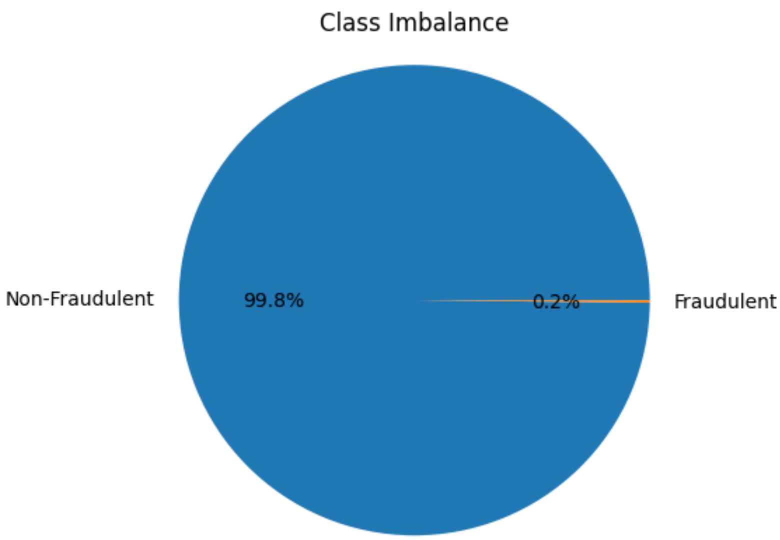
Credit Card Fraud Detection

Dataset link : https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

> Anonymized credit card transactions labeled as fraudulent or genuine.

Content

- 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'.
- 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.
- Given the class imbalance ratio, we recommend measuring the accuracy using the Area Under the Precision-Recall Curve (AUPRC). Confusion matrix accuracy is not meaningful for unbalanced classification.



Acknowledgements :

- The dataset has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (Université Libre de Bruxelles) on big data mining and fraud detection.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

pd.options.display.max_rows = None
pd.options.display.max_columns = None
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
df = pd.read_csv("C:\\Users\\Saikrupa\\Documents\\DATASETS\\creditcard-classification\\creditcard.csv")
df.head()
```

Out[2]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13
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
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
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
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
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

as they have already mentioned in the description of this dataset.... this doesn't make any sense to us as it is the output from Principal component analysis.

In [3]:

```
def explore_data(df):
    # Get the basic information about the DataFrame
    print("Data Shape:")
    print(df.shape)

    print("\nData Head:")
    print(df.head())

    print("\nData Columns:")
    print(df.columns)

    print("\nData Info:")
    print(df.info())

    print("\nData Summary:")
    print(df.describe())

    # Check for missing values
    print("\nMissing Values:")
    print(df.isnull().sum())

    # Check for duplicate rows
    print("\nDuplicate Rows:")
    print(df.duplicated().sum())

    # Explore unique values in categorical columns
    categorical_cols = df.select_dtypes(include=["object"]).columns
    if len(categorical_cols) > 0:
        print("\nUnique Values in Categorical Columns:")
        for col in categorical_cols:
            print(f"\n{col}:")
            print(df[col].unique())

    # Explore numerical columns
    numerical_cols = df.select_dtypes(include=["int64", "float64"]).columns
    if len(numerical_cols) > 0:
        print("\nNumerical Column Statistics:")
        for col in numerical_cols:
            print(f"\n{col}:")
            print(f"Minimum: {df[col].min()}")
            print(f"Maximum: {df[col].max()}")
            print(f"Mean: {df[col].mean()}")
            print(f"Median: {df[col].median()}")
            print(f"Standard Deviation: {df[col].std()}")

    # Explore datetime columns
    datetime_cols = df.select_dtypes(include=["datetime64"]).columns
    if len(datetime_cols) > 0:
        print("\nDatetime Column Statistics:")
        for col in datetime_cols:
            print(f"\n{col}:")
            print(f"Minimum Date: {df[col].min()}")
            print(f"Maximum Date: {df[col].max()}")

    # Visualize the DataFrame (optional)
    # You can add code here to generate plots or visualizations

explore_data(df)
```

```
Data Shape:
(284807, 31)

Data Head:
   Time        V1        V2        V3        V4        V5        V6        V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9       V10       V11       V12       V13       V14  \
0  0.098698  0.363787  0.090794 -0.551600 -0.617801 -0.991390 -0.311169
1  0.085102 -0.255425 -0.166974  1.612727  1.065235  0.489095 -0.143772
2  0.247676 -1.514654  0.207643  0.624501  0.066084  0.717293 -0.165946
3  0.377436 -1.387024 -0.054952 -0.226487  0.178228  0.507757 -0.287924
4 -0.270533  0.817739  0.753074 -0.822843  0.538196  1.345852 -1.119670

        V15       V16       V17       V18       V19       V20       V21  \
0  1.468177 -0.470401  0.207971  0.025791  0.403993  0.251412 -0.018307
1  0.635558  0.463917 -0.114805 -0.183361 -0.145783 -0.069083 -0.225775
2  2.345865 -2.890083  1.109969 -0.121359 -2.261857  0.524980  0.247998
3 -0.631418 -1.059647 -0.684093  1.965775 -1.232622 -0.208038 -0.108300
4  0.175121 -0.451449 -0.237033 -0.038195  0.803487  0.408542 -0.009431

        V22       V23       V24       V25       V26       V27       V28  \
0  0.277838 -0.110474  0.066928  0.128539 -0.189115  0.133558 -0.021053
1 -0.638672  0.101288 -0.339846  0.167170  0.125895 -0.008983  0.014724
2  0.771679  0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752
3  0.005274 -0.190321 -1.175575  0.647376 -0.221929  0.062723  0.061458
4  0.798278 -0.137458  0.141267 -0.206010  0.502292  0.219422  0.215153

   Amount  Class
0  149.62      0
1    2.69      0
2  378.66      0
3  123.50      0
4   69.99      0

Data Columns:
Index(['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'],
      dtype='object')


Data Summary:
                Time            V1            V2            V3            V4  \
count  284807.000000  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean    94813.859575  1.168375e-15  3.416908e-16 -1.379537e-15  2.074095e-15
std     47488.145955  1.958696e+00  1.651309e+00  1.516255e+00  1.415869e+00
min         0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00
25%     54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01
50%     84692.000000  1.810880e-02  6.548556e-02  1.798463e-01 -1.984653e-02
75%    139320.500000  1.315642e+00  8.037239e-01  1.027196e+00  7.433413e-01
max    172792.000000  2.454930e+00  2.205773e+01  9.382558e+00  1.687534e+01

                 V5            V6            V7            V8            V9  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   9.604066e-16  1.487313e-15 -5.556467e-16  1.213481e-16 -2.406331e-15
std    1.380247e+00  1.332271e+00  1.237094e+00  1.194353e+00  1.098632e+00
min   -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01
25%   -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01
50%   -5.433583e-02 -2.741871e-01  4.010308e-02  2.235804e-02 -5.142873e-02
75%    6.119264e-01  3.985649e-01  5.704361e-01  3.273459e-01  5.971390e-01
max    3.480167e+01  7.330163e+01  1.205895e+02  2.000721e+01  1.559499e+01

                V10           V11           V12           V13           V14  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   2.239053e-15  1.673327e-15 -1.247012e-15  8.190001e-16  1.207294e-15
std    1.088850e+00  1.020713e+00  9.992014e-01  9.952742e-01  9.585956e-01
min   -2.458826e+01 -4.797473e+00 -1.868371e+01 -5.791881e+00 -1.921433e+01
25%   -5.354257e-01 -7.624942e-01 -4.055715e-01 -6.485393e-01 -4.255740e-01
50%   -9.291738e-02 -3.275735e-02  1.400326e-01 -1.356806e-02  5.060132e-02
75%    4.539234e-01  7.395934e-01  6.182380e-01  6.625050e-01  4.931498e-01
max    2.374514e+01  1.201891e+01  7.848392e+00  7.126883e+00  1.052677e+01

                V15           V16           V17           V18           V19  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   4.887456e-15  1.437716e-15 -3.772171e-16  9.564149e-16  1.039917e-15
std    9.153160e-01  8.762529e-01  8.493371e-01  8.381762e-01  8.140405e-01
min   -4.498945e+00 -1.412985e+01 -2.516280e+01 -9.498746e+00 -7.213527e+00
25%   -5.828843e-01 -4.680368e-01 -4.837483e-01 -4.988498e-01 -4.562989e-01
50%    4.807155e-02  6.641332e-02 -6.567575e-02 -3.636312e-03  3.734823e-03
75%    6.488208e-01  5.232963e-01  3.996750e-01  5.008067e-01  4.589494e-01
```

```
max    8.877742e+00  1.731511e+01  9.253526e+00  5.041069e+00  5.591971e+00

                V20           V21           V22           V23           V24  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   6.406204e-16  1.654067e-16 -3.568593e-16  2.578648e-16  4.473266e-15
std    7.709250e-01  7.345240e-01  7.257016e-01  6.244603e-01  6.056471e-01
min   -5.449772e+01 -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00
25%   -2.117214e-01 -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01
50%   -6.248109e-02 -2.945017e-02  6.781943e-03 -1.119293e-02  4.097606e-02
75%    1.330408e-01  1.863772e-01  5.285536e-01  1.476421e-01  4.395266e-01
max    3.942090e+01  2.720284e+01  1.050309e+01  2.252841e+01  4.584549e+00

                V25           V26           V27           V28         Amount  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  284807.000000
mean   5.340915e-16  1.683437e-15 -3.660091e-16 -1.227390e-16      88.349619
std    5.212781e-01  4.822270e-01  4.036325e-01  3.300833e-01     250.120109
min   -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01       0.000000
25%   -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02       5.600000
50%    1.659350e-02 -5.213911e-02  1.342146e-03  1.124383e-02      22.000000
75%    3.507156e-01  2.409522e-01  9.104512e-02  7.827995e-02      77.165000
max    7.519589e+00  3.517346e+00  3.161220e+01  3.384781e+01   25691.160000

               Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000



Duplicate Rows:
1081
```

In [4]:

```
print("Original number of rows:", len(df))
df = df.drop_duplicates()
print("Number of rows after removing duplicates:", len(df))
```

```
Original number of rows: 284807
Number of rows after removing duplicates: 283726
```

In [5]:

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, MaxAbsScaler
import numpy as np

def scale_columns(dataframe):
    # Check if scaling is needed
    scale_needed = False
    features_to_scale = []
    for col in dataframe.columns:
        if dataframe[col].dtype in [np.float64, np.float32, np.int64, np.int32]:
            mean = dataframe[col].mean()
            std = dataframe[col].std()
            if np.abs(mean) > 1 or np.abs(std) > 1:
                scale_needed = True
                features_to_scale.append((col, mean, std))

    if not scale_needed:
        print("No scaling needed.")
        return

    # Prompt the user for scaling option
    print("Scaling is recommended for the following reasons:")
    for feature in features_to_scale:
        print(f"- Feature: {feature[0]}, Mean: {feature[1]}, Standard Deviation: {feature[2]}")
    print("- Data has numerical features with mean or standard deviation larger than 1.")
    print("Available scaling options:")
    print("1. Min-Max Scaling")
    print("2. Standard Scaling")
    print("3. Robust Scaling")
    print("4. Max Abs Scaling")
    print("5. Log Transformation")
    print("6. Power Transformation")
    scaling_option = int(input("Enter the scaling option number (0 to skip scaling): "))

    if scaling_option == 0:
        return dataframe

    # Prompt the user to select columns to scale
    print("\nAvailable numerical columns:")
    numerical_columns = dataframe.select_dtypes(include=[np.float64, np.float32, np.int64, np.int32]).columns
    for col in numerical_columns:
        print(col)
```

```
    column_choice = input("Enter 'all' to select all columns or provide a comma-separated list of column names to scale: ")

    if column_choice.lower() == 'all':
        columns = numerical_columns
    else:
        columns = [col.strip() for col in column_choice.split(",")]

    scaled_data = dataframe.copy()

    if scaling_option == 1:
        scaler = MinMaxScaler()
        scaled_data[columns] = scaler.fit_transform(scaled_data[columns])
    elif scaling_option == 2:
        scaler = StandardScaler()
        scaled_data[columns] = scaler.fit_transform(scaled_data[columns])
    elif scaling_option == 3:
        scaler = RobustScaler()
        scaled_data[columns] = scaler.fit_transform(scaled_data[columns])
    elif scaling_option == 4:
        scaler = MaxAbsScaler()
        scaled_data[columns] = scaler.fit_transform(scaled_data[columns])
    elif scaling_option == 5:
        scaled_data[columns] = np.log(scaled_data[columns])
    elif scaling_option == 6:
        power = float(input("Enter the power for power transformation: "))
        scaled_data[columns] = np.power(scaled_data[columns], power)
    else:
        print("Invalid scaling option selected.")

    return scaled_data

df = scale_columns(df)
df.head()
```

```
Scaling is recommended for the following reasons:
- Feature: Time, Mean: 94811.07759951502, Standard Deviation: 47481.047890619506
- Feature: V1, Mean: 0.005917149836165761, Standard Deviation: 1.948026141625471
- Feature: V2, Mean: -0.0041347556281216905, Standard Deviation: 1.6467029642463507
- Feature: V3, Mean: 0.0016131193558786181, Standard Deviation: 1.5086819162059164
- Feature: V4, Mean: -0.0029663077203488635, Standard Deviation: 1.4141840144475144
- Feature: V5, Mean: 0.0018275601130338598, Standard Deviation: 1.3770082792800886
- Feature: V6, Mean: -0.001139488189738493, Standard Deviation: 1.331930591715164
- Feature: V7, Mean: 0.0018006917653071734, Standard Deviation: 1.2276638954422558
- Feature: V8, Mean: -0.0008544525734540372, Standard Deviation: 1.1790544275788069
- Feature: V9, Mean: -0.0015961996217021513, Standard Deviation: 1.0954924810736155
- Feature: V10, Mean: -0.0014407104850314848, Standard Deviation: 1.0764073501381102
- Feature: V11, Mean: 0.00020175763995932604, Standard Deviation: 1.0187201526753205
- Feature: Amount, Mean: 88.47268731099724, Standard Deviation: 250.39943711577337
- Data has numerical features with mean or standard deviation larger than 1.
Available scaling options:
1. Min-Max Scaling
2. Standard Scaling
3. Robust Scaling
4. Max Abs Scaling
5. Log Transformation
6. Power Transformation
Enter the scaling option number (0 to skip scaling): 1

Available numerical columns:
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
Enter 'all' to select all columns or provide a comma-separated list of column names to scale: all
```

Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	
0	0.000000	0.935192	0.766490	0.881365	0.313023	0.763439	0.267669	0.266815	0.786444	0.475312	0.510600	0.252484	0.680908	0.371591	0.63
1	0.000000	0.978542	0.770067	0.840298	0.271796	0.766120	0.262192	0.264875	0.786298	0.453981	0.505267	0.381188	0.744342	0.486190	0.64
2	0.000006	0.935217	0.753118	0.868141	0.268766	0.762329	0.281122	0.270177	0.788042	0.410603	0.513018	0.322422	0.706683	0.503854	0.64
3	0.000006	0.941878	0.765304	0.868484	0.213661	0.765647	0.275559	0.266803	0.789434	0.414999	0.507585	0.271817	0.710910	0.487635	0.63
4	0.000012	0.938617	0.776520	0.864251	0.269796	0.762975	0.263984	0.268968	0.782484	0.490950	0.524303	0.236355	0.724477	0.552509	0.60

In [6]:

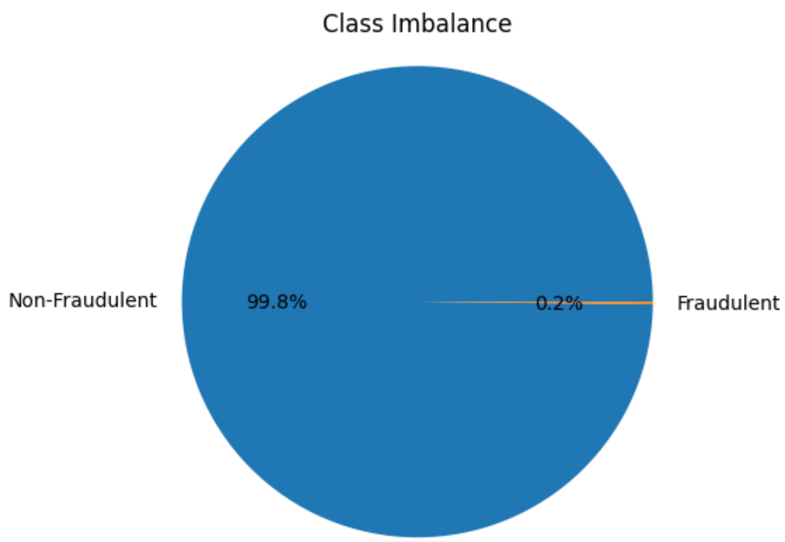
```
scale_columns(df)
```

```
No scaling needed.
```

now lets see the Imbalance in the Target variable...

In [7]:

```
import matplotlib.pyplot as plt

labels = ['Non-Fraudulent', 'Fraudulent']
counts = [df['Class'].value_counts()[0], df['Class'].value_counts()[1]]
plt.pie(counts, labels=labels, autopct='%1.1f%%', startangle=0)
plt.axis('equal')
plt.title('Class Imbalance')
plt.show()
```



Should we handle the Imbalance or NOT ?

> - **what if we don't handle class imbalance ?**

> - are we not training the model, as it happens in the real world ?

> - isn't the real world scenario biased ? like see the current data which is so biased ???

XGBClassifier to handle class imbalance

We can also use specialized algorithms that have built-in features or parameters to handle imbalanced data. XGBoost is one such algorithm that provides options for addressing class imbalance.

Specifically, if we can use the scale_pos_weight parameter in XGBoost to handle class imbalance.

This parameter helps in assigning higher weights to the minority class during the training process.

In [8]:
```
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
xgb = XGBClassifier(scale_pos_weight=(len(y_train) - y_train.sum()) / y_train.sum())
xgb.fit(X_train, y_train)

y_pred = xgb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("XGBoost Model Performance:")
print("Accuracy:", accuracy)
print("Classification Report:")
print(report)
```

```
XGBoost Model Performance:
Accuracy: 0.9995241955380115
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     56656
         1.0       0.93      0.76      0.83        90

    accuracy                           1.00     56746
   macro avg       0.97      0.88      0.92     56746
weighted avg       1.00      1.00      1.00     56746
```

Inferences that can be drawn from the XGBoost model performance:

- The model achieved a high accuracy of 0.9995, indicating that it is able to correctly classify the majority of the instances in the dataset.

- The precision for class 0 (non-fraudulent transactions) is perfect, indicating that all the predicted non-fraudulent transactions are indeed non-fraudulent.
- The precision for class 1 (fraudulent transactions) is 0.93, suggesting that around 93% of the predicted fraudulent transactions are correctly classified.
- The recall (also known as sensitivity or true positive rate) for class 1 is 0.76, indicating that the model is able to identify approximately 76% of the actual fraudulent transactions.
- The F1-score for class 1 is 0.83, which is a measure of the model's balance between precision and recall for class 1.
- The support column shows the number of instances in each class. In this case, there are 56656 instances of class 0 (non-fraudulent) and 90 instances of class 1 (fraudulent).
- The macro average F1-score is 0.92, which represents the overall performance of the model across both classes, giving equal weight to each class.
- The weighted average F1-score is also 0.92, but it takes into account the class imbalance by considering the number of instances in each class during averaging.

> Overall, the model demonstrates strong performance in accurately classifying non-fraudulent transactions. However, there is room for improvement in identifying fraudulent transactions, as indicated by lower recall and F1-score for class 1. Further analysis and adjustment of the model's parameters or considering techniques like oversampling or undersampling may help improve its performance for the minority class.

And now may be, Let us handle the Class Imbalance using Random Under Sampling...

In [9]:
```
import pandas as pd
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import ADASYN

def examine_dataset(df, target_column):
    """
    Examine the dataset to check if there is an imbalance in the target column.

    Parameters:
        - df (pandas DataFrame): The input DataFrame.
        - target_column (str): The name of the target column.

    Returns:
        - bool: True if the dataset is imbalanced, False otherwise.
    """
    class_counts = df[target_column].value_counts()
    imbalance_ratio = class_counts.iloc[0] / class_counts.iloc[1]
    print("Class Distribution:")
    print(class_counts)
    print("Imbalance Ratio:", imbalance_ratio)
    return imbalance_ratio > 2.0

def handle_imbalanced_data(df, target_column):
    """
    Handle imbalanced pandas DataFrame based on user-selected option.

    Parameters:
        - df (pandas DataFrame): The input DataFrame.
        - target_column (str): The name of the target column.

    Returns:
        - pandas DataFrame: The balanced DataFrame.
    """
    imbalance = examine_dataset(df, target_column)
    if not imbalance:
        print("No imbalance found in the dataset.")
        return

    print("Select an option to handle the imbalanced dataset:")
    print("1. Random Oversampling")
    print("2. Random Undersampling")
    print("3. SMOTE (Synthetic Minority Over-sampling Technique)")
    print("4. ADASYN (Adaptive Synthetic)")
    print("5. Proceed without handling")
    choice = input("Enter your choice (1-5): ")

    # Separate features and target variable
    X = df.drop(target_column, axis=1)
    y = df[target_column]

    if choice == '1':
```

```
        # Apply random oversampling
        oversampler = RandomOverSampler()
        X_resampled, y_resampled = oversampler.fit_resample(X, y)
    elif choice == '2':
        # Apply random undersampling
        undersampler = RandomUnderSampler()
        X_resampled, y_resampled = undersampler.fit_resample(X, y)
    elif choice == '3':
        # Apply SMOTE
        oversampler = SMOTE()
        X_resampled, y_resampled = oversampler.fit_resample(X, y)
    elif choice == '4':
        # Apply ADASYN
        oversampler = ADASYN()
        X_resampled, y_resampled = oversampler.fit_resample(X, y)
    elif choice == '5':
        # Proceed without handling
        print("Proceeding without handling the imbalanced dataset.")
        return df
    else:
        print("Invalid choice. Proceeding without handling the imbalanced dataset.")
        return df

    # Create a new balanced DataFrame
    balanced_df = pd.concat([X_resampled, y_resampled], axis=1)

    return balanced_df


df = handle_imbalanced_data(df, 'Class')
df.head()
```

```
Class Distribution:
0.0    283253
1.0       473
Name: Class, dtype: int64
Imbalance Ratio: 598.8435517970402
Select an option to handle the imbalanced dataset:
1. Random Oversampling
2. Random Undersampling
3. SMOTE (Synthetic Minority Over-sampling Technique)
4. ADASYN (Adaptive Synthetic)
5. Proceed without handling
Enter your choice (1-5): 2
```

Out[9]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	
0	0.971307	0.878885	0.816146	0.802221	0.178457	0.765046	0.247672	0.273601	0.778833	0.612346	0.651383	0.279824	0.718421	0.507478	0.58
1	0.261378	0.977004	0.766658	0.857762	0.309294	0.758913	0.258647	0.262265	0.786019	0.482115	0.508806	0.252416	0.698917	0.414842	0.64
2	0.906576	0.945481	0.783251	0.824064	0.243778	0.768718	0.248887	0.270570	0.785549	0.453935	0.489719	0.254110	0.711783	0.502423	0.61
3	0.700374	0.992061	0.767966	0.809338	0.267802	0.768230	0.256427	0.266049	0.784201	0.469909	0.504179	0.363355	0.739699	0.450885	0.62
4	0.253467	0.935001	0.777917	0.872331	0.291658	0.753736	0.275875	0.268961	0.791807	0.447182	0.496648	0.361221	0.739513	0.454688	0.65

In [10]:
```
handle_imbalanced_data(df, 'Class')
```

```
Class Distribution:
0.0    473
1.0    473
Name: Class, dtype: int64
Imbalance Ratio: 1.0
No imbalance found in the dataset.
```

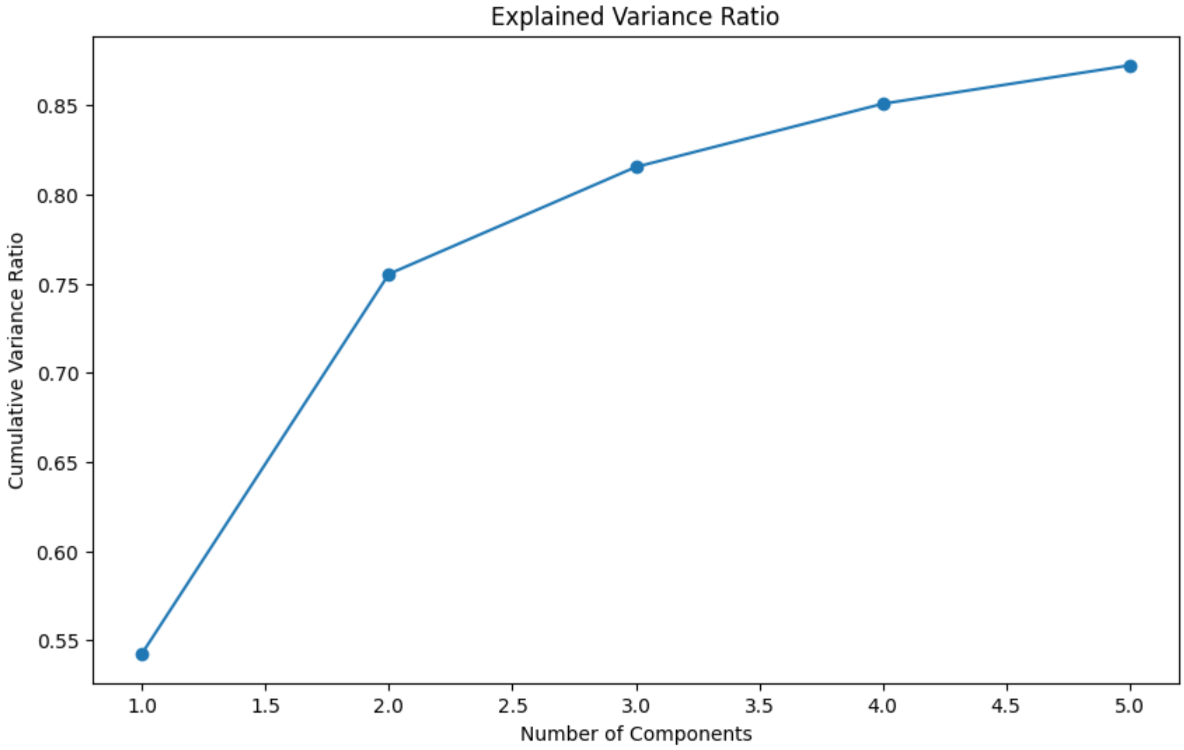
now that the imbalance is taken care of let us proceed to next step - feature importance !!!

In [11]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import Lasso
from sklearn.decomposition import PCA

def perform_feature_selection(dataframe):
    num_features = int(input("Enter the number of features to select: "))

    print("Available feature selection algorithms:")
    print("1. Recursive Feature Elimination (RFE)")
    print("2. Feature Importance (FI)")
    print("3. Lasso Regularization (L1 Regularization)")
    print("4. Principal Component Analysis (PCA)")
    print("5. Correlation Analysis")
    print("6. Proceed without selecting any algorithm")
```

```
choice = input("Enter the number corresponding to the algorithm you want to use (or enter 6 to proceed without selectio

if choice == "1":  # Recursive Feature Elimination (RFE)
    target_variable = input("Enter the name of the target variable: ")

    X = dataframe.drop(target_variable, axis=1)
    y = dataframe[target_variable]

    model = RandomForestClassifier()  # Replace with your desired model
    rfe = RFE(estimator=model, n_features_to_select=num_features)
    selected_features = rfe.fit_transform(X, y)

    selected_columns = X.columns[rfe.support_].tolist()

    # Plotting feature ranking
    feature_ranking = pd.Series(rfe.ranking_, index=X.columns)
    feature_ranking = feature_ranking.sort_values(ascending=True)

    plt.figure(figsize=(10, 6))
    sns.barplot(x=feature_ranking.values, y=feature_ranking.index)
    plt.xlabel('Rank')
    plt.ylabel('Features')
    plt.title('Feature Ranking')
    plt.show()

    return selected_columns

elif choice == "2":  # Feature Importance (FI)
    target_variable = input("Enter the name of the target variable: ")

    X = dataframe.drop(target_variable, axis=1)
    y = dataframe[target_variable]

    model = RandomForestClassifier()  # Replace with your desired model
    model.fit(X, y)

    importance_scores = model.feature_importances_
    selected_columns = X.columns[importance_scores.argsort()[-num_features:]].tolist()

    # Plotting feature importance
    feature_importance = pd.Series(importance_scores, index=X.columns)
    feature_importance = feature_importance.sort_values(ascending=False)

    plt.figure(figsize=(10, 6))
    sns.barplot(x=feature_importance.values, y=feature_importance.index)
    plt.xlabel('Importance Score')
    plt.ylabel('Features')
    plt.title('Feature Importance')
    plt.show()

    return selected_columns

elif choice == "3":  # Lasso Regularization (L1 Regularization)
    target_variable = input("Enter the name of the target variable: ")

    X = dataframe.drop(target_variable, axis=1)
    y = dataframe[target_variable]

    lasso = Lasso()
    lasso.fit(X, y)

    selected_columns = X.columns[lasso.coef_ != 0].tolist()

    return selected_columns

elif choice == "4":  # Principal Component Analysis (PCA)
    target_variable = input("Enter the name of the target variable: ")

    X = dataframe.drop(target_variable, axis=1)
    y = dataframe[target_variable]

    pca = PCA(n_components=num_features)
    selected_features = pca.fit_transform(X)

    selected_columns = ['PCA_Component_' + str(i+1) for i in range(num_features)]

    # Explained variance ratio plot
    explained_variance_ratio = pca.explained_variance_ratio_
    cumulative_variance_ratio = np.cumsum(explained_variance_ratio)

    print(f"\nexplained variance ratio : {explained_variance_ratio}")
    print(f"\ncumulative variance ratio: {cumulative_variance_ratio}")

    plt.figure(figsize=(10, 6))
    plt.plot(range(1, num_features+1), cumulative_variance_ratio, marker='o')
```

```
        plt.xlabel('Number of Components')
        plt.ylabel('Cumulative Variance Ratio')
        plt.title('Explained Variance Ratio')
        plt.show()

        return selected_features

    elif choice == "5":  # Correlation Analysis
        target_variable = input("Enter the name of the target variable: ")
        threshold = float(input("Enter the correlation threshold (between 0 and 1): "))

        X = dataframe.drop(target_variable, axis=1)
        y = dataframe[target_variable]

        corr_matrix = dataframe.corr()
        selected_columns = corr_matrix[target_variable][abs(corr_matrix[target_variable]) > threshold].index.tolist()

        # Heatmap of correlations
        plt.figure(figsize=(10, 8))
        sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
        plt.title('Correlation Heatmap')
        plt.show()

        return selected_columns

    elif choice == "6":  # Proceed without selecting any algorithm
        return dataframe.columns.tolist()

    else:
        print("Invalid choice. Please enter a valid option.")
        return []


Selection = perform_feature_selection(df)
```

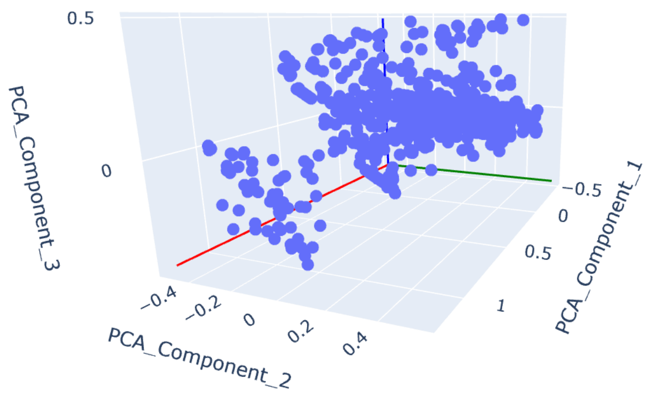
```
Enter the number of features to select: 5
Available feature selection algorithms:
1. Recursive Feature Elimination (RFE)
2. Feature Importance (FI)
3. Lasso Regularization (L1 Regularization)
4. Principal Component Analysis (PCA)
5. Correlation Analysis
6. Proceed without selecting any algorithm
Enter the number corresponding to the algorithm you want to use (or enter 6 to proceed without selection): 4
Enter the name of the target variable: Class

explained variance ratio : [0.54258507 0.21287048 0.06000881 0.03546485 0.0215647 ]

cumulative variance ratio: [0.54258507 0.75545555 0.81546436 0.85092921 0.87249391]
```



In [12]:
```
import plotly.graph_objects as go

x = Selection[:, 0]
y = Selection[:, 1]
```

```python
z = Selection[:, 2]

# Create the scatter trace
scatter_trace = go.Scatter3d(
    x=x,
    y=y,
    z=z,
    mode='markers',
    marker=dict(size=5),
    name='Data Points'
)

# Calculate the minimum and maximum values for each component
x_min, x_max = x.min(), x.max()
y_min, y_max = y.min(), y.max()
z_min, z_max = z.min(), z.max()

# Create the center line traces
center_line_trace1 = go.Scatter3d(
    x=[x_min, x_max],
    y=[y_min, y_min],
    z=[z_min, z_min],
    mode='lines',
    line=dict(color='red', width=3),
    name='Center Line - Component 1'
)

center_line_trace2 = go.Scatter3d(
    x=[x_min, x_min],
    y=[y_min, y_max],
    z=[z_min, z_min],
    mode='lines',
    line=dict(color='green', width=3),
    name='Center Line - Component 2'
)

center_line_trace3 = go.Scatter3d(
    x=[x_min, x_min],
    y=[y_min, y_min],
    z=[z_min, z_max],
    mode='lines',
    line=dict(color='blue', width=3),
    name='Center Line - Component 3'
)

# Create the figure and add the traces
fig = go.Figure(data=[scatter_trace, center_line_trace1, center_line_trace2, center_line_trace3])

# Set axis labels and title
fig.update_layout(
    scene=dict(
        xaxis_title='PCA_Component_1',
        yaxis_title='PCA_Component_2',
        zaxis_title='PCA_Component_3'
    ),
    title='Data in Reduced Dimensional Space'
)

# Show the interactive plot
fig.show()
```

Data in Reduced Dimensional Space



Having looked at the components.... Lets now build our model and put these principle components to use

ML Model Pipeline...

In [13]:

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score, classification_report

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)


# Define the pipeline with model selection
pipeline = Pipeline([
    ('model', SVC())
])

# Define the hyperparameters to tune for each model
parameters = [
    {
        'model': [SVC()],
        'model__C': [0.1, 1, 10],
        'model__kernel': ['linear', 'rbf']
    },
    {
        'model': [RandomForestClassifier()],
        'model__n_estimators': [100, 200, 300],
        'model__max_depth': [None, 5, 10]
    },
    {
        'model': [LogisticRegression()],
        'model__C': [0.1, 1, 10],
        'model__solver': ['liblinear', 'lbfgs']
    },
    {
        'model': [XGBClassifier()],
```

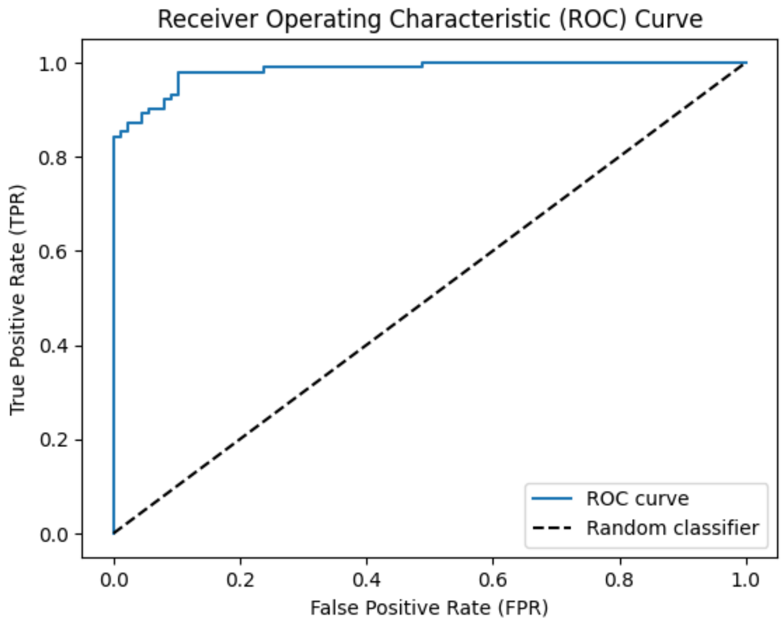
```
        'model__learning_rate': [0.1, 0.01],
        'model__max_depth': [3, 5, 7],
        'model__n_estimators': [100, 200]
    },
    {
        'model': [LGBMClassifier()],
        'model__learning_rate': [0.1, 0.01],
        'model__max_depth': [3, 5, 7],
        'model__n_estimators': [100, 200]
    }
]

# Perform grid search to find the best model and hyperparameters
grid_search = GridSearchCV(pipeline, parameters, scoring='accuracy', cv=5)
grid_search.fit(X_train, y_train)

# Get the results for all models
results = grid_search.cv_results_

"""
# Print the performance details for all models
for mean_score, params in zip(results["mean_test_score"], results["params"]):
    print("Model:", params)
    print("Mean Accuracy:", mean_score)
    print("-" * 50)

"""

# Print the best model and its hyperparameters
print("Best Model: ", grid_search.best_estimator_)
print("Best Hyperparameters: ", grid_search.best_params_)
print("-" * 50)

# Use the best model for predictions
y_pred = grid_search.predict(X_test)

# Evaluate the best model's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Best Model Performance:")
print("Accuracy:", accuracy)
print("Classification Report:")
print(report)
```

```
X_train shape: (756, 30)
y_train shape: (756,)
X_test shape: (190, 30)
y_test shape: (190,)
Best Model:  Pipeline(steps=[('model',
                 XGBClassifier(base_score=None, booster=None, callbacks=None,
                               colsample_bylevel=None, colsample_bynode=None,
                               colsample_bytree=None,
                               early_stopping_rounds=None,
                               enable_categorical=False, eval_metric=None,
                               feature_types=None, gamma=None, gpu_id=None,
                               grow_policy=None, importance_type=None,
                               interaction_constraints=None, learning_rate=0.1,
                               max_bin=None, max_cat_threshold=None,
                               max_cat_to_onehot=None, max_delta_step=None,
                               max_depth=3, max_leaves=None,
                               min_child_weight=None, missing=nan,
                               monotone_constraints=None, n_estimators=200,
                               n_jobs=None, num_parallel_tree=None,
                               predictor=None, random_state=None, ...))])
Best Hyperparameters:  {'model': XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=3, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=200, n_jobs=None, num_parallel_tree=None,
              predictor=None, random_state=None, ...), 'model__learning_rate': 0.1, 'model__max_depth': 3, 'model__n_estima
tors': 200}
--------------------------------------------------
Best Model Performance:
Accuracy: 0.9210526315789473
Classification Report:
              precision    recall  f1-score   support

         0.0       0.91      0.92      0.92        88
         1.0       0.93      0.92      0.93       102

    accuracy                           0.92       190
   macro avg       0.92      0.92      0.92       190
weighted avg       0.92      0.92      0.92       190
```

In [14]:

```python
from sklearn.metrics import roc_curve, roc_auc_score

y_pred_prob = grid_search.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
auc_roc = roc_auc_score(y_test, y_pred)
print(f"auc_roc: {auc_roc}")

plt.plot(fpr, tpr, label='ROC curve')
plt.plot([0, 1], [0, 1], 'k--', label='Random classifier')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```

```
auc_roc: 0.9210115864527628
```



Here are some inferences that can be drawn from the output of the best model:

- The best model used in the pipeline is an XGBoost classifier.
- The hyperparameters that resulted in the best model are:
- Learning rate: 0.1
- Maximum depth: 3
- Number of estimators: 200
- The best model achieved an accuracy of 0.921, indicating that it correctly classified approximately 92.1% of the instances.
- The precision for both class 0 and class 1 is high, with values of 0.91 and 0.93, respectively.
- The recall (sensitivity) for both class 0 and class 1 is also high, with values of 0.92 for both classes.
- The F1-score for both classes is also high, indicating a good balance between precision and recall.
- The support column shows the number of instances in each class, with 88 instances of class 0 and 102 instances of class 1.
- The macro average F1-score is 0.92, indicating the overall performance of the model across both classes, giving equal weight to each class.
- The weighted average F1-score is also 0.92, taking into account the class imbalance by considering the number of instances in each class during averaging.
- The AUC-ROC (Area Under the Receiver Operating Characteristic Curve) is 0.921, which is a measure of the model's ability to distinguish between the positive and negative classes.

> Overall, the best model demonstrates strong performance with high accuracy, precision, recall, and F1-score for both classes. It suggests that the model is effective in classifying instances in this dataset.

Model Comparision

	Model 1 (Before handling class imbalance):	**Model 2 (After handling class imbalance):**
Accuracy	0.9995	0.921
Precision (Class 0)	1.00	0.91
Precision (Class 1)	0.93	0.93
Recall (Class 0)	1.00	0.92
Recall (Class 1)	0.76	0.92
F1-Score (Class 0)	1.00	0.92
F1-Score (Class 1)	0.83	0.93

Based on these metrics, we can make the following observations:

> Accuracy:

- Model 1 achieved a higher accuracy of 0.9995 compared to 0.921 in Model 2. However, it's important to note that Model 1 may be overly optimistic due to the class imbalance.

> Precision:

- Model 1 had perfect precision for class 0 (non-fraudulent transactions) while Model 2 had a slightly lower precision of 0.91. However, the precision for class 1 (fraudulent transactions) was the same in both models at 0.93.

> Recall:

- Model 1 had a lower recall of 0.76 for class 1, indicating that it missed a significant number of actual fraudulent transactions. In contrast, Model 2 had an improved recall of 0.92 for class 1, suggesting it performed better in identifying fraudulent transactions.

> F1-Score:

- Model 1 had an F1-score of 0.83 for class 1, while Model 2 achieved a higher F1-score of 0.93 for class 1. This indicates that Model 2 had a better balance between precision and recall for identifying fraudulent transactions.

Conclusion :

Considering these points, it can be concluded that *Model 2 (the one after handling class imbalance) is better*.

Although Model 1 achieved a higher overall accuracy, Model 2 demonstrated improved performance in correctly identifying fraudulent transactions **(higher recall and F1-score for class 1).**

By addressing the class imbalance, Model 2 provides a more reliable and balanced prediction for both classes.

FINAL NOTES

Should we handle the Imbalance or NOT ?

> - **what if we don't handle class imbalance ?**

> - are we not training the model, as it happens in the real world ?

> - isn't the real world scenario biased ? like see the current data which is so biased ???