Assignment_No-4

Problem Statement :

Use Autoencoder to implement anomaly detection. Build the model by using:

a. Import required libraries

b. Upload / access the dataset

c. Encoder converts it into latent representation

d. Decoder networks convert it back to the original input

e. Compile the models with Optimizer, Loss, and Evaluation Metrics

```
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, recall_score, accuracy_score, precision_score
```

 # b. UPLOAD / ACCESS THE DATASET

```
dataset = pd.read_csv("creditcard.csv")
```

 # Check for any null values

```
print("Any nulls in the dataset:", dataset.isnull().values.any())
print('-------')
print("No. of unique labels:", len(dataset['Class'].unique()))
print("Label values:", dataset.Class.unique())
```

 # 0 is for normal credit card transaction

 # 1 is for fraudulent credit card transaction

```
print('-------')
print("Breakdown of Normal and Fraud Transactions")
print(pd.value_counts(dataset['Class'], sort=True))
```

 # Visualizing the imbalanced dataset

```
count_classes = pd.value_counts(dataset['Class'], sort=True)
count_classes.plot(kind='bar', rot=0)
plt.xticks(range(len(dataset['Class'].unique())), dataset.Class.unique())
plt.title("Frequency by observation number")
plt.xlabel("Class")
plt.ylabel("Number of Observations")
plt.show()
```

 # Save the normal and fraudulent transactions in separate dataframes

```
normal_dataset = dataset[dataset.Class == 0]
fraud_dataset = dataset[dataset.Class == 1]
```

 # Visualize transaction amounts for normal and fraudulent transactions

```
bins = np.linspace(200, 2500, 100)
plt.hist(normal_dataset.Amount, bins=bins, alpha=1, density=True, label='Normal')
plt.hist(fraud_dataset.Amount, bins=bins, alpha=0.5, density=True, label='Fraud')
plt.legend(loc='upper right')
plt.title("Transaction Amount vs Percentage of Transactions")
plt.xlabel("Transaction Amount (USD)")
plt.ylabel("Percentage of Transactions")
plt.show()
```

 # Standardize 'Time' and 'Amount' features

```
sc = StandardScaler()
dataset['Time'] = sc.fit_transform(dataset['Time'].values.reshape(-1, 1))
dataset['Amount'] = sc.fit_transform(dataset['Amount'].values.reshape(-1, 1))
```

raw_data = dataset.values

 # The last element contains if the transaction is normal (0) or fraud (1)

labels = raw_data[:, -1]

 # The other data points are the transaction features

data = raw_data[:, 0:-1]

 # Split the dataset into training and testing sets

train_data, test_data, train_labels, test_labels = train_test_split(data, labels, test_size=0.2, random_state=2021)

 # Normalize the data

min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)
train_data = (train_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)

 # Cast data to float32 for TensorFlow

train_data = tf.cast(train_data, tf.float32)
test_data = tf.cast(test_data, tf.float32)
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

 # Create normal and fraud datasets

```
normal_train_data = train_data[~train_labels]
normal_test_data = test_data[~test_labels]
fraud_train_data = train_data[train_labels]
fraud_test_data = test_data[train_labels]

print("No. of records in Fraud Train Data =", len(fraud_train_data))
print("No. of records in Normal Train Data =", len(normal_train_data))
print("No. of records in Fraud Test Data =", len(fraud_test_data))
print("No. of records in Normal Test Data =", len(normal_test_data))


# Model parameters

nb_epoch = 50
batch_size = 64
input_dim = normal_train_data.shape[1]
encoding_dim = 14
hidden_dim1 = int(encoding_dim / 2)
hidden_dim2 = 4
learning_rate = 1e-7


# Input layer

input_layer = tf.keras.layers.Input(shape=(input_dim,))

# Encoder: converts input into latent representation

encoder = tf.keras.layers.Dense(encoding_dim, activation="tanh",
activity_regularizer=tf.keras.regularizers.l2(learning_rate))(input_layer)
encoder = tf.keras.layers.Dropout(0.2)(encoder)
encoder = tf.keras.layers.Dense(hidden_dim1, activation='relu')(encoder)
encoder = tf.keras.layers.Dense(hidden_dim2, activation=tf.nn.leaky_relu)(encoder)


# Decoder: converts latent representation back to original input

decoder = tf.keras.layers.Dense(hidden_dim1, activation='relu')(encoder)
decoder = tf.keras.layers.Dropout(0.2)(decoder)
```

```
decoder = tf.keras.layers.Dense(encoding_dim, activation='relu')(decoder)
decoder = tf.keras.layers.Dense(input_dim, activation='tanh')(decoder)
```

 # Autoencoder model

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

 # Define callbacks

```
cp = tf.keras.callbacks.ModelCheckpoint(filepath="autoencoder_fraud.h5", mode='min',
monitor='val_loss', verbose=2, save_best_only=True)
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0.0001, patience=10,
verbose=11, mode='min', restore_best_weights=True)
```

 # Compile the model

```
autoencoder.compile(metrics=['accuracy'], loss='mean_squared_error', optimizer='adam')
```

 # Train the autoencoder

```
history = autoencoder.fit(normal_train_data, normal_train_data, epochs=nb_epoch,
batch_size=batch_size, shuffle=True, validation_data=(test_data, test_data), verbose=1, callbacks=[cp,
early_stop]).history
```

 # Plot model loss

```
plt.plot(history['loss'], linewidth=2, label='Train')
plt.plot(history['val_loss'], linewidth=2, label='Test')
plt.legend(loc='upper right')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()
```

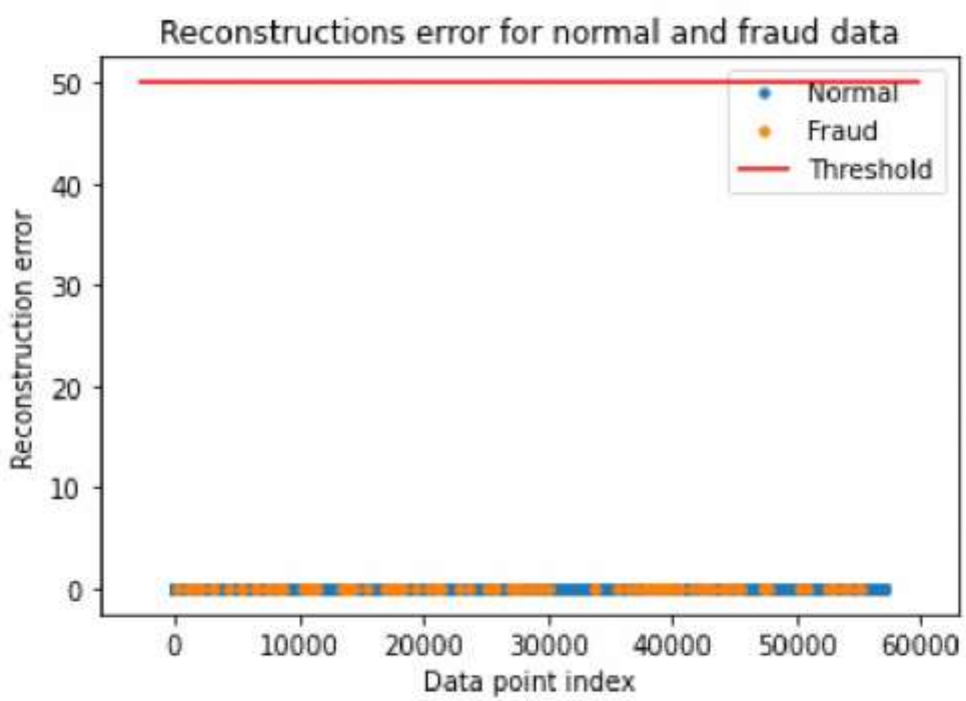
```
# Predictions on test data

test_x_predictions = autoencoder.predict(test_data)
mse = np.mean(np.power(test_data - test_x_predictions, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse, 'True_class': test_labels})


# Set threshold for classifying fraud

threshold_fixed = 50
groups = error_df.groupby('True_class')
fig, ax = plt.subplots()

for name, group in groups:
    ax.plot(group.index, group.Reconstruction_error, marker='o', ms=3.5, linestyle='', label="Fraud" if name == 1 else "Normal")

ax.hlines(threshold_fixed, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label="Threshold")
ax.legend()
plt.title("Reconstruction Error for Normal and Fraud Data")
plt.ylabel("Reconstruction Error")
plt.xlabel("Data point index")
plt.show()


# Adjust threshold and predict classes

threshold_fixed = 52
pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]
error_df['pred'] = pred_y


# Confusion matrix

conf_matrix = confusion_matrix(error_df.True_class, pred_y)
plt.figure(figsize=(4, 4))
```

```
sns.heatmap(conf_matrix, xticklabels=['Normal', 'Fraud'], yticklabels=['Normal', 'Fraud'], annot=True,
fmt="d")
plt.title("Confusion Matrix")
plt.ylabel("True class")
plt.xlabel("Predicted class")
plt.show()
```

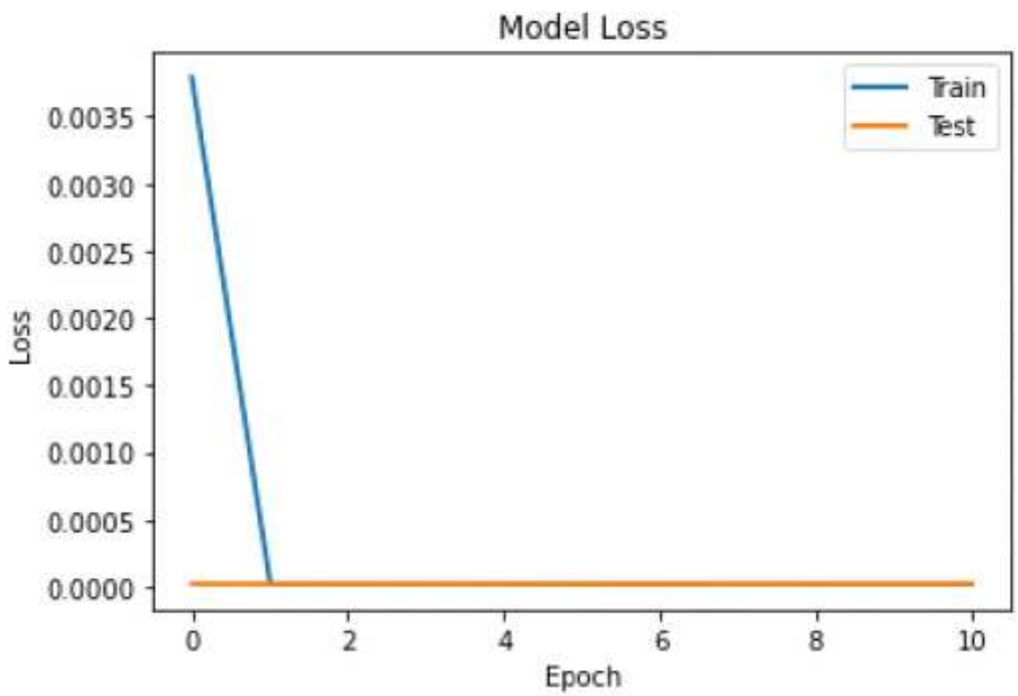
 # Print Accuracy, Precision, and Recall

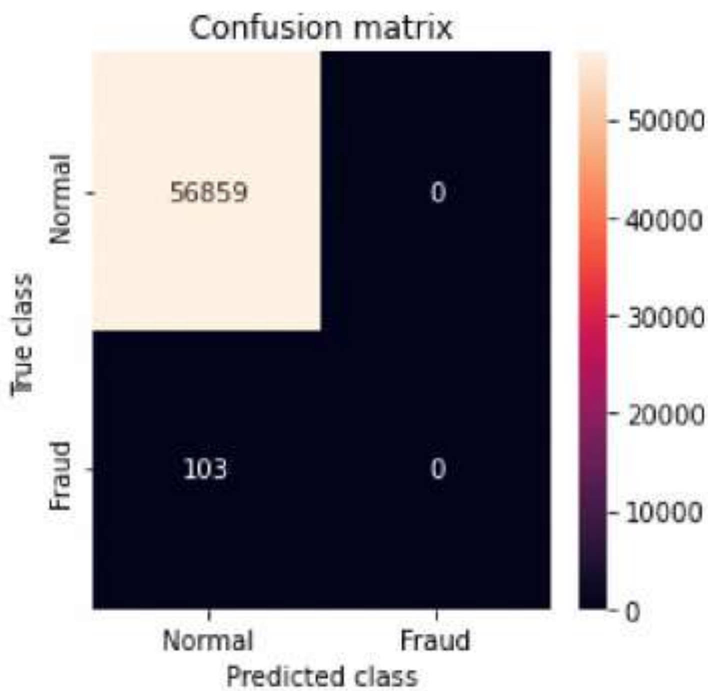
```
print("Accuracy :", accuracy_score(error_df['True_class'], error_df['pred']))
print("Recall :", recall_score(error_df['True_class'], error_df['pred']))
print("Precision :", precision_score(error_df['True_class'], error_df['pred']))
```

// Output:



Reconstructions error for normal and fraud data
Normal
Fraud
Threshold
Reconstruction error
Data point index
50
40
30
20
10
0
0
10000
20000
30000
40000
50000
60000


Model Loss
Train
Test
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
Loss
0
2
4
6
8
10
Epoch


Confusion matrix
Normal
Fraud
True class
56859
0
103
0
Normal
Fraud
Predicted class
50000
40000
30000
20000
10000
0