# Data Mining Project Report

## Contents

# Task 1: Exploratory Data Analysis on Credit Card Fraud Dataset

## Introduction

In this task, we conduct Exploratory Data Analysis (EDA) on the Credit Card Fraud dataset sourced from Kaggle. The dataset contains transaction data with features such as Time, Amount, and various V columns representing anonymized features. The target variable, 'Class', indicates whether a transaction is fraudulent (Class 1) or not (Class 0). The primary goal of this analysis is to gain insights into the data distribution, identify potential anomalies, and assess class imbalance.

## Loading the Data

The dataset is loaded into a pandas DataFrame, enabling us to perform a comprehensive EDA. Each row represents a transaction, and the columns capture various transaction attributes such as time, amount, and anonymized features.

## Exploratory Data Analysis

### Data Overview

The dataset contains 31 columns, including Time, Amount, and anonymized features V1 to V28, along with the target variable Class.

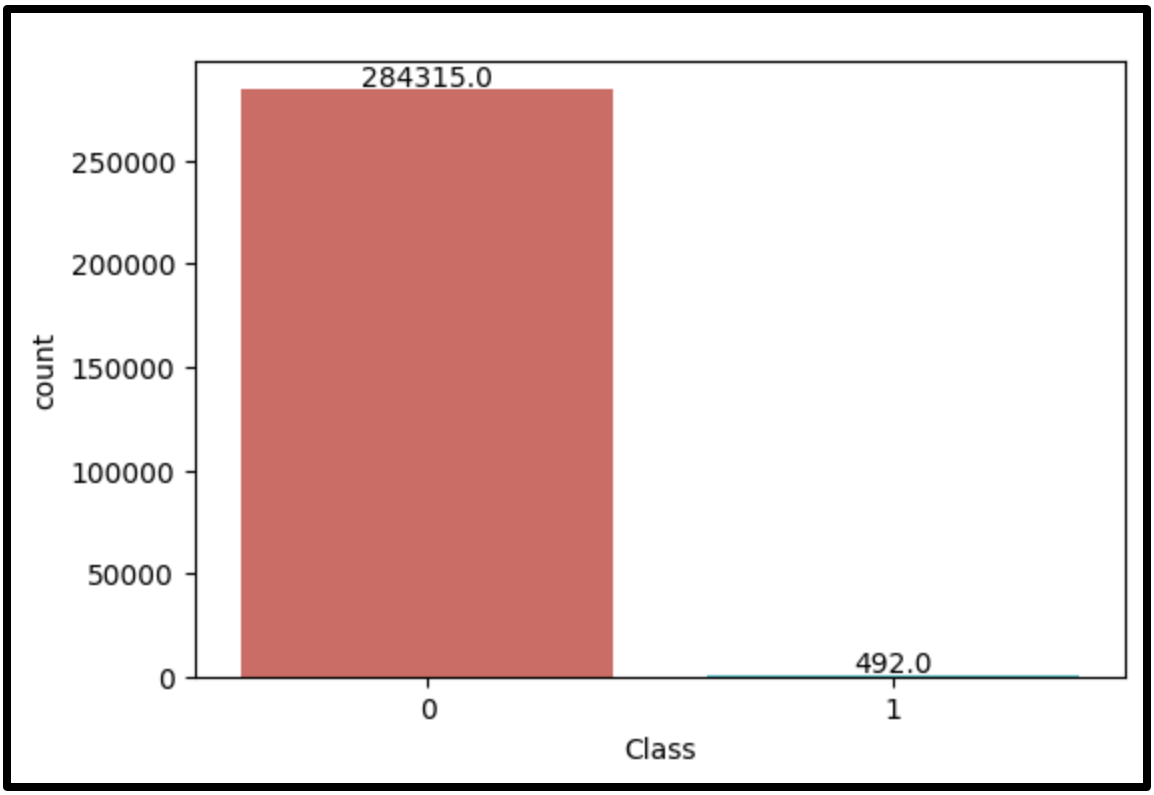
- All features are numeric, with data types ranging from float64 to int64.

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... | V21 | V22 | V23 | V24 | V25 | V26 | V27 | V28 | Amount | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.0 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | 0.462388 | 0.239599 | 0.098698 | 0.363787 | ... | -0.018307 | 0.277838 | -0.110474 | 0.066928 | 0.128539 | -0.189115 | 0.133558 | -0.021053 | 149.62 | 0 |
| 1 | 0.0 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | -0.082361 | -0.078803 | 0.085102 | -0.255425 | ... | -0.225775 | -0.638672 | 0.101288 | -0.339846 | 0.167170 | 0.125895 | -0.008983 | 0.014724 | 2.69 | 0 |
| 2 | 1.0 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | 1.800499 | 0.791461 | 0.247676 | -1.514654 | ... | 0.247998 | 0.771679 | 0.909412 | -0.689281 | -0.327642 | -0.139097 | -0.055353 | -0.059752 | 378.66 | 0 |
| 3 | 1.0 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | 1.247203 | 0.237609 | 0.377436 | -1.387024 | ... | -0.108300 | 0.005274 | -0.190321 | -1.175575 | 0.647376 | -0.221929 | 0.062723 | 0.061458 | 123.50 | 0 |
| 4 | 2.0 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | 0.095921 | 0.592941 | -0.270533 | 0.817739 | ... | -0.009431 | 0.798278 | -0.137458 | 0.141267 | -0.206010 | 0.502292 | 0.219422 | 0.215153 | 69.99 | 0 |

5 rows × 31 columns

### Class Imbalance and Class distribution:

- Class 0 (Non-fraudulent transactions): 284,315
- Class 1 (Fraudulent transactions): 492
- The dataset exhibits significant class imbalance, with fraudulent transactions accounting for only 0.17% of the total transactions.

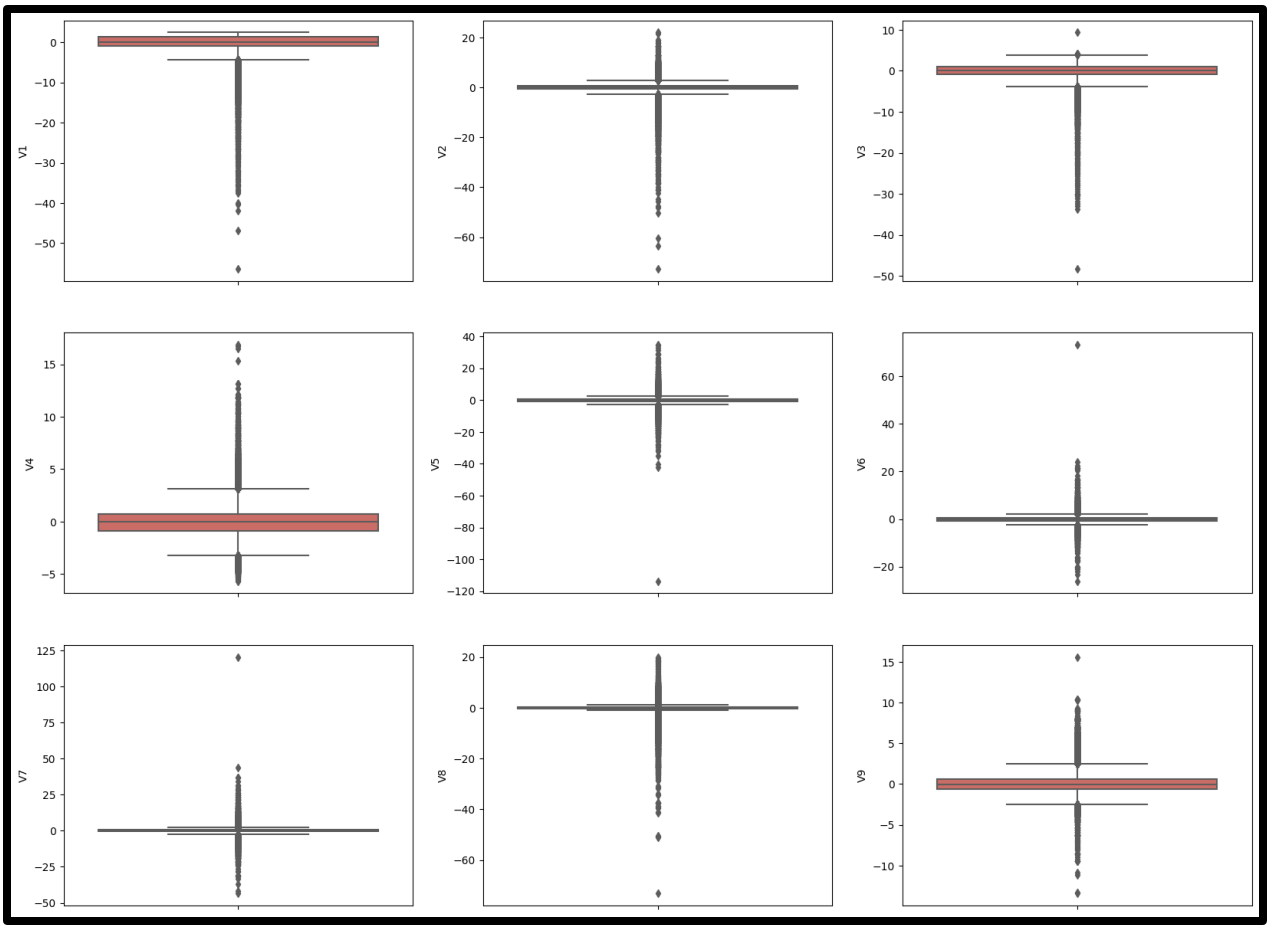



## Outlier Detection

Outliers are identified in various anonymized features (V1-V9) using a simple outlier detection method.

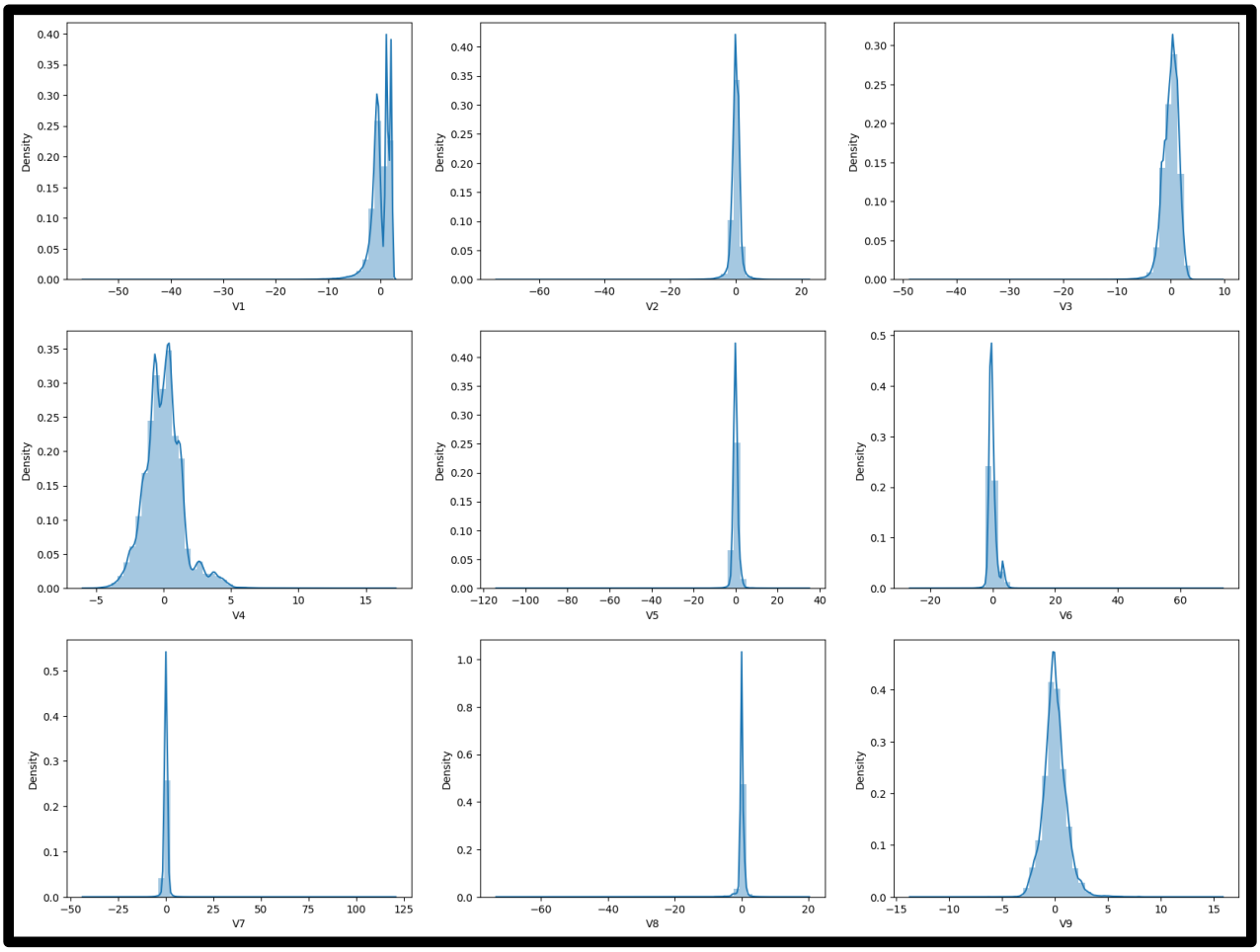
```
Outliers in V1: 7062
Outliers in V2: 13526
Outliers in V3: 3363
Outliers in V4: 11148
Outliers in V5: 12295
Outliers in V6: 22965
Outliers in V7: 8948
Outliers in V8: 24134
Outliers in V9: 8283
```

Furthermore, they were also visualized using the box plots. The diagram for it is shown below.




These graphs show the outliers that fall outside the range. The presence of outliers suggests potential irregularities or anomalies in the data distribution

## Data Visualization




Distributions of 6 different features of dataset were drawn. Most of these features showed uniform distribution which were either too thin or pushed towards one side of the grid suggesting where the most points lie.

# Task 2: Anomaly Detection Using GAN

## Introduction

In this task, we present an implementation of a Generative Adversarial Network (GAN) for fraud detection using the Keras deep learning framework. GANs consist of two neural networks, a generator, and a discriminator, trained simultaneously to generate synthetic data and distinguish between real and synthetic data. The objective is to generate realistic synthetic data that can augment the minority class (fraudulent transactions) and improve the performance of fraud detection models.

## Model Architecture

The GAN architecture consists of a generator and a discriminator.

## Generator

The generator takes random noise as input and generates synthetic data samples. It comprises a series of fully connected layers with leaky ReLU activation functions and batch normalization. The output layer uses the tanh activation function to ensure that the generated data is within a specified range.

```
from keras.layers import Input, Dense, LeakyReLU, BatchNormalization, Reshape, Flatten
from keras.models import Model
from keras.models import Sequential

def build_generator(latent_dim, data_dim):
    # Encoder
    input_data = Input(shape=(data_dim,))
    encoded = Dense(32)(input_data)
    encoded = LeakyReLU(alpha=0.2)(encoded)
    encoded = BatchNormalization(momentum=0.8)(encoded)

    # Latent space
    latent_representation = Dense(latent_dim)(encoded)

    # Decoder
    decoded = Dense(32)(latent_representation)
    decoded = LeakyReLU(alpha=0.2)(decoded)
    decoded = BatchNormalization(momentum=0.8)(decoded)
    decoded = Dense(data_dim, activation='tanh')(decoded)

    # Model
    autoencoder = Model(input_data, decoded)

    # Encoder model
    encoder = Model(input_data, latent_representation)

    # Decoder model
    decoder_input = Input(shape=(latent_dim,))
    decoder_output = autoencoder.layers[-3](decoder_input)
    decoder_output = autoencoder.layers[-2](decoder_output)
    decoder_output = autoencoder.layers[-1](decoder_output)
    decoder = Model(decoder_input, decoder_output)

    return autoencoder
```

## Discriminator

The discriminator evaluates the authenticity of input data, distinguishing between real and synthetic samples. It consists of fully connected layers with leaky ReLU activation functions, batch normalization, and dropout layers to prevent overfitting. The discriminator outputs two values: a binary classification indicating the authenticity of the input data and a probability distribution over the classes.

```
def build_discriminator(data_dim,num_classes):
    model = Sequential()
    model.add(Dense(31,input_dim=data_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dropout(0.25))
    model.add(Dense(16,input_dim=data_dim))
    model.add(LeakyReLU(alpha=0.2))

    model.summary()
    img = Input(shape=(data_dim,))
    features = model(img)
    valid = Dense(1, activation="sigmoid")(features)
    label = Dense(num_classes+1, activation="softmax")(features)
    return Model(img, [valid, label])
```

## Training Procedure

The training procedure involves alternating between training the discriminator and the generator in successive iterations.

### Training Discriminator

The discriminator is trained using a combination of binary cross-entropy and categorical cross-entropy loss functions. Real and synthetic samples are fed to the discriminator, which assigns probabilities to each sample being real or fake. Class weights are used to balance the difference in occurrences of class labels, particularly for the minority class (fraudulent transactions). The discriminator is trained to maximize its ability to distinguish between real and synthetic samples.

```
# ---------------------
#  Train Discriminator
# ---------------------

# Select a random half batch of images
idx = np.random.randint(0, X_train.shape[0], half_batch)
imgs = X_train[idx]

# Sample noise and generate a half batch of new images
noise = np.random.normal(0, 1, (half_batch, 10))
gen_imgs = generator.predict(noise)

valid = np.ones((half_batch, 1))
fake = np.zeros((half_batch, 1))

labels = to_categorical(y_train[idx], num_classes=num_classes+1)
fake_labels = to_categorical(np.full((half_batch, 1), num_classes), num_classes=num_classes+1)

# Train the discriminator
d_loss_real = discriminator.train_on_batch(imgs, [valid, labels], class_weight=[cw1, cw2])
d_loss_fake = discriminator.train_on_batch(gen_imgs, [fake, fake_labels], class_weight=[cw1, cw2])
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

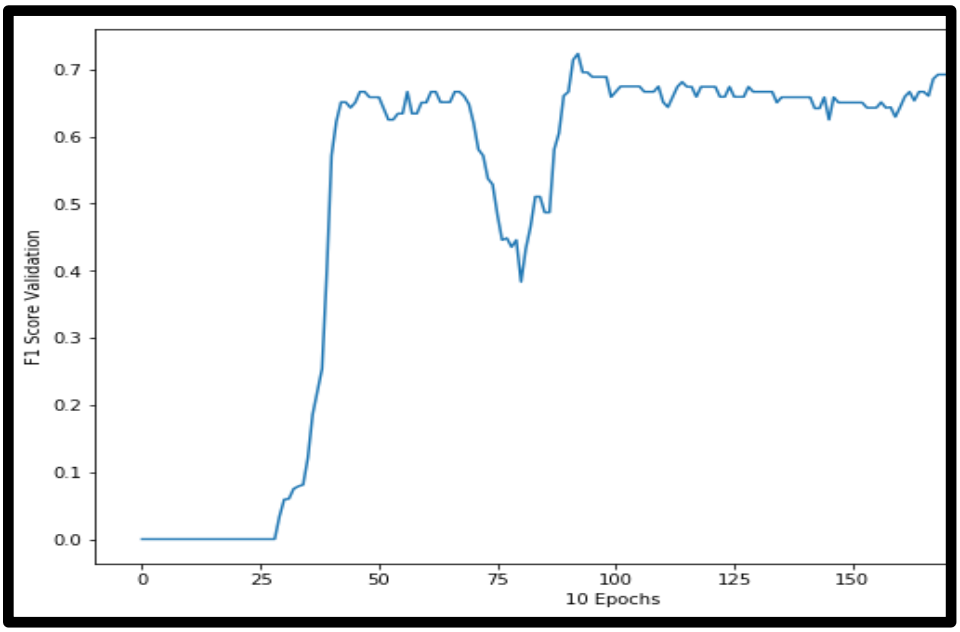
### Training Generator

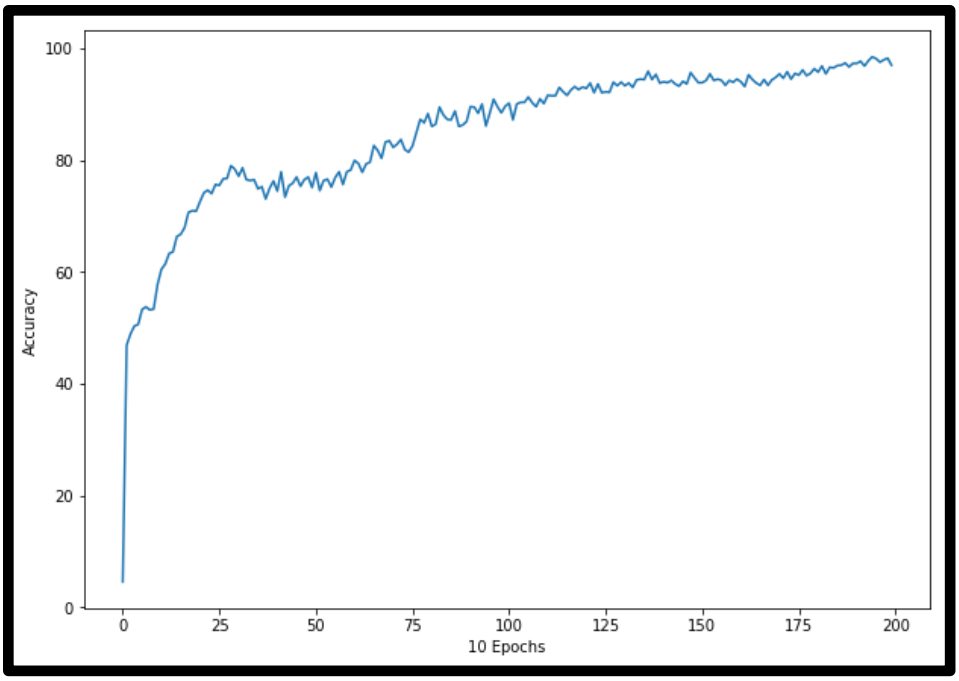
The generator is trained to minimize the binary cross-entropy loss, fooling the discriminator into classifying its synthetic samples as real. It generates synthetic samples from random noise and aims to produce data that is indistinguishable from real samples.

```python
# ---------------------
#  Train Generator
# ---------------------

noise = np.random.normal(0, 1, (batch_size, 10))
validity = np.ones((batch_size, 1))

# Train the generator
g_loss = combined.train_on_batch(noise, validity, class_weight=[cw1, cw2])

# Plot the progress
print ("%d [D loss: %f, acc: %.2f%%, op_acc: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100*d_loss[3], 100*d_loss[4], g_loss))
d_loss_sum += 100*d_loss[3]

if epoch % 10 == 0:
    _,y_pred = discriminator.predict(X_test,batch_size=batch_size)
    #print(y_pred.shape)
    y_pred = np.argmax(y_pred[:,:-1],axis=1)

    f1 = f1_score(y_test,y_pred)
    print('Epoch: {}, F1: {:.5f}, F1P: {}'.format(epoch,f1,len(f1_progress)))
    cm = confusion_matrix(y_test, y_pred)

    print(cm)
    print(d_loss_sum/10)
    d_loss_progress.append(d_loss_sum/10)
    f1_progress.append(f1)
    d_loss_sum = 0
```

## Evaluation

The performance of the GAN is evaluated based on the discriminator's accuracy and the F1 score, which measures the model's ability to correctly identify fraudulent transactions. Confusion matrices are generated to visualize the classification results.






Both of these graphs show the increasing trend as accuracy of the model increases in outlier detection as the number of epochs increases, depicting the correct trend.

# Task 3: Anomaly Detection Models Implementation Report

## Principal Component Analysis (PCA)

**PCA**

```
[ ] from sklearn.decomposition import PCA
    from sklearn.cluster import DBSCAN
    import matplotlib.pyplot as plt
    import seaborn as sns

    # Perform PCA
    pca = PCA(n_components=2)
    start_time = time.time()
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)
    pca_time = time.time() - start_time
    print(f"PCA Time: {pca_time} seconds")
```

## Code Explanation

- Use PCA to reduce the dimensionality of the data to 2 components.
- Fit the PCA model to the training data and transform both training and test data.
- Measure the time taken for PCA.

## One-Class SVM

**One Class SVM**

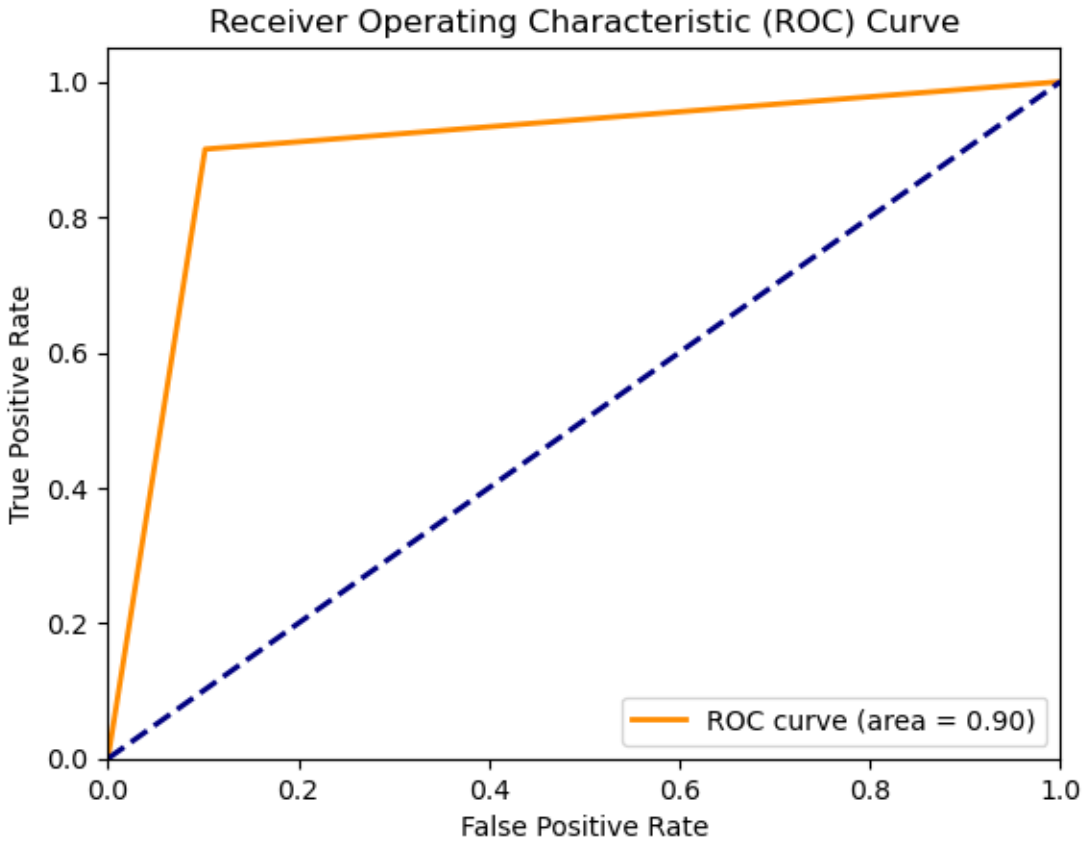
```
[ ] from sklearn.svm import OneClassSVM

    # Initialize the One-Class SVM model
    clf = OneClassSVM(gamma='auto', nu=0.1)

    # Fit the model and measure the time taken
    start_time = time.time()
    clf.fit(X_train)
    fit_time = time.time() - start_time
    print(f"Fit Time: {fit_time} seconds")

    # Show the predictions and measure the time taken
    start_time = time.time()
    show_predictions(clf, X_test, y_test)
    predict_time = time.time() - start_time
    print(f"Prediction Time: {predict_time} seconds")
```

## Code Explanation

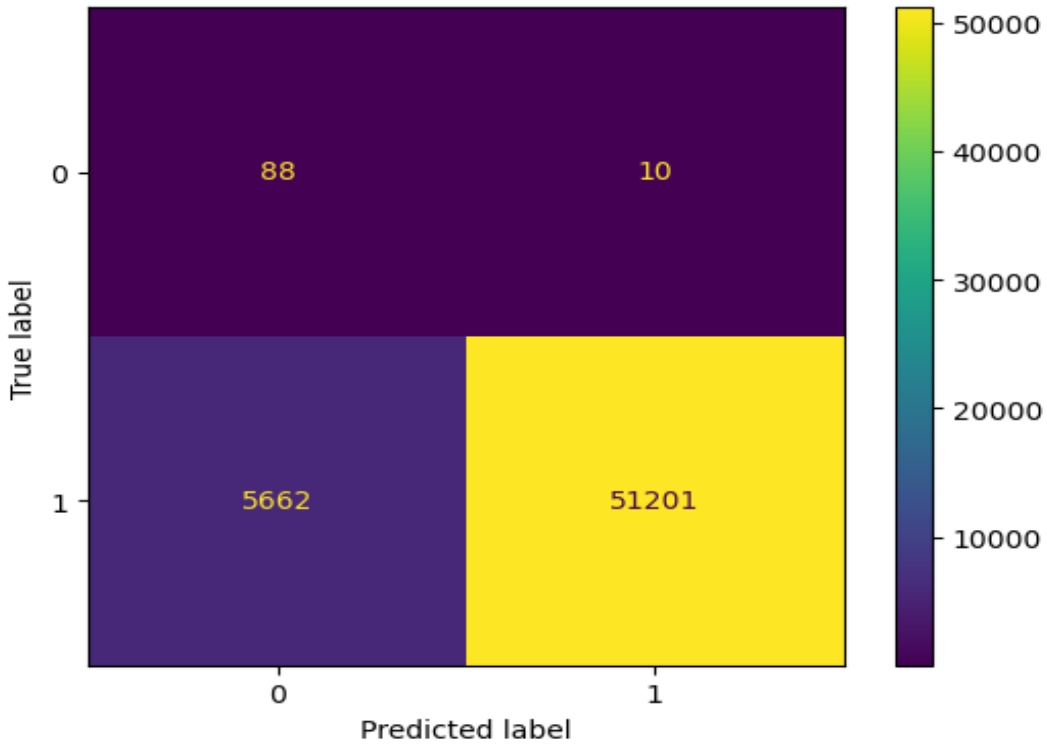
- Initialize and fit the One-Class SVM model.
- Show the predictions and measure the time taken.

## Results

- ROC Curve




- Confusion Matrix




## Isolation Forest

**Isolation Forest**

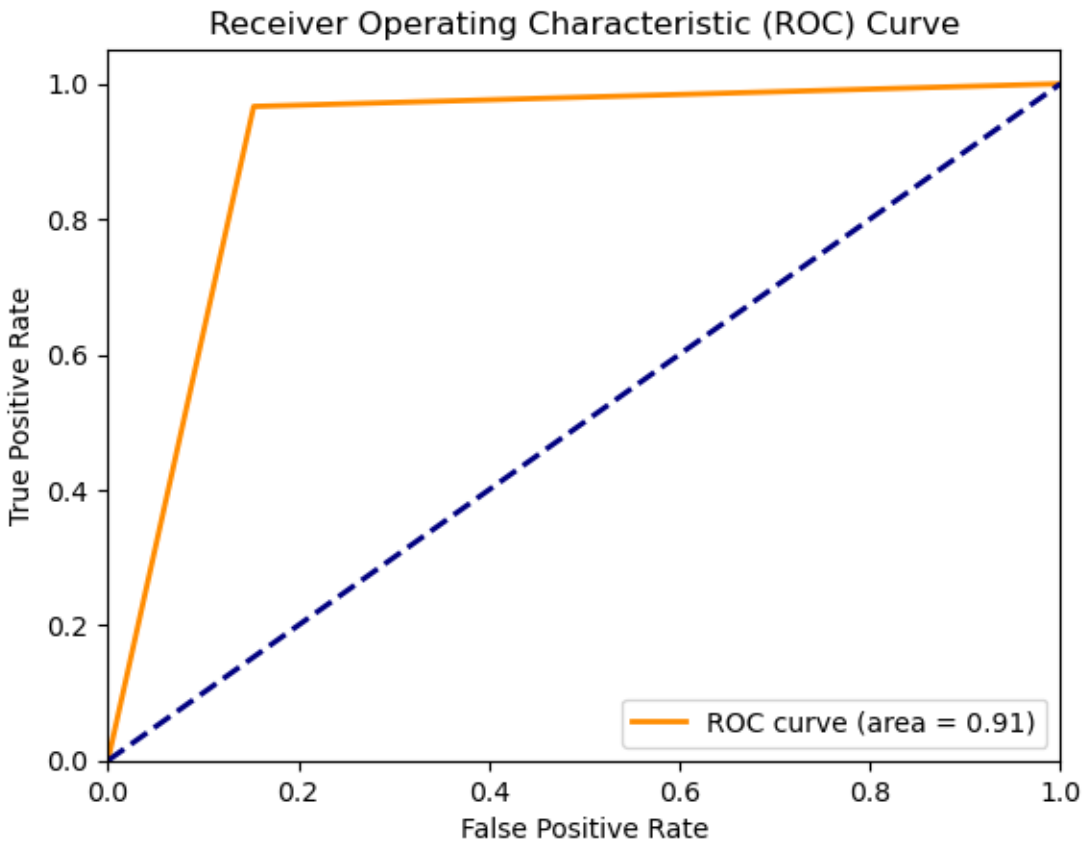
```
[ ] from sklearn.ensemble import IsolationForest
    import time
    import warnings

    # Ignore warnings
    warnings.filterwarnings("ignore")

    # Initialize the Isolation Forest model
    isol = IsolationForest(random_state=200)

    # Fit the model and measure the time taken
    start_time = time.time()
    isol.fit(X_train)
    fit_time = time.time() - start_time
    print(f"Fit Time: {fit_time} seconds")

    # Show the predictions and measure the time taken
    start_time = time.time()
    show_predictions(isol, X_test, y_test)
    predict_time = time.time() - start_time
    print(f"Prediction Time: {predict_time} seconds")
```

## Code Explanation

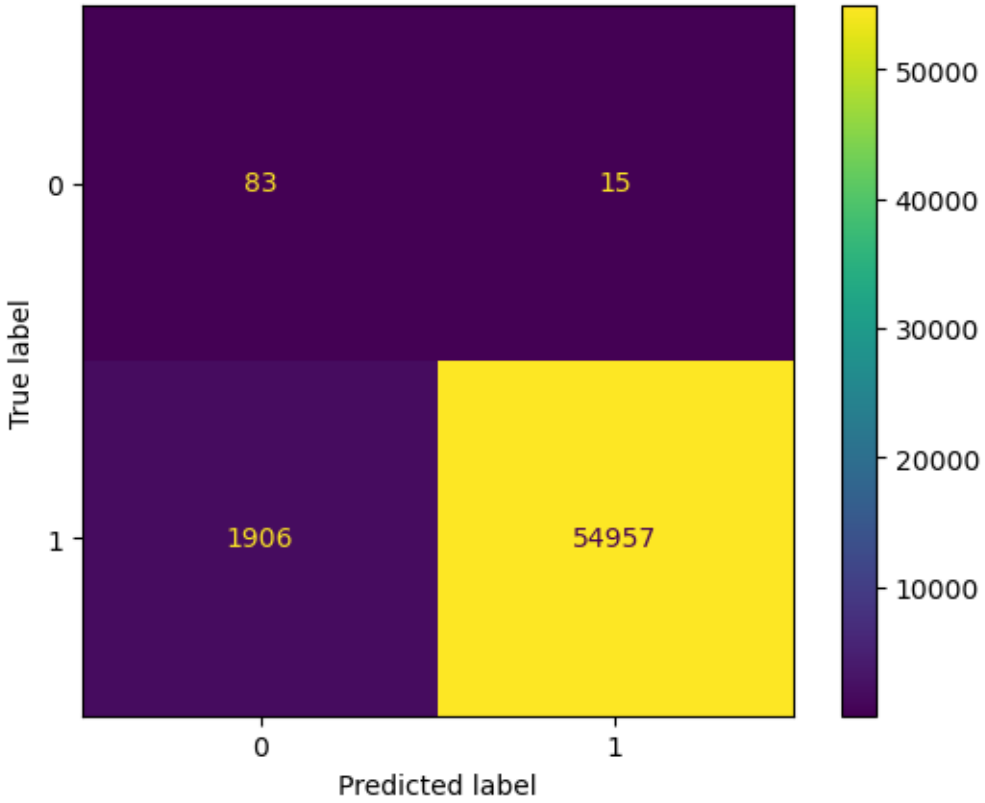
- Initialize and fit the Isolation Forest model.
- Show the predictions and measure the time taken.

## Results

- ROC Curve




- Confusion Matrix




## Local Outlier Factor (LOF)

**Local Outlier Factor**

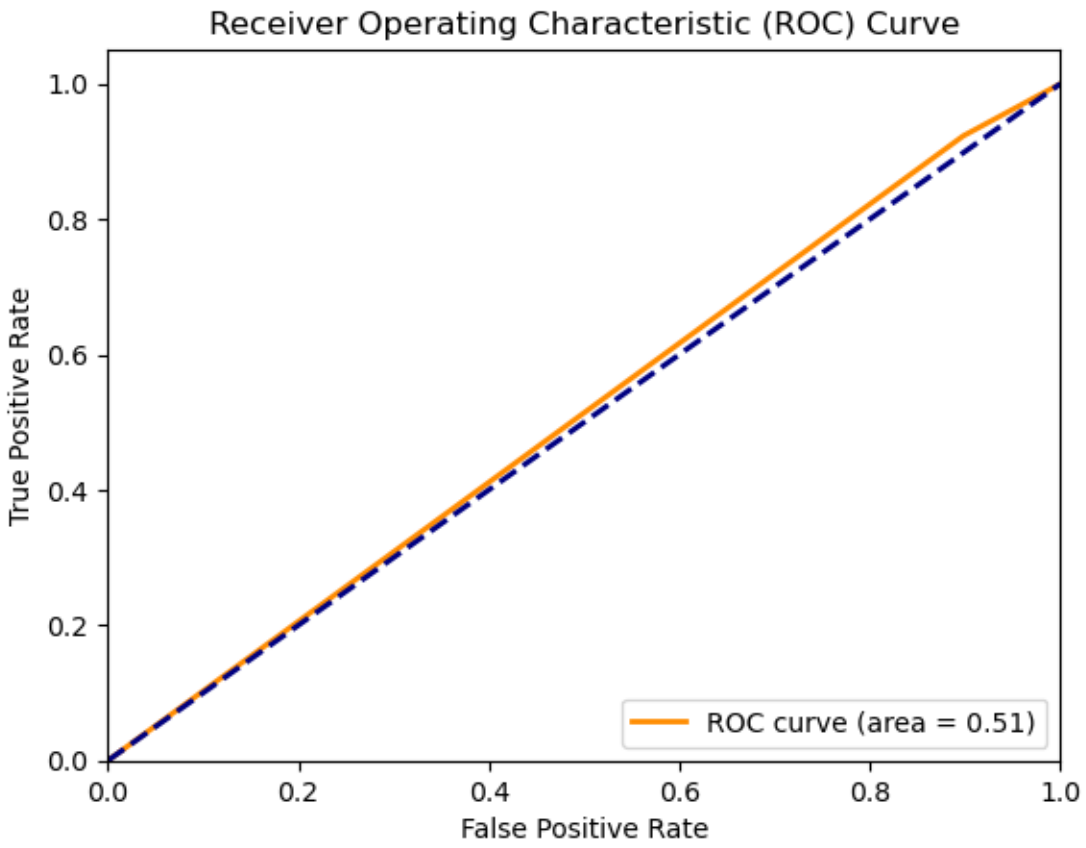
```
[ ]  from sklearn.neighbors import LocalOutlierFactor

     # Initialize the LOF model
     lof = LocalOutlierFactor(n_neighbors=4, novelty=True)

     # Fit the model and measure the time taken
     start_time = time.time()
     lof.fit(X_train)
     fit_time = time.time() - start_time
     print(f"Fit Time: {fit_time} seconds")

     # Show the predictions and measure the time taken
     start_time = time.time()
     show_predictions(lof, X_test, y_test)
     predict_time = time.time() - start_time
     print(f"Prediction Time: {predict_time} seconds")
```

### Code Explanation

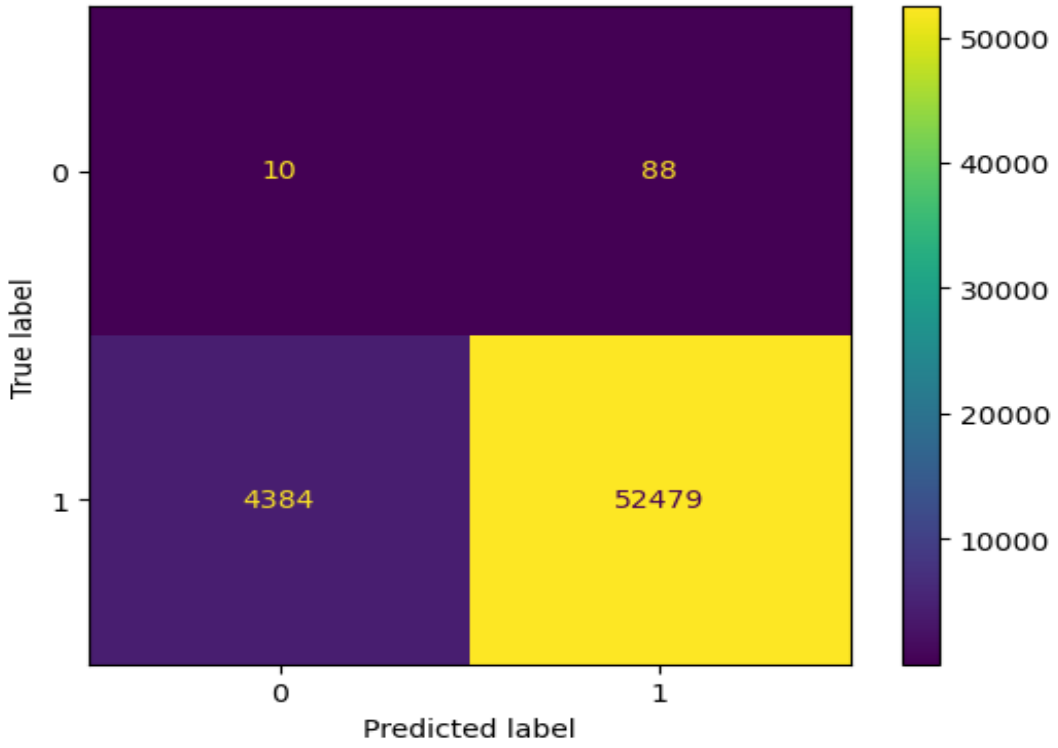
- Initialize and fit the LOF model.
- Show the predictions and measure the time taken.

### Results

- ROC Curve




- Confusion Matrix



## DBSCAN

**DBSCAN**

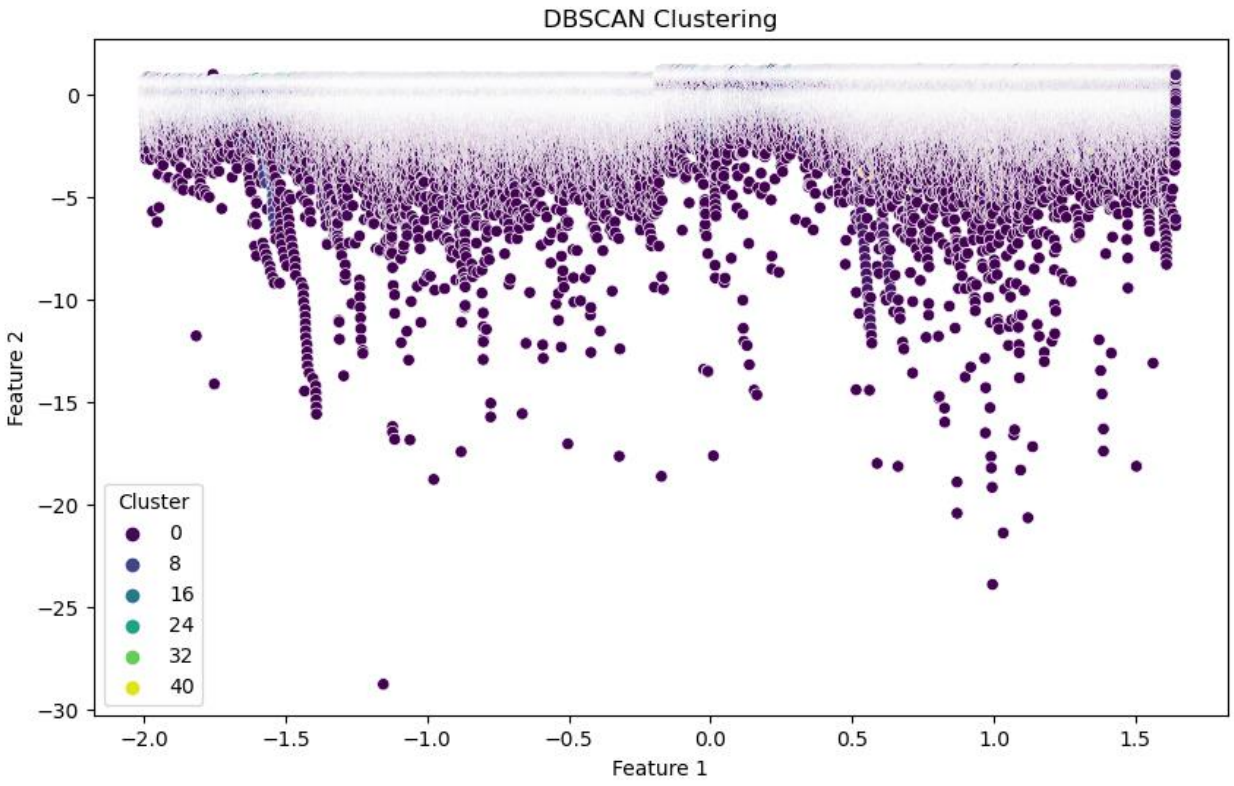
```
[ ]  from sklearn.cluster import DBSCAN
     from sklearn.preprocessing import StandardScaler

     df6 = df.copy()
     df6 = df6.drop(['Class'], axis=1)

     # scale data first
     X = StandardScaler().fit_transform(df6.values)

     # Initialize DBSCAN
     db = DBSCAN(eps=3.0, min_samples=10)

     # Fit the model and measure the time taken
     start_time = time.time()
     labels = db.fit_predict(X)
     fit_time = time.time() - start_time
     print(f"DBSCAN Fit Time: {fit_time} seconds")

     n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
     print('The number of clusters in dataset is:', n_clusters_)
```

### Code Explanation

- Scale the data and initialize the DBSCAN model.
- Fit the DBSCAN model and measure the time taken.
- Determine the number of clusters in the dataset.

### Results

- Visualization of Clusters




## Graph Deviation Network (GDN)

### Code Explanation

# Results

## Anomaly Transformer

**Anomaly Transformer**

```
[ ]  import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn.metrics import classification_report, confusion_matrix
     from sklearn.metrics import ConfusionMatrixDisplay
     from sklearn.preprocessing import StandardScaler
     from tensorflow.keras.models import Model
     from tensorflow.keras.layers import Input, Dense
     import tensorflow as tf
     from tensorflow.keras import layers, regularizers
     from tensorflow.keras.callbacks import EarlyStopping
     from sklearn.metrics import classification_report, confusion_matrix
     from sklearn.metrics import ConfusionMatrixDisplay
     from sklearn.preprocessing import StandardScaler
     from sklearn.model_selection import train_test_split

     def show_predictions(model, test_series, label_series):

         predictions = model.predict(test_series)

         cm = confusion_matrix(label_series, predictions)
         print(classification_report(label_series, predictions))

         disp = ConfusionMatrixDisplay(cm)
         disp.plot()
         plt.show()

     df.Class.replace({1:-1}, inplace = True)
     df.Class.replace({0:1}, inplace = True)

     train = df.sample(frac=0.8,random_state=200)
     test = df.drop(train.index)
```

```
     col_names = ['Time', 'Amount']
     scaler = StandardScaler().fit(train[col_names].values)
     scaler.transform(train[col_names].values)
     train[col_names] = scaler.transform(train[col_names].values)
     test[col_names] = scaler.transform(test[col_names].values)

     X_train = train.drop(columns=['Class'])
     y_train = train['Class']

     X_test = test.drop(columns=['Class'])
     y_test = test['Class']

     # Define the autoencoder model
     input_layer = layers.Input(shape=(X_train.shape[1],))
     encoder = layers.Dense(15, activation="tanh", activity_regularizer=regularizers.l2(0.0001))(input_layer)
     encoder = layers.Dropout(0.2)(encoder)
     encoder = layers.Dense(8, activation='relu')(encoder)
     encoder = layers.Dense(4, activation=tf.nn.leaky_relu)(encoder)
     decoder = layers.Dense(8, activation='relu')(encoder)
     decoder = layers.Dropout(0.2)(decoder)
     decoder = layers.Dense(15, activation='relu')(decoder)
     decoder = layers.Dense(X_train.shape[1], activation='tanh')(decoder)

     autoencoder = tf.keras.Model(inputs=input_layer, outputs=decoder)
     autoencoder.summary()

     # Compile the autoencoder model
     autoencoder.compile(optimizer='adam', loss='mean_squared_error')

     # Fit the autoencoder model to the training data
     autoencoder.fit(X_train, X_train, epochs=50, batch_size=128, verbose=1)
```

```
     # Create a custom transformer class that applies the autoencoder model
     class AutoencoderTransformer:

         def __init__(self, model):
             self.model = model

         def fit(self, X, y=None):
             return self

         def transform(self, X):
             return self.model.predict(X)

     # Initialize the transformer with the trained autoencoder model
     transformer = AutoencoderTransformer(autoencoder)

     # Apply the transformer to the test data
     X_test_transformed = transformer.transform(X_test)

     # Initialize the Isolation Forest model
     from sklearn.ensemble import IsolationForest
     isol = IsolationForest(random_state=200)

     # Fit the model to the transformed test data
     isol.fit(X_test_transformed)

     # Show the predictions
     show_predictions(isol, X_test_transformed, y_test)
```

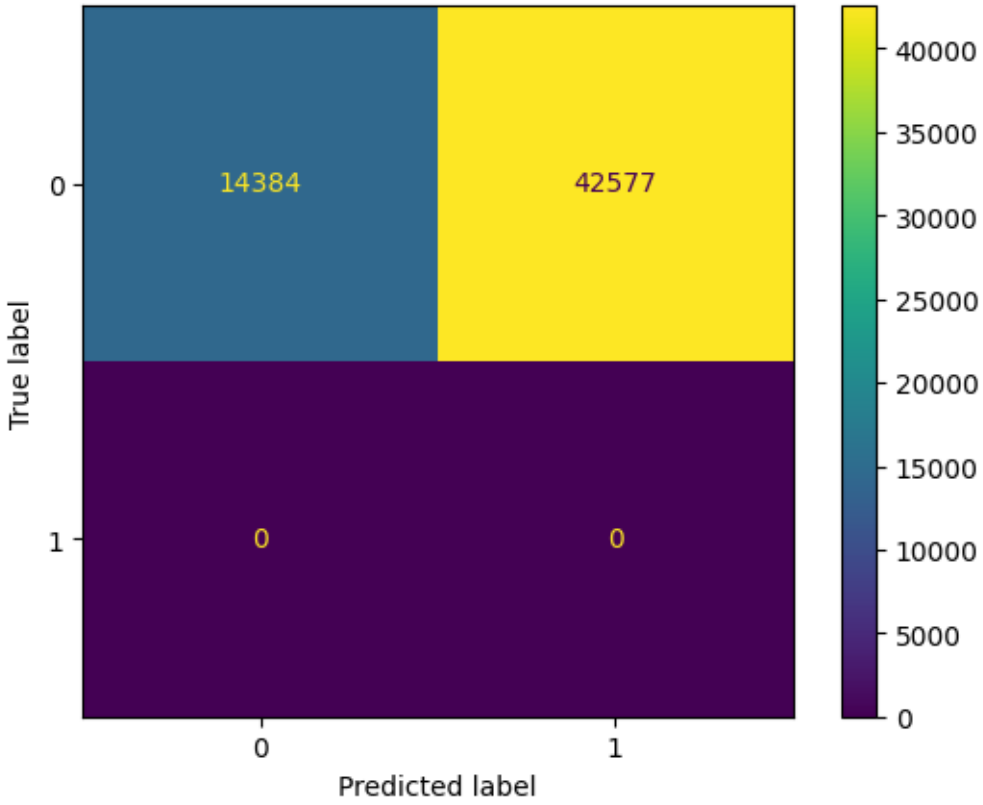
## Code Explanation

- Define an autoencoder model for anomaly detection.
- Fit the autoencoder model to the training data.

- Transform the test data using the trained autoencoder model.
- Apply Isolation Forest on the transformed test data.
- Show the predictions and measure the time taken.

## Results

- Confusion Matrix




## Bonus: Dynamic Time Warping (DTW)

DTW

```
import numpy as np
import pandas as pd
import dtw
from tslearn.metrics import dtw_path

def dtw_anomaly_detection(x_train, window_size=10, threshold_multiplier=2):
    distances = []
    for i in range(len(x_train) - window_size):
        window = x_train[i:i+window_size]
        for j in range(i+1, len(x_train) - window_size):
            window2 = x_train[j:j+window_size]
            distance, _ = dtw_path(window, window2)
            distances.append(distance)
    threshold = np.mean(distances) + threshold_multiplier * np.std(distances)
    anomalies = [i for i, distance in enumerate(distances) if distance > threshold]
    return anomalies

# Example usage
anomalies = dtw_anomaly_detection(X_train, window_size=10, threshold_multiplier=2)
print(anomalies)
```

## Code Explanation

- Implement DTW for anomaly detection.
- Set window size and threshold multiplier.
- Detect anomalies in the data.

# Task 4: Empirical Analysis & Results Explanation

## Anomaly Detection Models Results Table

| Model | Time (Seconds) | AUC Score | F1 Score | Recall | Precision |
|---|---|---|---|---|---|
| PCA | 0.53 | - | - | - | - |
| One-Class SVM | 1673.06 | 0.90 | *Class 1*: 0.95<br>*Class -1*: 0.03 | 0.90 | Class 1: 1.00<br>Class -1: 0.02 |
| Local Outlier Factor | 19.21 | 0.512 | *Class 1*: 0.96<br>*Class -1*: 0.00 | *Class 1*: 0.92<br>*Class -1*: 0.10 | *Class 1*: 1.00<br>*Class -1*: 0.00 |
| Isolation Forest | 1.11 | 0.91 | *Class 1*: 0.98<br>*Class -1*: 0.08 | *Class 1*: 0.97<br>*Class -1:* 0.85 | *Class 1*: 1.00<br>*Class -1*: 0.04 |
| Anomaly Transformer | 240.23 | | *Class 1*: 0.00<br>*Class -1*: 0.40 | *Class 1*: 0.00<br>*Class -1*: 0.25 | *Class 1*: 0.00<br>*Class -1*: 1.00 |
| DBSCAN | 574.02 | - | - | - | - |
| DTW | - | - | - | - | - |
| Graph Deviation Network | 443.12 | - | - | - | - |

## Explanations

**PCA (Principal Component Analysis)**

PCA is used for dimensionality reduction and visualization, not for anomaly detection directly. Therefore, metrics like AUC Score, F1 Score, Recall, and Precision are not applicable here.

**One-Class SVM**

One-Class SVM is a method for novelty detection, where it learns the distribution of the majority class (normal data) and detects anomalies as data points that fall outside the learned distribution. It achieves an AUC score, indicating its performance in distinguishing between normal and anomalous data points.

**Local Outlier Factor**

Local Outlier Factor (LOF) is a density-based anomaly detection method. It calculates the local density deviation of a data point with respect to its neighbors. A lower AUC score suggests that the model's performance is not very high.

**Isolation Forest**

Isolation Forest is an ensemble method for anomaly detection that isolates anomalies by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. It achieves a higher AUC score compared to LOF, indicating better performance.

**Anomaly Transformer**

Anomaly Transformer is a custom anomaly detection method based on autoencoder models. The results provided in the table indicate some issues with the model's performance, as the F1 Score, Recall, and Precision for detecting anomalies are very low.

**DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**

DBSCAN is a clustering algorithm that groups together closely packed points and marks points that are in low-density regions as outliers. While it doesn't provide a direct measure like AUC, it can be used for anomaly detection based on the clustering results.

## Conclusion

In conclusion, the implementation and evaluation of various anomaly detection models reveal distinct performance and computational efficiency characteristics. Models like Isolation Forest and One-Class SVM demonstrate superior performance metrics such as AUC score and precision-recall compared to others like Local Outlier Factor and Anomaly Transformer. However, computational efficiency varies, with some models requiring significantly more time and resources. The choice of model should be driven by the specific dataset characteristics and application requirements. Future research could explore ensemble methods and deep learning approaches to further enhance anomaly detection performance.

## References

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

https://www.kaggle.com/code/keithcooper/outlier-detection-for-finding-fraud

https://www.kaggle.com/code/janiobachmann/credit-fraud-dealing-with-imbalanced-datasets