      0.97      0.97        34

    accuracy                           0.97        68
   macro avg       0.97      0.97      0.97        68
weighted avg       0.97      0.97      0.97        68

Prediction using the DecisionTree Classifier 97.0 %
```

d. Using the SVM

```
0s
1  from sklearn.svm import SVC
2
3  svm = SVC(kernel="linear")
4  svm.fit(X_train_NM, Y_train_NM)
5
6  # Predict
7  y_pred_svm = svm.predict(X_test_NM)
8  test_acc = accuracy_score(y_pred_svm, Y_test_NM)
9  print("Prediction using the SVM Classifier",round(test_acc, 2) * 100, "%")
10
11 # Evaluate
12 print("Support Vector Classification Model for NearMiss Undersampling")
13 print(classification_report(Y_test_NM, y_pred_svm))
```

And the result are:

```
Prediction using the SVM Classifier 96.0 %
Support Vector Classification Model for NearMiss Undersampling
              precision    recall  f1-score   support

         0.0       0.94      0.97      0.96        34
         1.0       0.97      0.94      0.96        34

    accuracy                           0.96        68
   macro avg       0.96      0.96      0.96        68
weighted avg       0.96      0.96      0.96        68
```

3. Using Clustered Centroid

a. Train test split

This section will use the Clustered Centroid method for undersampling

```python
# Initialize ClusterCentroids object
from imblearn.under_sampling import ClusterCentroids

cc = ClusterCentroids(random_state=42)

clustered_fraud_df = new_df.drop(columns=['Class'])
clustered_non_fraud_df = new_df['Class']

clustered_non_fraud_df = clustered_non_fraud_df.dropna()
clustered_fraud_df = clustered_fraud_df.dropna()

# Resample the dataset
X_cluster, y_cluster = cc.fit_resample(clustered_fraud_df, clustered_non_fraud_df)

# Check the class distribution
print("Resampled class distribution:")
print(clustered_non_fraud_df.value_counts())
```

And after sampling we get the new dataset as well:

```
Resampled class distribution:
Class
0    492
1    492
Name: count, dtype: int64
```

After that we will train-test-split with the dataset

```python
X_train_cc, X_test_cc, Y_train_cc, Y_test_cc = train_test_split(X_cluster, y_cluster,
                                                    test_size=0.2,
                                                    stratify=y_cluster, random_state=10)
```

b. Using Decision Tree Model

The code below show the function to use this method:

```python
decision_model = DecisionTreeClassifier()
decision_model.fit(X_train_cc, Y_train_cc)
prediction = decision_model.predict(X_test_cc)

test_data = accuracy_score(prediction, Y_test_cc)
print("Prediction using the DecisionTree Classifier",round(test_data, 2) * 100, "%")

print("Decision Tree Report")
print(classification_report(Y_test_cc, prediction))
```

And the result are

```
Prediction using the DecisionTree Classifier 90.0 %
Decision Tree Report
              precision    recall  f1-score   support

           0       0.91      0.89      0.90        98
           1       0.89      0.91      0.90        99

    accuracy                           0.90       197
   macro avg       0.90      0.90      0.90       197
weighted avg       0.90      0.90      0.90       197
```

c. Using SVM

```
svm = SVC(kernel="linear")
svm.fit(X_train_cc, Y_train_cc)

# Predict
y_pred_svm = svm.predict(X_test_cc)

# Evaluate
print("Support Vector Classification Model for ClusterdCentroid Undersampling")
print(classification_report(Y_test_cc, y_pred_svm))
```

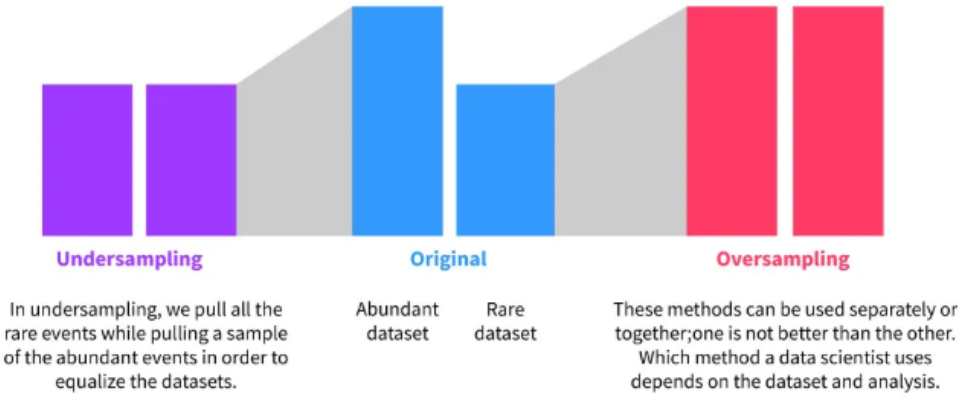
And the result are

```
Support Vector Classification Model for ClusterdCentroid Undersampling
              precision    recall  f1-score   support

           0       0.92      0.96      0.94        98
           1       0.96      0.92      0.94        99

    accuracy                           0.94       197
   macro avg       0.94      0.94      0.94       197
weighted avg       0.94      0.94      0.94       197
```

III. Understanding The Over Sampling Method

1. Using traditional SMOTE approach

The dataset we work imbalanced, because of the inequality of the fraudalent - non-fraudulent. We need to Scale the dataset to, with SMOTE (to balanced out the imbalanced one), by multiplying the fraudent one to match the scale of the non-fraudulent. Oversampling in here are using the SMOTE, ADASYN




```python
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit, StratifiedKFold
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline

from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, RandomizedSearchCV

sss = StratifiedKFold(n_splits=5, random_state=None, shuffle=False)

for train_index, test_index in sss.split(X, y):
    # print("Train:", train_index, "Test:", test_index)
    X_train_ovs, X_test_ovs = X.iloc[train_index], X.iloc[test_index]
    Y_train_ovs, Y_test_ovs = y.iloc[train_index], y.iloc[test_index]

# Turn into an array
SMOTE_X_train = X_train_ovs.values
SMOTE_X_test = X_test_ovs.values
SMOTE_Y_train = Y_train_ovs.values
SMOTE_Y_test = Y_test_ovs.values

# See if both the train and test label distribution are similarly distributed
train_unique_label, train_counts_label = np.unique(SMOTE_Y_train, return_counts=True)
test_unique_label, test_counts_label = np.unique(SMOTE_Y_test, return_counts=True)
print (SMOTE_X_train.shape, SMOTE_Y_train.shape)
print (SMOTE_X_test.shape, SMOTE_Y_test.shape)
print('Label Distributions: \n')
print(train_counts_label/ len(SMOTE_Y_train))
print(test_counts_label/ len(SMOTE_Y_test))
```

- The code imports necessary libraries and modules.
- It initializes StratifiedKFold for 5 splits.
- The loop splits the data into training and testing sets using stratified sampling to ensure balanced label distribution.
- The training and testing sets are converted to arrays.
- Finally, it checks and prints the label distribution in both training and testing sets to ensure they are similarly distributed.

The result of using SMOTE Oversampling are

```
(788, 30) (788,)
(196, 30) (196,)
Label Distributions:

[0.5 0.5]
[0.5 0.5]
```

- **Goal:** Train and evaluate a Logistic Regression model to detect anomalies using imbalanced datasets and improve performance through hyperparameter tuning and cross-validation.

- **Steps:**

    1. **Import Libraries:** Bring in necessary libraries for data preprocessing, model training, and evaluation.
    2. **Set Up Parameters:** Define hyperparameters for Logistic Regression and initialize models.
    3. **Cross-Validation:** Use StratifiedKFold for cross-validation, ensuring balanced label distribution.
    4. **Handle Imbalanced Data:** Apply SMOTE (Synthetic Minority Over-sampling Technique) to handle class imbalance.
    5. **Train and Evaluate:** Train the model, make predictions, and calculate performance metrics (accuracy, precision, recall, F1-score) for each fold.
    6. **Report Results:** Print the average performance metrics.

- **Purpose:** To build a reliable anomaly detection model and ensure fair evaluation using cross-validation and proper handling of imbalanced data.

After declare the data validation using SMOTE, the code will train and put as the accuracy are below

```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import recall_score, precision_score, f1_score, accuracy_score


log_reg_params = {"penalty": ['l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}
log_reg_sm = LogisticRegression()
rand_log_reg = RandomizedSearchCV(LogisticRegression(), log_reg_params, n_iter=4)


accuracy_lst = []
precision_lst = []
recall_lst = []
f1_lst = []
auc_lst = []


for train, test in sss.split(SMOTE_X_train, SMOTE_Y_train):
    pipeline = imbalanced_make_pipeline(SMOTE(sampling_strategy='minority'), rand_log_reg) #
    model = pipeline.fit(SMOTE_X_train[train], SMOTE_Y_train[train])
    best_est = rand_log_reg.best_estimator_
    prediction = best_est.predict(SMOTE_X_train[test])

    accuracy_lst.append(pipeline.score(SMOTE_X_train[test], SMOTE_Y_train[test]))
    precision_lst.append(precision_score(SMOTE_Y_train[test], prediction))
    recall_lst.append(recall_score(SMOTE_Y_train[test], prediction))
    f1_lst.append(f1_score(SMOTE_Y_train[test], prediction))

print("accuracy: {}".format(round(np.mean(accuracy_lst)*100)),"%")
print("precision: {}".format(round(np.mean(precision_lst)*100)),"%")
print("recall: {}".format(round(np.mean(recall_lst)*100)),"%")
print("f1: {}".format(round(np.mean(f1_lst)*100)),"%")

```

And the result are:

```
accuracy: 93 %
precision: 96 %
recall: 89 %
f1: 92 %
```

2. Alternative way to implement

The alternative way is to use as an array method with library to use the Oversampling method

```
from imblearn.over_sampling import SMOTE, RandomOverSampler, ADASYN
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
```

Import all the library for the oversampling the method

After import all the method sampling, the next step is to clustered in an object:

```
oversamplers = {
    "SMOTE": SMOTE(random_state=42),
    "RandomOverSampler": RandomOverSampler(random_state=42),
    "ADASYN": ADASYN(random_state=42)
}
```

And put aside with the modeler:

```
models = {
    "Logistic Regression": LogisticRegression(),
    "Decision Tree": DecisionTreeClassifier(),
    "K-Nearest Neighbors": KNeighborsClassifier()
}
```

From this one, we'll import all with a for loop and run those model to get the classification report

1. **SMOTE Oversampling Method**

   a. Result Logistic Regression with SMOTE Oversampling:

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.92      0.95      0.93        98
           1       0.95      0.92      0.93        98

    accuracy                           0.93       196
   macro avg       0.93      0.93      0.93       196
weighted avg       0.93      0.93      0.93       196
```

The result for using the Logistic Regression are 93%

   b. Result Decision Tree with SMOTE Oversampling

```
Decision Tree:
              precision    recall  f1-score   support

           0       0.93      0.89      0.91        98
           1       0.89      0.93      0.91        98

    accuracy                           0.91       196
   macro avg       0.91      0.91      0.91       196
weighted avg       0.91      0.91      0.91       196
```

The result for using the Decision Tree are 91%

   c. Result K-Nearest Neighbors with SMOTE Oversampling

```
K-Nearest Neighbors:
              precision    recall  f1-score   support

           0       0.93      0.96      0.94        98
           1       0.96      0.93      0.94        98

    accuracy                           0.94       196
   macro avg       0.94      0.94      0.94       196
weighted avg       0.94      0.94      0.94       196
```

The result for using the K-Nearest Neighbors are 94%

2. Using Random Over-sampler Method (ROS)

a. Result Logistic Regression with ROS:

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.92      0.95      0.93        98
           1       0.95      0.92      0.93        98

    accuracy                           0.93       196
   macro avg       0.93      0.93      0.93       196
weighted avg       0.93      0.93      0.93       196
```

b. Result with Decision Tree with ROS

```
Decision Tree:
              precision    recall  f1-score   support

           0       0.93      0.89      0.91        98
           1       0.89      0.93      0.91        98

    accuracy                           0.91       196
   macro avg       0.91      0.91      0.91       196
weighted avg       0.91      0.91      0.91       196
```

c. Result with K-Nearest Neighbors with ROS:

```
K-Nearest Neighbors:
              precision    recall  f1-score   support

           0       0.93      0.96      0.94        98
           1       0.96      0.93      0.94        98

    accuracy                           0.94       196
   macro avg       0.94      0.94      0.94       196
weighted avg       0.94      0.94      0.94       196
```

3. Using ADASYN:

a. Result Logistic Regression with ADASYN:

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.92      0.95      0.93        98
           1       0.95      0.92      0.93        98

    accuracy                           0.93       196
   macro avg       0.93      0.93      0.93       196
weighted avg       0.93      0.93      0.93       196
```

b. Result Decision Tree Method with ADASYN

```
Decision Tree:
              precision    recall  f1-score   support

           0       0.93      0.91      0.92        98
           1       0.91      0.93      0.92        98

    accuracy                           0.92       196
   macro avg       0.92      0.92      0.92       196
weighted avg       0.92      0.92      0.92       196
```

c. Result K-Nearest Neighbors with ADASYN:

```
K-Nearest Neighbors:
              precision    recall  f1-score   support

           0       0.93      0.96      0.94        98
           1       0.96      0.93      0.94        98

    accuracy                           0.94       196
   macro avg       0.94      0.94      0.94       196
weighted avg       0.94      0.94      0.94       196
```

References

1 https://blog.tomorrowmarketers.org/thuat-toan-phan-loai-classification-machine-learning/

2 https://www.kaggle.com/code/chanchal24/credit-card-fraud-detection

3 https://www.kaggle.com/code/gpreda/credit-card-fraud-detection-predictive-models