In [1]:
```
import pandas as pd
import numpy as np

import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

In [2]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, recall_score, accuracy_score, precisi
RANDOM_SEED = 2021
TEST_PCT = 0.3
LABELS = ["Normal","Fraud"]
```

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

```
['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']
```

Out[3]:

	Time	V1	V2	V3	V4	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8480
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.6040
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.3802
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.1374
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.9159
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.4335
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.1192
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.4801

8 rows × 31 columns

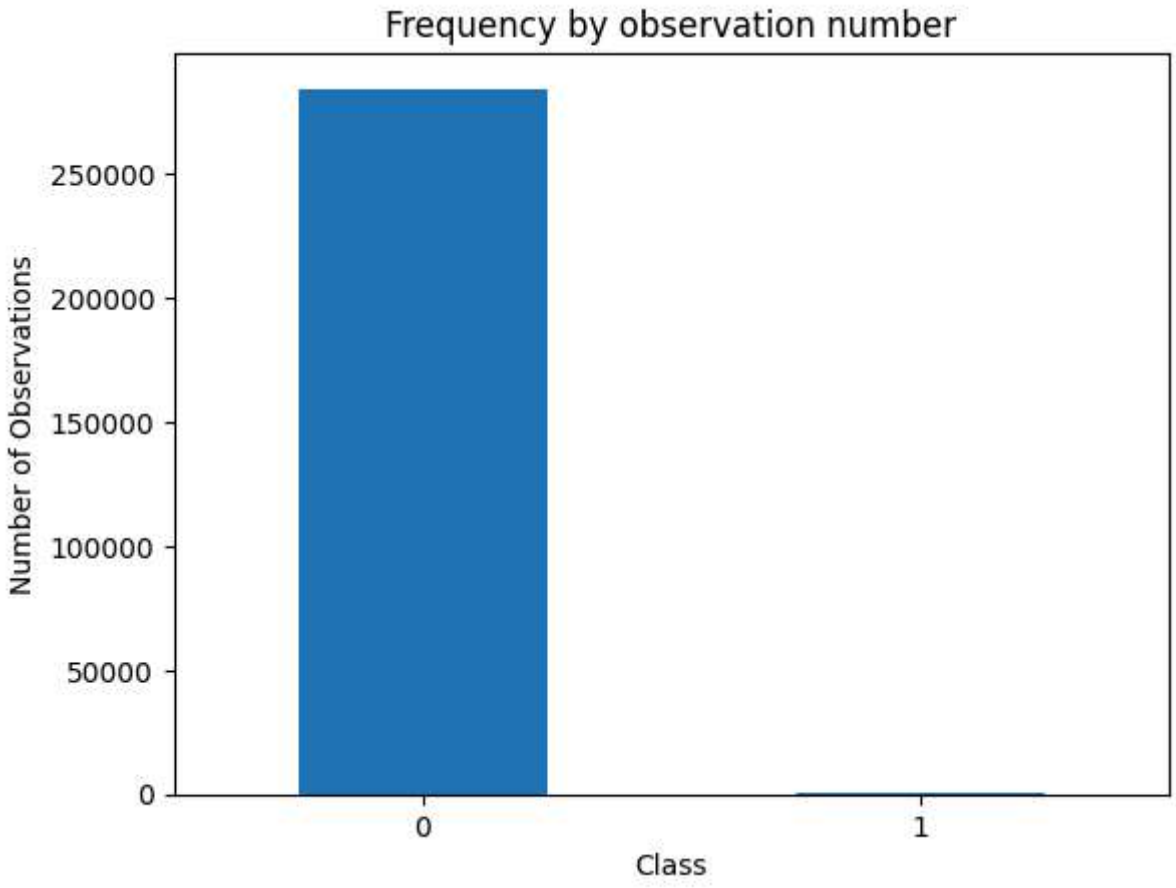
In [4]:
```
#check for any  nullvalues
print("Any nulls in the dataset ",dataset.isnull().values.any() )
print('-------')
print("No. of unique labels ", len(dataset['Class'].unique()))
print("Label values ",dataset.Class.unique())
#0 is for normal credit card transaction
#1 is for fraudulent credit card transaction
print('-------')
print("Break down of the Normal and Fraud Transactions")
print(pd.value_counts(dataset['Class'], sort = True) )
```

```
Any nulls in the dataset  False
-------
No. of unique labels  2
Label values  [0 1]
-------
Break down of the Normal and Fraud Transactions
Class
0    284315
1       492
Name: count, dtype: int64
```

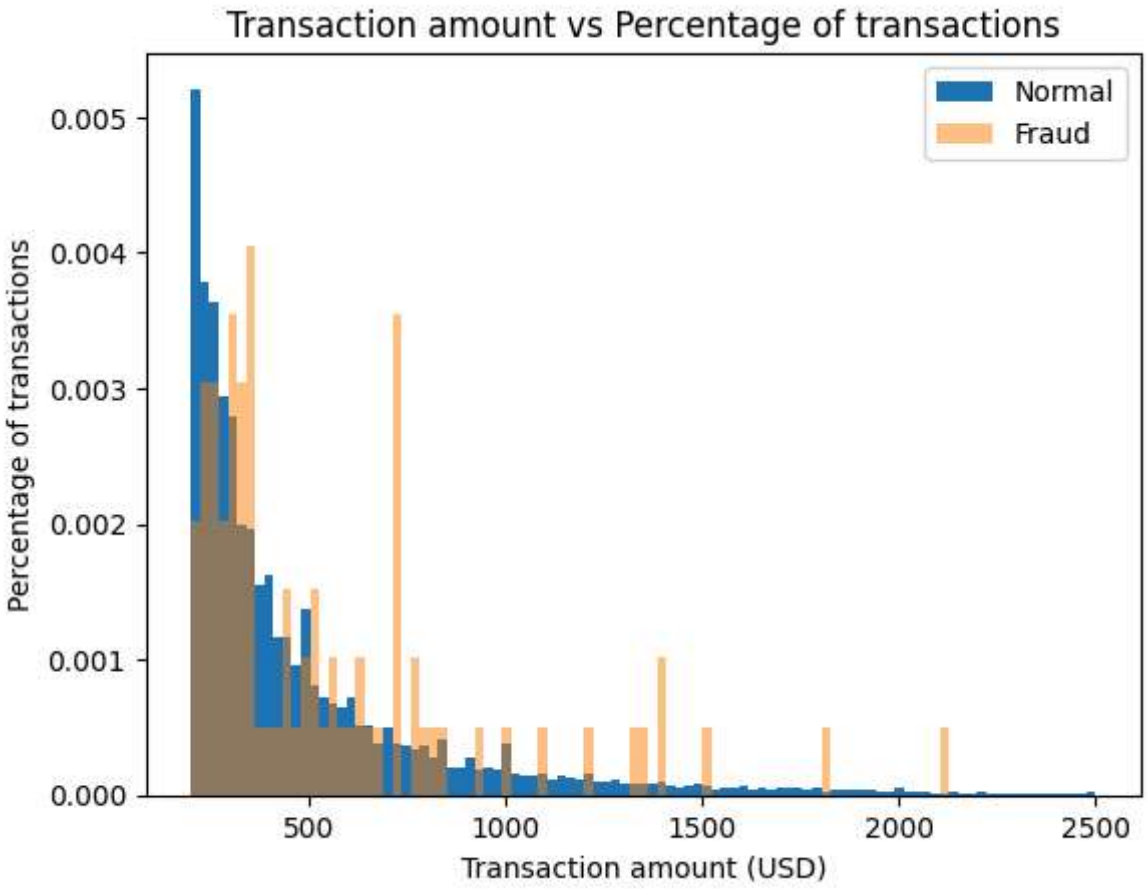
```
C:\Users\SMIT DESHMUKH\AppData\Local\Temp\ipykernel_21400\2454488302.py:10: FutureWa
rning: pandas.value_counts is deprecated and will be removed in a future version. Us
e pd.Series(obj).value_counts() instead.
  print(pd.value_counts(dataset['Class'], sort = True) )
```

In [5]:
```
#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");
```

```
C:\Users\SMIT DESHMUKH\AppData\Local\Temp\ipykernel_21400\3399164077.py:2: FutureWar
ning: pandas.value_counts is deprecated and will be removed in a future version. Use
pd.Series(obj).value_counts() instead.
  count_classes = pd.value_counts(dataset['Class'], sort = True)
```



In [6]:
```
# Save the normal and fradulent transactions in separate dataframe
normal_dataset = dataset[dataset.Class == 0]
fraud_dataset = dataset[dataset.Class == 1]
#Visualize transactionamounts for normal and fraudulent transactions
bins = np.linspace(200, 2500, 100) # Return evenly spaced numbers over a specified
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()
```



In [8]:
```
sc=StandardScaler()
dataset['Time'] = sc.fit_transform(dataset['Time'].values.reshape(-1, 1))
dataset['Amount'] = sc.fit_transform(dataset['Amount'].values.reshape(-1, 1))
```

In [9]:
```
dataset['Amount']
```

Out[9]:
```
0         0.244964
1        -0.342475
2         1.160686
3         0.140534
4        -0.073403
            ...
284802   -0.350151
284803   -0.254117
284804   -0.081839
284805   -0.313249
284806    0.514355
Name: Amount, Length: 284807, dtype: float64
```

In [10]:
```
train_x, test_x = train_test_split(dataset, test_size=TEST_PCT, random_state=RANDOM
train_x = train_x[train_x.Class == 0]       # where normal transactions
train_x = train_x.drop(['Class'], axis=1)   # drop the class column


test_y = test_x['Class']                    # save the class column for the test se
test_x = test_x.drop(['Class'], axis=1)     # drop the class column

train_x = train_x.values                    # transform to ndarray
test_x = test_x.values                      # transform to ndarray
```

Autoencoder Layer Structure and Parameters

Autoencoder has symmetric encoding and decoding layers that are "dense". We are reducing the input into some form of simplified encoding and then expanding it again. The input and output dimension is the feature space (e.g. 30 columns), so the encoding layer should be smaller by an amount that expect to represent some feature. In this case, I am encoding 30 columns into 14 dimensions so I am expecting high-level features to be represented by roughly two columns (30/14 = 2.1). Of those high-level features, I am expecting them to map to roughly seven hidden/latent features in the data.

Additionally, the epochs, batch size, learning rate, learning policy, and activation functions were all set to values empirically good values.

In [11]:
```
nb_epoch = 50
batch_size = 64
input_dim = train_x.shape[1] #num of columns, 30
encoding_dim = 14
hidden_dim_1 = int(encoding_dim / 2) #
hidden_dim_2=4
learning_rate = 1e-7
```

In [12]:
```
#input Layer
input_layer = tf.keras.layers.Input(shape=(input_dim, ))

#Encoder
encoder = tf.keras.layers.Dense(encoding_dim, activation="tanh",
                                activity_regularizer=tf.keras.regularizers.l2(learning_rate
# encoder=tf.keras.layers.Dropout(0.2)(encoder)
```

```
encoder = tf.keras.layers.Dense(hidden_dim_1, activation='relu')(encoder)
encoder = tf.keras.layers.Dense(hidden_dim_2, activation=tf.nn.leaky_relu)(encoder)

# Decoder
decoder = tf.keras.layers.Dense(hidden_dim_1, 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
autoencoder = tf.keras.Model(inputs=input_layer, outputs=decoder)
autoencoder.summary()
```

Model: "functional"

Layer (type)	Output Shape	
input_layer (InputLayer)	(None, 30)	
dense (Dense)	(None, 14)	
dense_1 (Dense)	(None, 7)	
dense_2 (Dense)	(None, 4)	
dense_3 (Dense)	(None, 7)	
dense_4 (Dense)	(None, 14)	
dense_5 (Dense)	(None, 30)	

Total params: 1,168 (4.56 KB)

Trainable params: 1,168 (4.56 KB)

Non-trainable params: 0 (0.00 B)

In [13]:
```
cp = tf.keras.callbacks.ModelCheckpoint(filepath="autoencoder_fraud.h5",
                                mode='min', monitor='val_loss', verbose=2, save_best
# define our early stopping
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    min_delta=0.0001,
    patience=10,
    verbose=1,
    mode='min',
    restore_best_weights=True)
```

In [14]:
```
#Compile the Autoencoder

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

In [15]:
```
#Train the Autoencoder

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

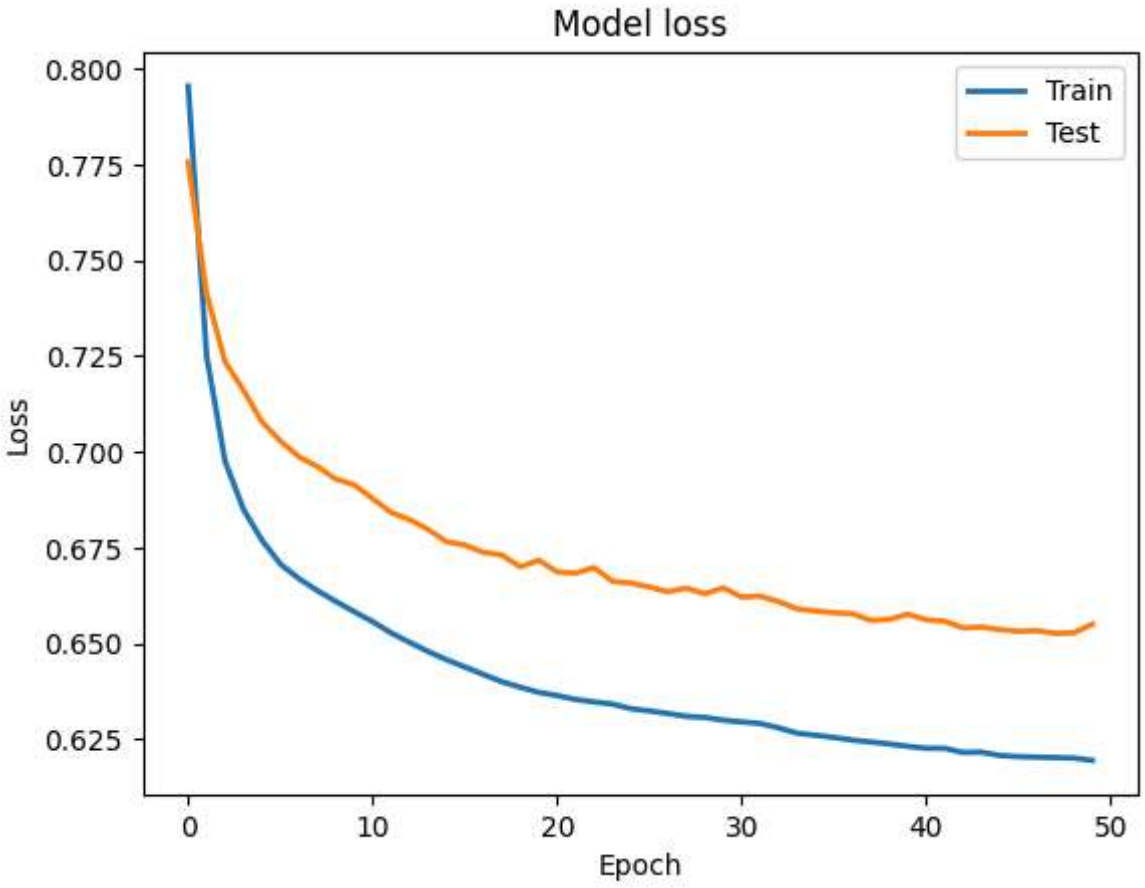
```
Epoch 1/50
3105/3110 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2923 - loss: 0.8646
Epoch 1: val_loss improved from None to 0.77564, saving model to autoencoder_fraud.h
5
3110/3110 ━━━━━━━━━━━━━━━━━━━━ 19s 5ms/step - accuracy: 0.3691 - loss: 0.7954 - val_
accuracy: 0.4247 - val_loss: 0.7756
Epoch 2/50
3105/3110 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.4269 - loss: 0.7345
Epoch 2: val_loss improved from 0.77564 to 0.74126, saving model to autoencoder_frau
d.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.
saving.save_model(model)`. This file format is considered legacy. We recommend using
instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.savin
g.save_model(model, 'my_model.keras')`.
3110/3110 ━━━━━━━━━━━━━━━━━━━━ 17s 5ms/step - accuracy: 0.4285 - loss: 0.7250 - val_
accuracy: 0.4248 - val_loss: 0.7413
Epoch 3/50
3095/3110 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4184 - loss: 0.6950
Epoch 3: val_loss improved from 0.74126 to 0.72348, saving model to autoencoder_frau
d.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.
saving.save_model(model)`. This file format is considered legacy. We recommend using
instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.savin
g.save_model(model, 'my_model.keras')`.
3110/3110 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - accuracy: 0.4138 - loss: 0.6975 - val_
accuracy: 0.4064 - val_loss: 0.7235
Epoch 4/50
3093/3110 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4043 - loss: 0.6704
Epoch 4: val_loss improved from 0.72348 to 0.71604, saving model to autoencoder_frau
d.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.
saving.save_model(model)`. This file format is considered legacy. We recommend using
instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.savin
g.save_model(model, 'my_model.keras')`.
3110/3110 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - accuracy: 0.4043 - loss: 0.6849 - val_
accuracy: 0.4016 - val_loss: 0.7160
Epoch 5/50
3094/3110 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4034 - loss: 0.6773
Epoch 5: val_loss improved from 0.71604 to 0.70776, saving model to autoencoder_frau
d.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.
saving.save_model(model)`. This file format is considered legacy. We recommend using
instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.savin
g.save_model(model, 'my_model.keras')`.
```

```
0.4346 - val_loss: 0.6473 - val_accuracy: 0.4369
Epoch 44/50
3108/3110 [============================>.] - ETA: 0s - loss: 0.6156 - accuracy: 0.43
46
Epoch 44: val_loss did not improve from 0.64682
3110/3110 [==============================] - 7s 2ms/step - loss: 0.6155 - accuracy:
0.4347 - val_loss: 0.6473 - val_accuracy: 0.4360
Epoch 45/50
3089/3110 [============================>.] - ETA: 0s - loss: 0.6157 - accuracy: 0.43
45
Epoch 45: val_loss did not improve from 0.64682
3110/3110 [==============================] - 7s 2ms/step - loss: 0.6154 - accuracy:
0.4348 - val_loss: 0.6471 - val_accuracy: 0.4335
Epoch 46/50
3090/3110 [============================>.] - ETA: 0s - loss: 0.6156 - accuracy: 0.43
49
Epoch 46: val_loss did not improve from 0.64682
3110/3110 [==============================] - 8s 2ms/step - loss: 0.6153 - accuracy:
0.4348 - val_loss: 0.6480 - val_accuracy: 0.4354
Epoch 47/50
3096/3110 [============================>.] - ETA: 0s - loss: 0.6153 - accuracy: 0.43
43
Epoch 47: val_loss did not improve from 0.64682
3110/3110 [==============================] - 8s 2ms/step - loss: 0.6152 - accuracy:
0.4343 - val_loss: 0.6471 - val_accuracy: 0.4335
Epoch 48/50
3094/3110 [============================>.] - ETA: 0s - loss: 0.6160 - accuracy: 0.43
39
Epoch 48: val_loss did not improve from 0.64682
3110/3110 [==============================] - 7s 2ms/step - loss: 0.6151 - accuracy:
0.4340 - val_loss: 0.6482 - val_accuracy: 0.4374
Epoch 49/50
3104/3110 [============================>.] - ETA: 0s - loss: 0.6148 - accuracy: 0.43
42
Epoch 49: val_loss improved from 0.64682 to 0.64680, saving model to autoencoder_fra
ud.h5
3110/3110 [==============================] - 7s 2ms/step - loss: 0.6149 - accuracy:
0.4342 - val_loss: 0.6468 - val_accuracy: 0.4384
Epoch 50/50
3086/3110 [============================>.] - ETA: 0s - loss: 0.6139 - accuracy: 0.43
35
Epoch 50: val_loss did not improve from 0.64680
3110/3110 [==============================] - 7s 2ms/step - loss: 0.6147 - accuracy:
0.4337 - val_loss: 0.6477 - val_accuracy: 0.4380
```

In [16]:

```
#Plot training and test 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.ylim(ymin=0.70,ymax=1)
plt.show()
```



In [17]:
```
test_x_predictions = autoencoder.predict(test_x)
mse = np.mean(np.power(test_x - test_x_predictions, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse,
                         'True_class': test_y})
error_df.describe()
```

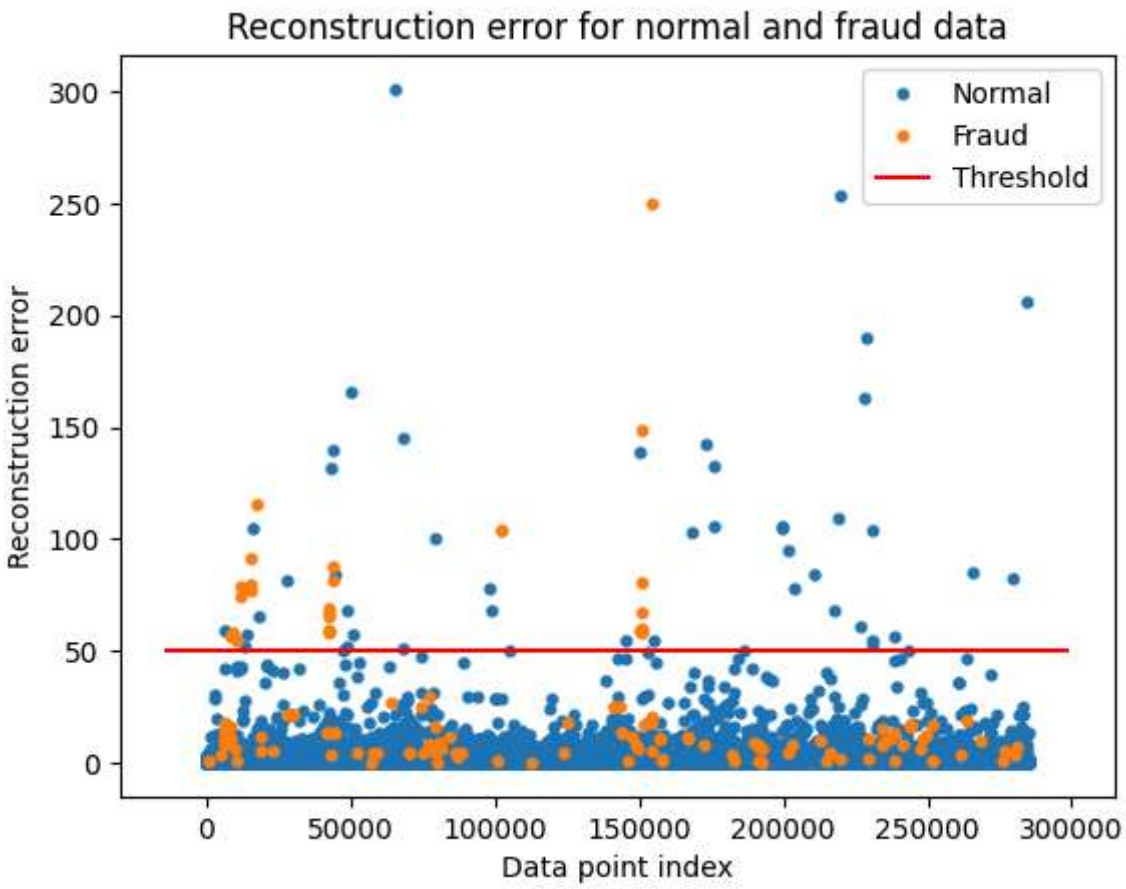
```
2671/2671 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step
```

Out[17]:

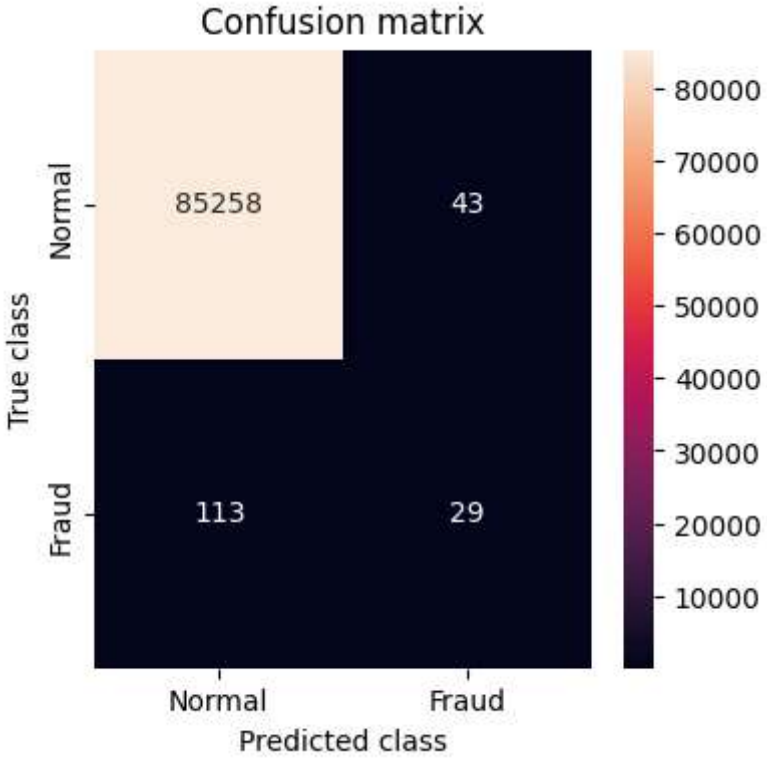
	Reconstruction_error	True_class
count	85443.000000	85443.000000
mean	0.652507	0.001662
std	3.567793	0.040733
min	0.017773	0.000000
25%	0.152447	0.000000
50%	0.263808	0.000000
75%	0.489267	0.000000
max	301.152230	1.000000

In [18]:
```
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=1
ax.legend()
plt.title("Reconstruction error for normal and fraud data")
plt.ylabel("Reconstruction error")
plt.xlabel("Data point index")
plt.show();
```



In [19]:
```
import seaborn as sns
threshold_fixed =52
pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.va
error_df['pred'] =pred_y
conf_matrix = confusion_matrix(error_df.True_class, pred_y)
plt.figure(figsize=(4, 4))
sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, 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']))
```




```
Accuracy:  0.9981742214107651
Recall:  0.20422535211267606
Precision:  0.4027777777777778
```