```
""" DL ASSIGNMENT
ML | Credit Card Fraud Detection
-> The challenge is to recognize fraudulent credit card transactions
so that the customers of credit card companies
are not charged for items that they did not purchase.
Main challenges involved in credit card fraud detection are:
1.Enormous Data is processed every day and the model build must be
fast enough to respond to the scam in time.
2.Imbalanced Data i.e most of the transactions (99.8%) are not
fraudulent which makes it really hard for detecting the
fraudulent ones
3.Data availability as the data is mostly private.
4.Misclassified Data can be another major issue, as not every
fraudulent transaction is caught and reported.
5.Adaptive techniques used against the model by the scammers.
"""
```

```
from google.colab import files
```

```
uploaded = files.upload()
```

Choose Files creditcard.csv

- **creditcard.csv**(text/csv) - 150828752 bytes, last modified: 9/25/2022 - 100% done

```
Saving creditcard.csv to creditcard.csv
```

```
#Setup
#We will be using TensorFlow 1.2 and Keras 2.0.4.
import pandas as pd
import numpy as np
import pickle
import matplotlib.pyplot as plt
from scipy import stats
import tensorflow as tf
import seaborn as sns
from pylab import rcParams
from sklearn.model_selection import train_test_split
from keras.models import Model, load_model
from keras.layers import Input, Dense
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras import regularizers

%matplotlib inline

sns.set(style='whitegrid', palette='muted', font_scale=1.5)

rcParams['figure.figsize'] = 14, 8

RANDOM_SEED = 42
LABELS = ["Normal", "Fraud"]
```

```
"""we will train an Autoencoder Neural Network (implemented in Keras) in unsupervised (or
 The trained model will be evaluated on pre-labeled and anonymized dataset."""


#Loading the data
#The dateset contains data about credit card transactions that occurred during a period of

"""All variables in the dataset are numerical. The data has been transformed using PCA tra
changed are Time and Amount. Time contains the seconds elapsed between each transaction an


import pandas as pd
import io

df = pd.read_csv(io.StringIO(uploaded['creditcard.csv'].decode('utf-8')))
print(df)
```

```
            Time         V1         V2        V3        V4        V5  \
0            0.0  -1.359807  -0.072781  2.536347  1.378155 -0.338321
1            0.0   1.191857   0.266151  0.166480  0.448154  0.060018
2            1.0  -1.358354  -1.340163  1.773209  0.379780 -0.503198
3            1.0  -0.966272  -0.185226  1.792993 -0.863291 -0.010309
4            2.0  -1.158233   0.877737  1.548718  0.403034 -0.407193
...          ...        ...        ...       ...       ...       ...
284802  172786.0 -11.881118  10.071785 -9.834783 -2.066656 -5.364473
284803  172787.0  -0.732789  -0.055080  2.035030 -0.738589  0.868229
284804  172788.0   1.919565  -0.301254 -3.249640 -0.557828  2.630515
284805  172788.0  -0.240440   0.530483  0.702510  0.689799 -0.377961
284806  172792.0  -0.533413  -0.189733  0.703337 -0.506271 -0.012546

              V6        V7        V8        V9  ...       V21       V22  \
0       0.462388  0.239599  0.098698  0.363787  ... -0.018307  0.277838
1      -0.082361 -0.078803  0.085102 -0.255425  ... -0.225775 -0.638672
2       1.800499  0.791461  0.247676 -1.514654  ...  0.247998  0.771679
3       1.247203  0.237609  0.377436 -1.387024  ... -0.108300  0.005274
4       0.095921  0.592941 -0.270533  0.817739  ... -0.009431  0.798278
...          ...       ...       ...       ...  ...       ...       ...
284802 -2.606837 -4.918215  7.305334  1.914428  ...  0.213454  0.111864
284803  1.058415  0.024330  0.294869  0.584800  ...  0.214205  0.924384
284804  3.031260 -0.296827  0.708417  0.432454  ...  0.232045  0.578229
284805  0.623708 -0.686180  0.679145  0.392087  ...  0.265245  0.800049
284806 -0.649617  1.577006 -0.414650  0.486180  ...  0.261057  0.643078

             V23       V24       V25       V26       V27       V28  Amount  \
0      -0.110474  0.066928  0.128539 -0.189115  0.133558 -0.021053  149.62
1       0.101288 -0.339846  0.167170  0.125895 -0.008983  0.014724    2.69
2       0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752  378.66
3      -0.190321 -1.175575  0.647376 -0.221929  0.062723  0.061458  123.50
4      -0.137458  0.141267 -0.206010  0.502292  0.219422  0.215153   69.99
...          ...       ...       ...       ...       ...       ...     ...
284802  1.014480 -0.509348  1.436807  0.250034  0.943651  0.823731    0.77
284803  0.012463 -1.016226 -0.606624 -0.395255  0.068472 -0.053527   24.79
284804 -0.037501  0.640134  0.265745 -0.087371  0.004455 -0.026561   67.88
284805 -0.163298  0.123205 -0.569159  0.546668  0.108821  0.104533   10.00
284806  0.376777  0.008797 -0.473649 -0.818267 -0.002415  0.013649  217.00

        Class
```

```
0         0
1         0
2         0
3         0
4         0
...     ...
284802    0
284803    0
284804    0
284805    0
284806    0

[284807 rows x 31 columns]
```

```
#Exploration
df.shape
```

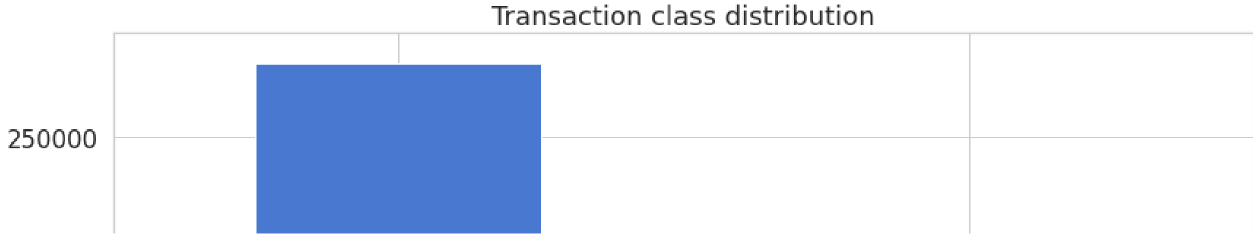
```
(284807, 31)
```

```
#31 columns, 2 of which are Time and Amount. The rest are output from the PCA transformati
```

```
df.isnull().values.any()
```

```
False
```

```
count_classes = pd.value_counts(df['Class'], sort = True)
count_classes.plot(kind = 'bar', rot=0)
plt.title("Transaction class distribution")
plt.xticks(range(2), LABELS)
plt.xlabel("Class")
plt.ylabel("Frequency");
```




```
#We have a highly imbalanced dataset on our hands. Normal transactions overwhelm the fraud
```

```
frauds = df[df.Class == 1]
normal = df[df.Class == 0]
```

```
frauds.shape
```

```
(492, 31)
```

```
normal.shape
```

```
(284315, 31)
```

```
#How different are the amount of money used in different transaction classes?
```

```
frauds.Amount.describe()
```

```
count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

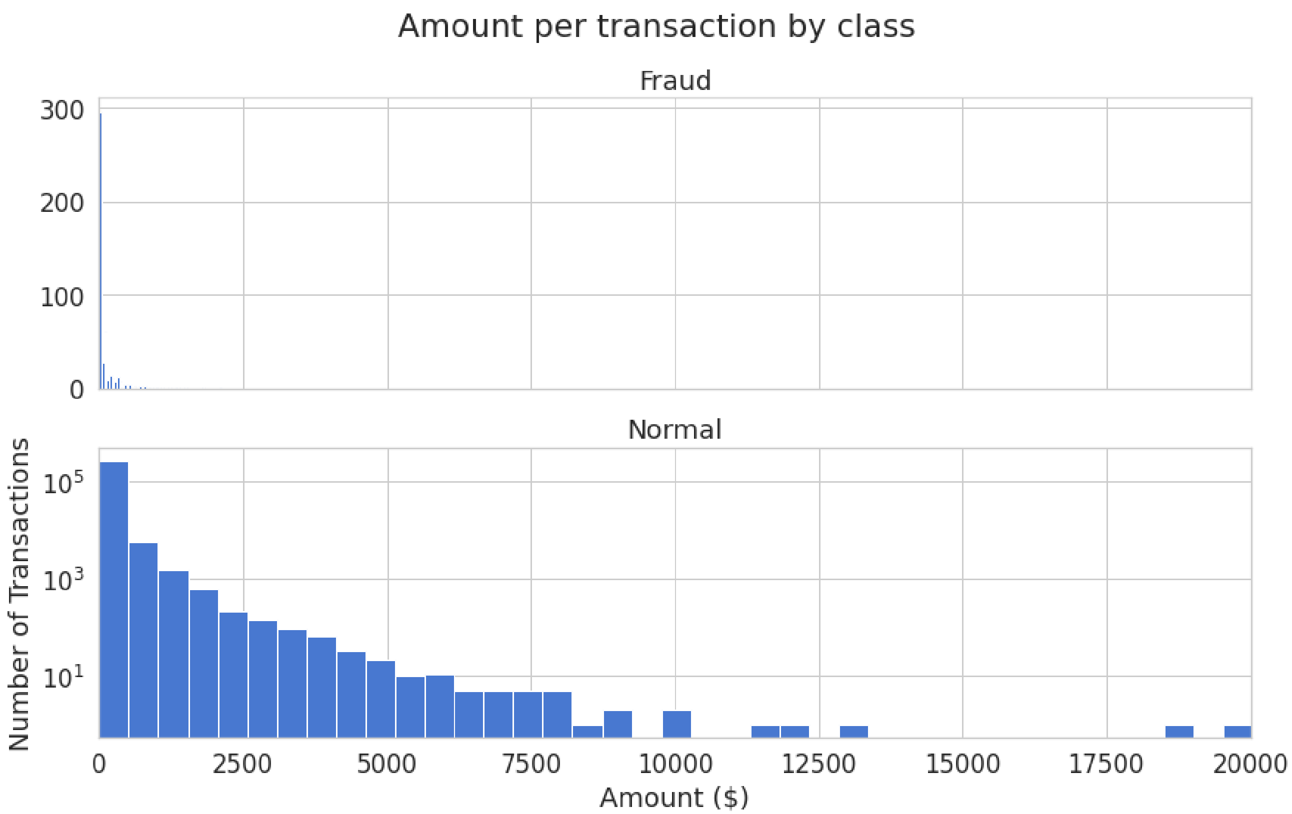
```
normal.Amount.describe()
```

```
count    284315.000000
mean         88.291022
std         250.105092
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
Name: Amount, dtype: float64
```

```
#Let's have a more graphical representation:
```

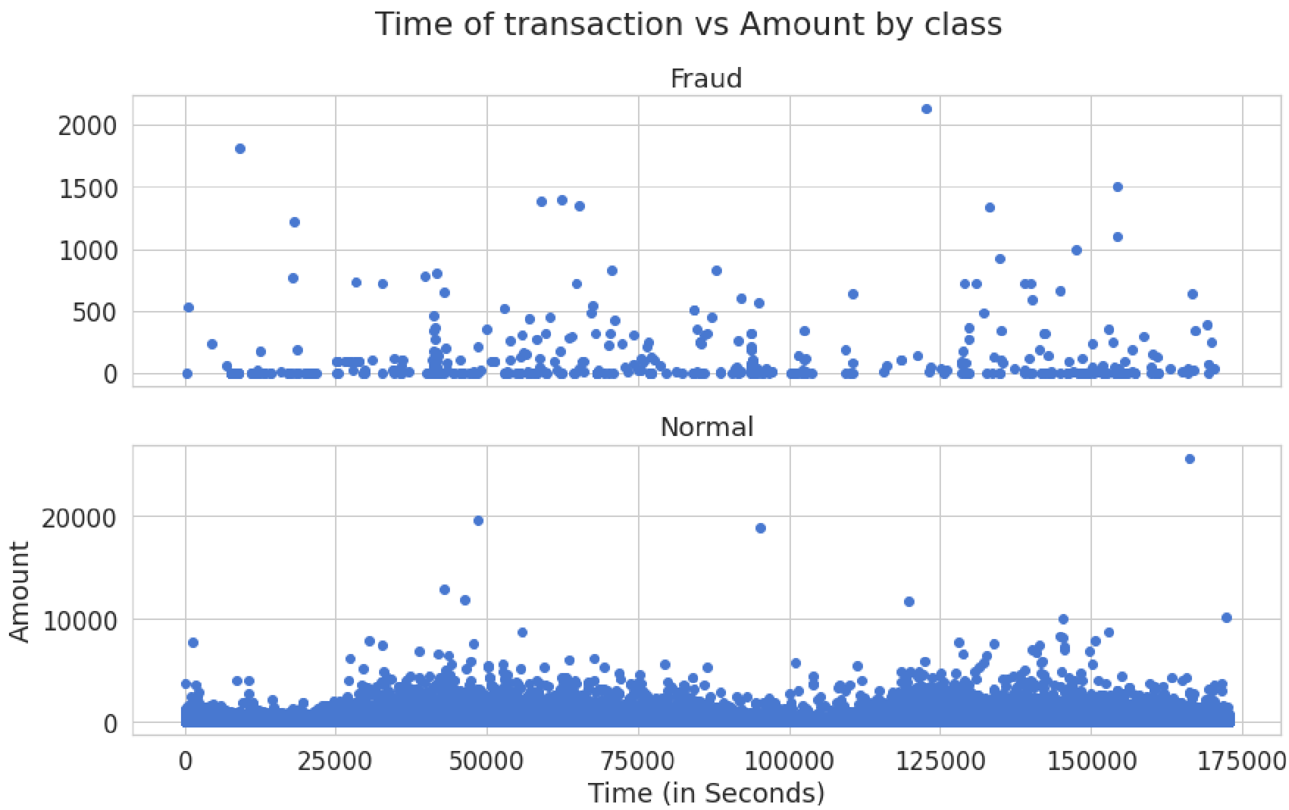
```
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
f.suptitle('Amount per transaction by class')
```

```
bins = 50

ax1.hist(frauds.Amount, bins = bins)
ax1.set_title('Fraud')

ax2.hist(normal.Amount, bins = bins)
ax2.set_title('Normal')

plt.xlabel('Amount ($)')
plt.ylabel('Number of Transactions')
plt.xlim((0, 20000))
plt.yscale('log')
plt.show();
```




```
#Do fraudulent transactions occur more often during certain time?


f, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
f.suptitle('Time of transaction vs Amount by class')

ax1.scatter(frauds.Time, frauds.Amount)
ax1.set_title('Fraud')

ax2.scatter(normal.Time, normal.Amount)
```

```
ax2.set_title('Normal')

plt.xlabel('Time (in Seconds)')
plt.ylabel('Amount')
plt.show()
```




```
"""Autoencoders
Autoencoders can seem quite bizarre at first. The job of those models is to predict the in

More specifically, let’s take a look at Autoencoder Neural Networks. This autoencoder trie

While trying to do just that might sound trivial at first, it is important to note that we
 This can be done by limiting the number of hidden units in the model. Those kind of autoe
"""


"""
---------Reconstruction error-----------
We optimize the parameters of our Autoencoder model in such way that a special kind of err


"""
--------Preparing the data---------
First, let's drop the Time column (not going to use it) and use the scikit's StandardScale
```

```
from sklearn.preprocessing import StandardScaler

data = df.drop(['Time'], axis=1)

data['Amount'] = StandardScaler().fit_transform(data['Amount'].values.reshape(-1, 1))


X_train, X_test = train_test_split(data, test_size=0.2, random_state=RANDOM_SEED)
X_train = X_train[X_train.Class == 0]
X_train = X_train.drop(['Class'], axis=1)

y_test = X_test['Class']
X_test = X_test.drop(['Class'], axis=1)

X_train = X_train.values
X_test = X_test.values


X_train.shape
```

```
(227451, 29)
```

```
"""
---------Building the model------------
Our Autoencoder uses 4 fully connected layers with 14, 7, 7 and 29 neurons respectively. T
 Additionally, L1 regularization will be used during training:"""


input_dim = X_train.shape[1]
encoding_dim = 14


input_layer = Input(shape=(input_dim, ))

encoder = Dense(encoding_dim, activation="tanh",
                activity_regularizer=regularizers.l1(10e-5))(input_layer)
encoder = Dense(int(encoding_dim / 2), activation="relu")(encoder)

decoder = Dense(int(encoding_dim / 2), activation='tanh')(encoder)
decoder = Dense(input_dim, activation='relu')(decoder)

autoencoder = Model(inputs=input_layer, outputs=decoder)


"""Let's train our model for 100 epochs with a batch size of 32 samples and save the best
 The ModelCheckpoint provided by Keras is really handy for such tasks. Additionally, the t


nb_epoch = 100
batch_size = 32

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

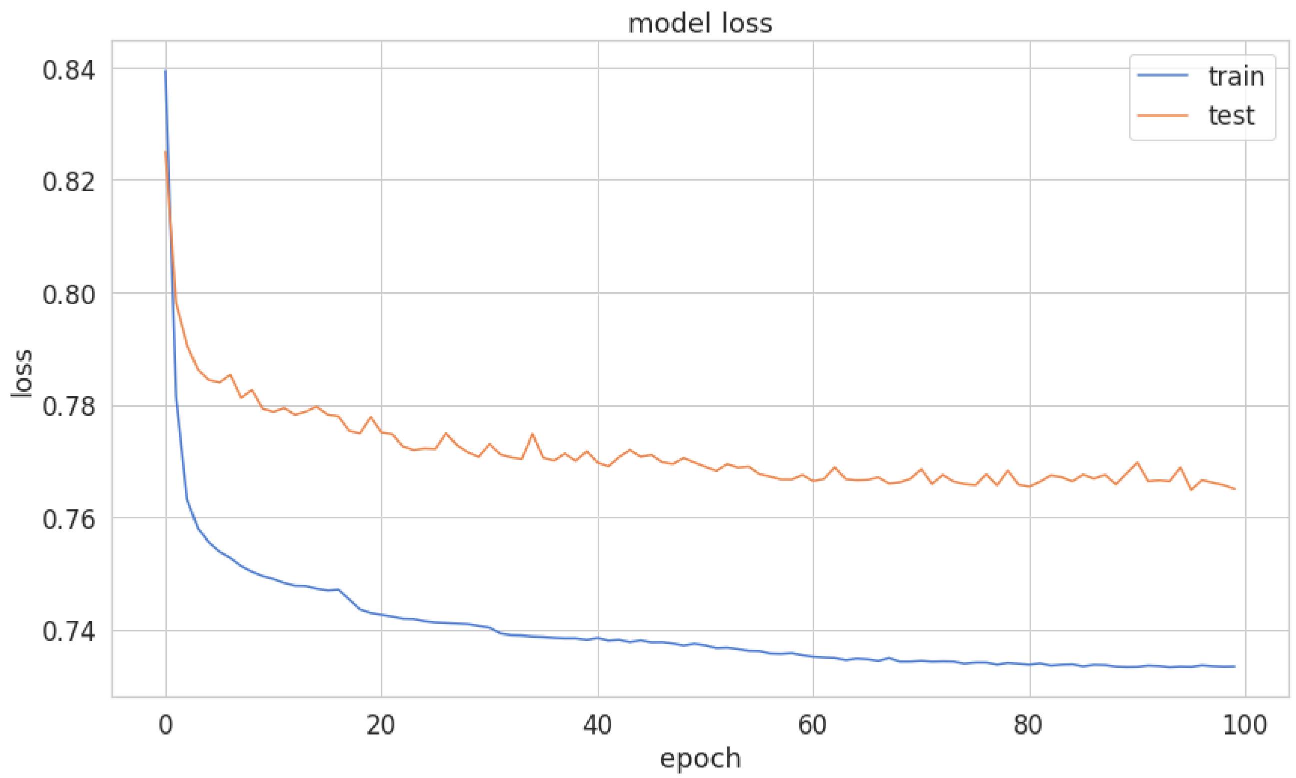
```
checkpointer = ModelCheckpoint(filepath="model.h5",
                               verbose=0,
                               save_best_only=True)
tensorboard = TensorBoard(log_dir='./logs',
                          histogram_freq=0,
                          write_graph=True,
                          write_images=True)

history = autoencoder.fit(X_train, X_train,
                    epochs=nb_epoch,
                    batch_size=batch_size,
                    shuffle=True,
                    validation_data=(X_test, X_test),
                    verbose=1,
                    callbacks=[checkpointer, tensorboard]).history
```

```
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7345 - accurac
Epoch 73/100
7108/7108 [==============================] - 15s 2ms/step - loss: 0.7344 - accurac
Epoch 74/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7343 - accurac
Epoch 75/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7340 - accurac
Epoch 76/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7342 - accurac
Epoch 77/100
7108/7108 [==============================] - 15s 2ms/step - loss: 0.7342 - accurac
Epoch 78/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7338 - accurac
Epoch 79/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7341 - accurac
Epoch 80/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7339 - accurac
Epoch 81/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7338 - accurac
Epoch 82/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7340 - accurac
Epoch 83/100
7108/7108 [==============================] - 15s 2ms/step - loss: 0.7336 - accurac
Epoch 84/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7338 - accurac
Epoch 85/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7338 - accurac
Epoch 86/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7335 - accurac
Epoch 87/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7337 - accurac
Epoch 88/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7337 - accurac
Epoch 89/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7334 - accurac
Epoch 90/100
7108/7108 [==============================] - 15s 2ms/step - loss: 0.7333 - accurac
Epoch 91/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7334 - accurac
Epoch 92/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7336 - accurac
Epoch 93/100
```

```
7108/7108 [==============================] - 15s 2ms/step - loss: 0.7335 - accurac
Epoch 94/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7333 - accurac
Epoch 95/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7334 - accurac
Epoch 96/100
7108/7108 [==============================] - 21s 3ms/step - loss: 0.7334 - accurac
Epoch 97/100
7108/7108 [==============================] - 17s 2ms/step - loss: 0.7337 - accurac
Epoch 98/100
7108/7108 [==============================] - 17s 2ms/step - loss: 0.7335 - accurac
Epoch 99/100
7108/7108 [==============================] - 19s 3ms/step - loss: 0.7334 - accurac
Epoch 100/100
7108/7108 [==============================] - 17s 2ms/step - loss: 0.7334 - accurac
```

```
autoencoder = load_model('model.h5')
```

```
#Evaluation
plt.plot(history['loss'])
plt.plot(history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right');
```



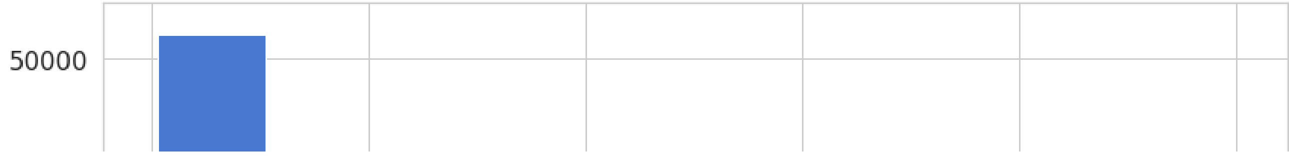
```
predictions = autoencoder.predict(X_test)
```

```
mse = np.mean(np.power(X_test - predictions, 2), axis=1)
error_df = pd.DataFrame({'reconstruction_error': mse,
                        'true_class': y_test})
```

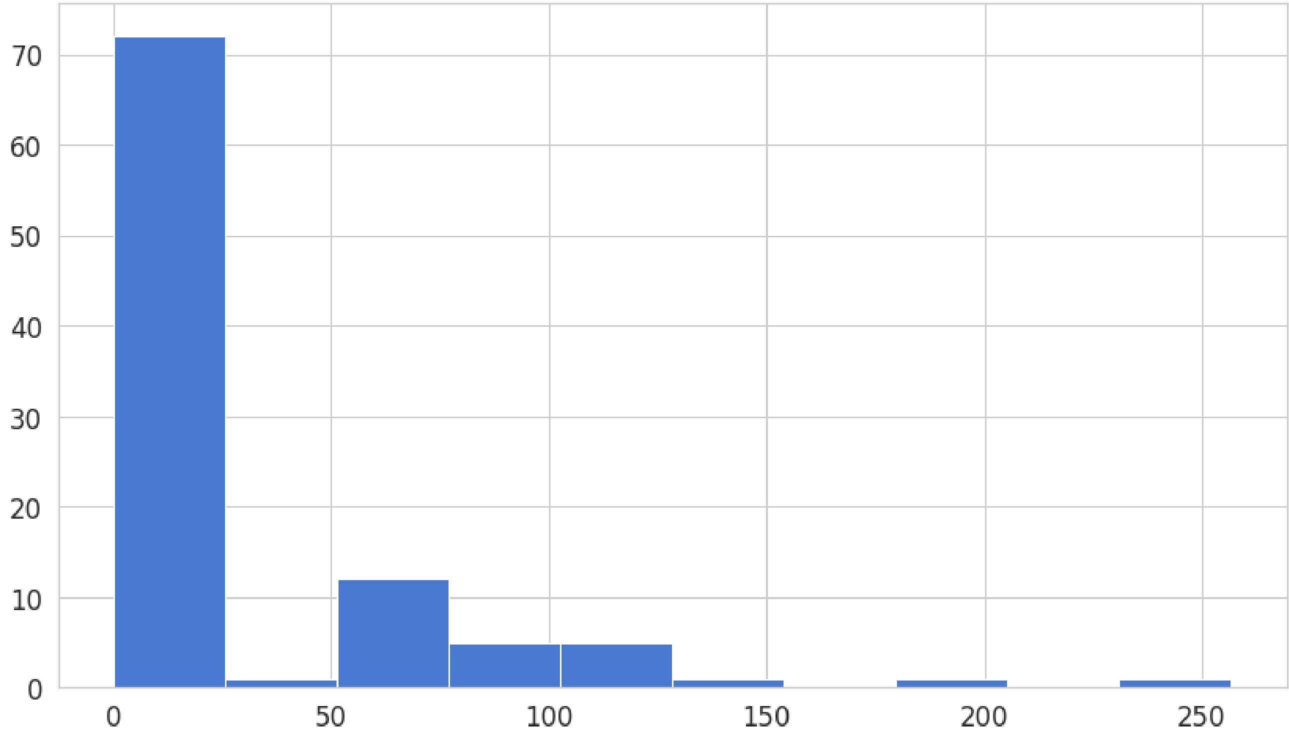
```
error_df.describe()
```

	reconstruction_error	true_class
count	56962.000000	56962.000000
mean	0.763890	0.001720
std	3.365745	0.041443
min	0.051223	0.000000
25%	0.283433	0.000000
50%	0.436176	0.000000
75%	0.660284	0.000000
max	256.665093	1.000000

```
#Reconstruction error without fraud
fig = plt.figure()
ax = fig.add_subplot(111)
normal_error_df = error_df[(error_df['true_class']== 0) & (error_df['reconstruction_error'
_ = ax.hist(normal_error_df.reconstruction_error.values, bins=10)
```




```
#Reconstruction error with fraud
fig = plt.figure()
ax = fig.add_subplot(111)
fraud_error_df = error_df[error_df['true_class'] == 1]
_ = ax.hist(fraud_error_df.reconstruction_error.values, bins=10)
```

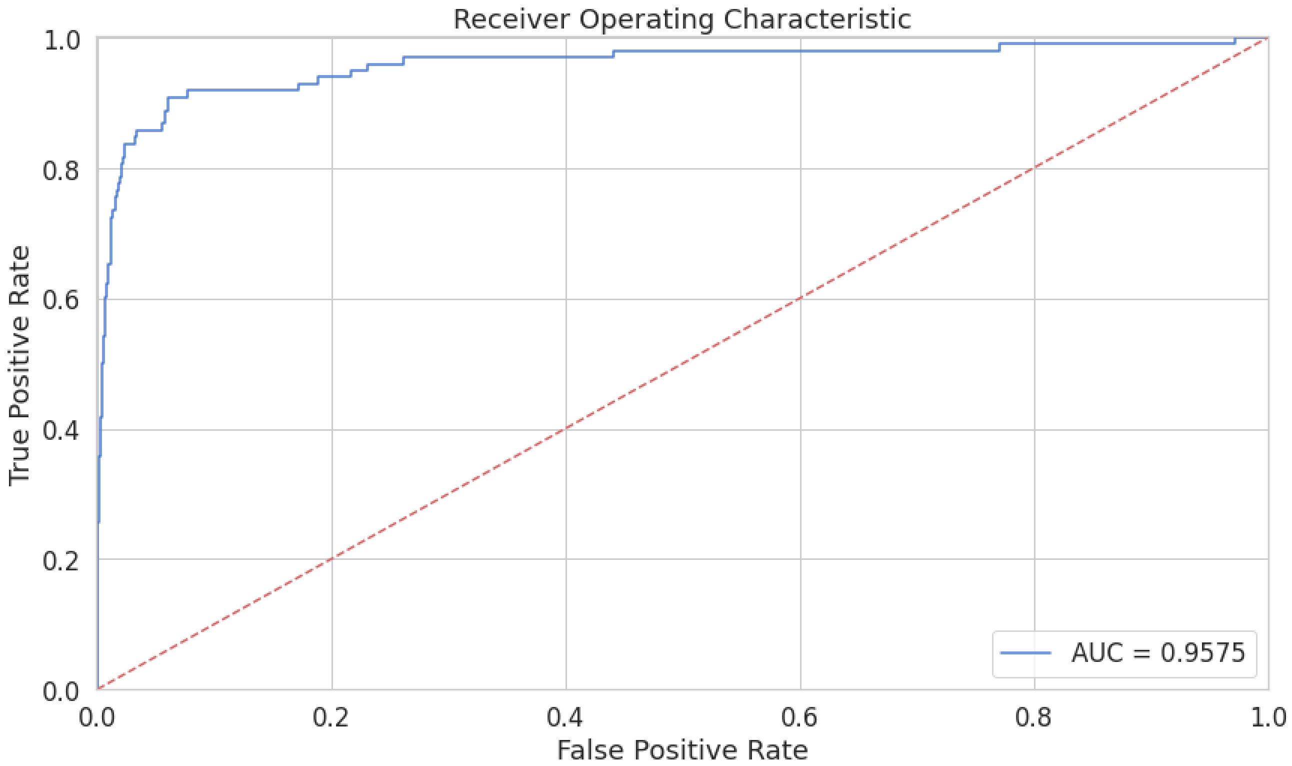



```
from sklearn.metrics import (confusion_matrix, precision_recall_curve, auc,
                             roc_curve, recall_score, classification_report, f1_score,
                             precision_recall_fscore_support)
```

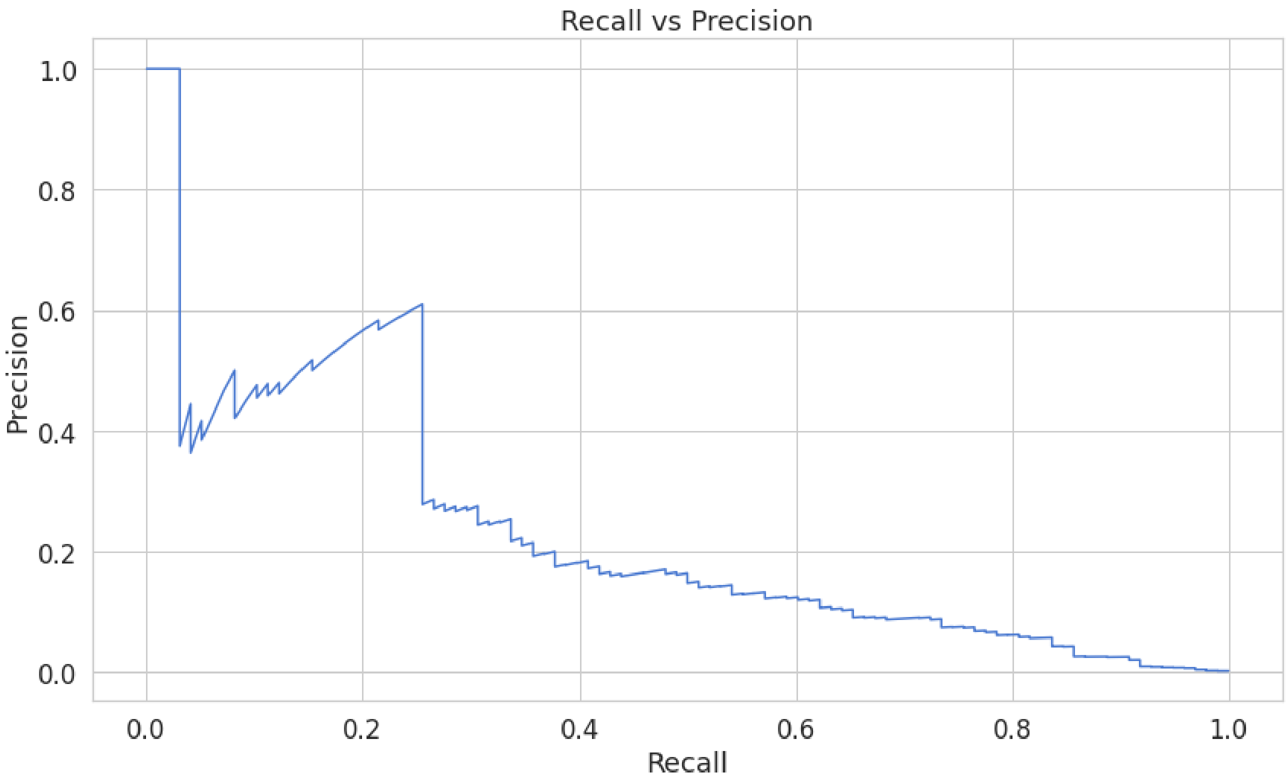
```
"""ROC curves are very useful tool for understanding the performance of binary classifiers
We have a very imbalanced dataset. Nonetheless, let's have a look at our ROC curve:"""
```

```
fpr, tpr, thresholds = roc_curve(error_df.true_class, error_df.reconstruction_error)
roc_auc = auc(fpr, tpr)

plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, label='AUC = %0.4f'% roc_auc)
plt.legend(loc='lower right')
plt.plot([0,1],[0,1],'r--')
plt.xlim([-0.001, 1])
```

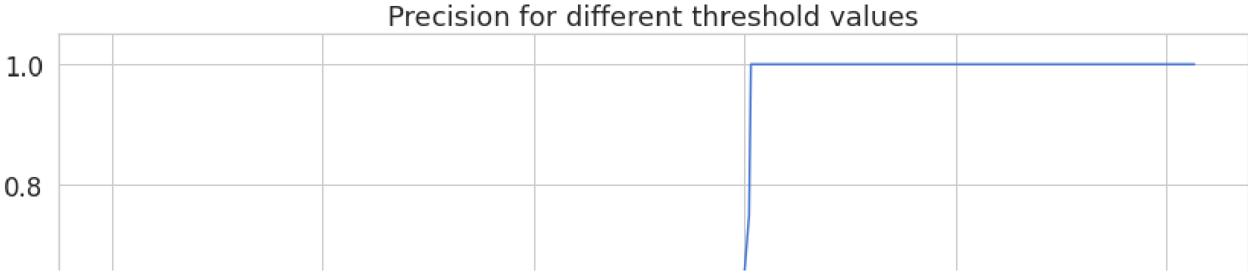
```
plt.ylim([0, 1.001])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show();
```



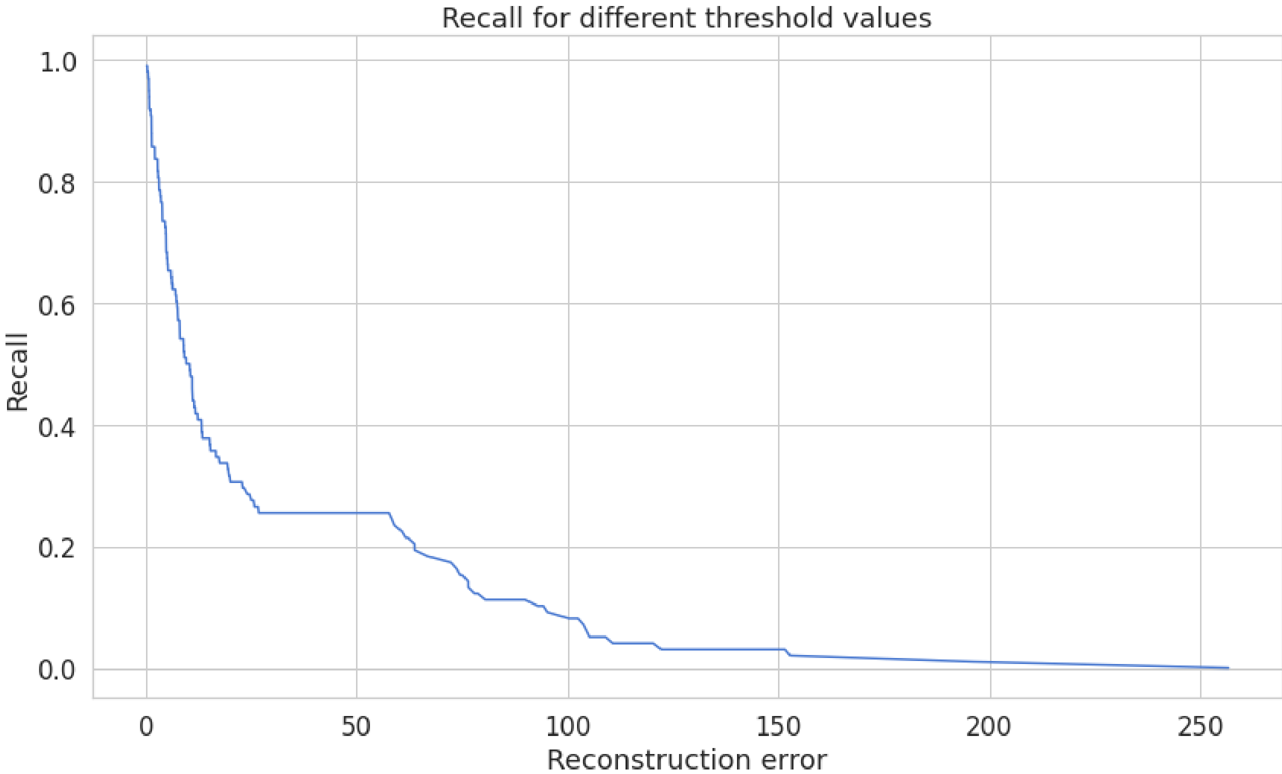

```
"""The ROC curve plots the true positive rate versus the false positive rate, over differe
 Basically, we want the blue line to be as close as possible to the upper left corner."""


"""Let's return to our example from Information Retrieval. High recall but low precision m
When precision is high but recall is low we have the opposite - few returned results with
Ideally, you would want high precision and high recall - many results with that are highly


precision, recall, th = precision_recall_curve(error_df.true_class, error_df.reconstructio
plt.plot(recall, precision, 'b', label='Precision-Recall curve')
plt.title('Recall vs Precision')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()
```




```
"""A high area under the curve represents both high recall and high precision, where high
and high recall relates to a low false negative rate. High scores for both show that the c
a majority of all positive results (high recall)."""


plt.plot(th, precision[1:], 'b', label='Threshold-Precision curve')
plt.title('Precision for different threshold values')
plt.xlabel('Threshold')
plt.ylabel('Precision')
plt.show()
```




```
#You can see that as the reconstruction error increases our precision rises as well. Let's

plt.plot(th, recall[1:], 'b', label='Threshold-Recall curve')
plt.title('Recall for different threshold values')
plt.xlabel('Reconstruction error')
plt.ylabel('Recall')
plt.show()
```

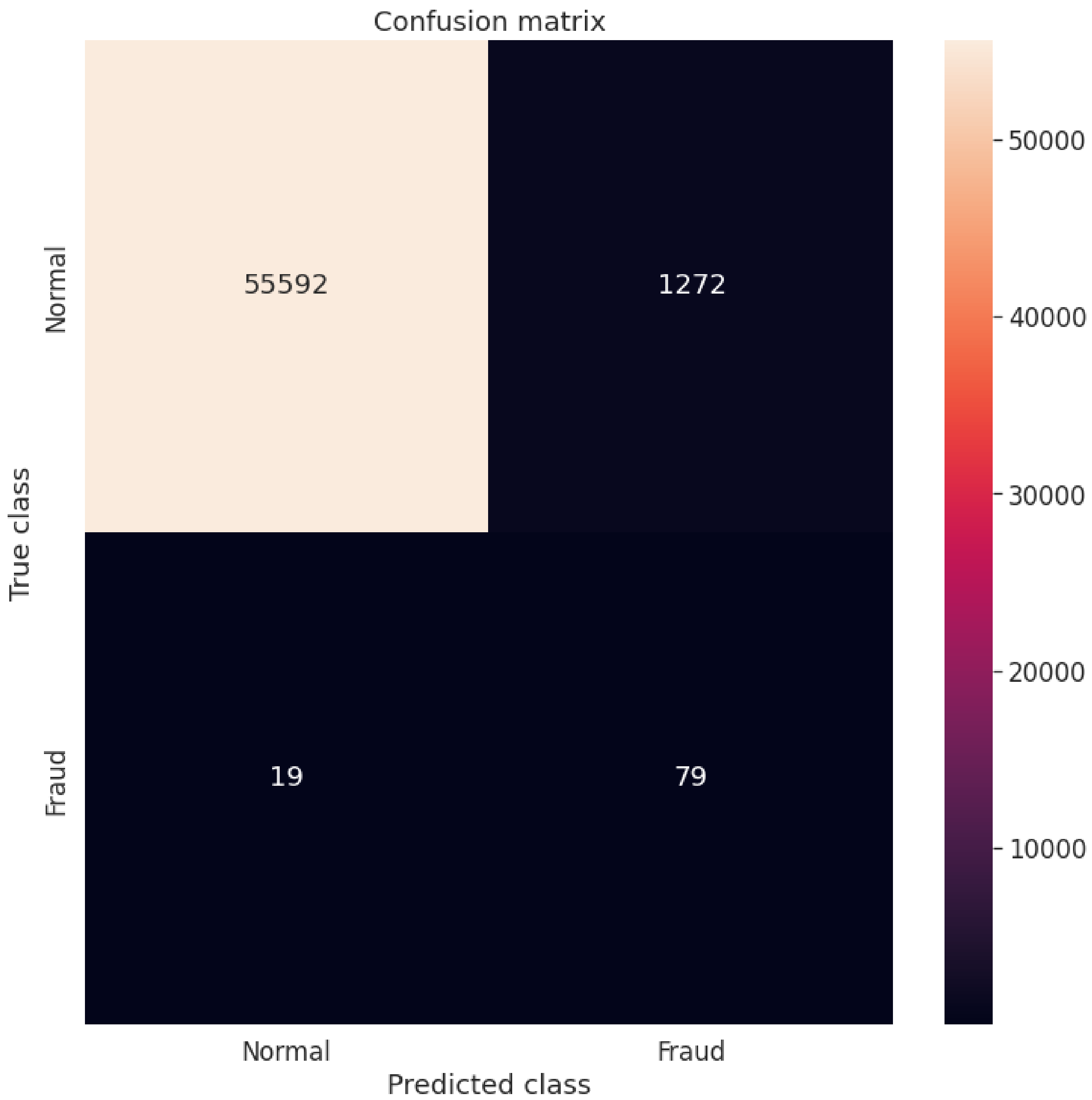



```
"""
---------Prediction------------
Our model is a bit different this time. It doesn't know how to predict new values. But we
 is normal or fraudulent, we'll calculate the reconstruction error from the transaction da
 If the error is larger than a predefined threshold, we'll mark it as a fraud (since our m
 Let's pick that value:"""


threshold = 2.9
```

```
#And see how well we're dividing the two types of transactions:

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, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label='Th
ax.legend()
plt.title("Reconstruction error for different classes")
plt.ylabel("Reconstruction error")
plt.xlabel("Data point index")
plt.show();
```


```
# Let's have a look at the confusion matrix:

y_pred = [1 if e > threshold else 0 for e in error_df.reconstruction_error.values]
conf_matrix = confusion_matrix(error_df.true_class, y_pred)

plt.figure(figsize=(12, 12))
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()
```




```
"""
---------------------
Our model seems to catch a lot of the fraudulent cases. Of course, there is a catch (see w
 The number of normal transactions classified as frauds is really high. Is this really a p
 depending on the problem. That one is up to you.

------Conclusion--------
We've created a very simple Deep Autoencoder in Keras that can reconstruct what non fraudu

Keras gave us very clean and easy to use API to build a non-trivial Deep Autoencoder.
You can search for TensorFlow implementations and see for yourself how much boilerplate yo
```

✓ 0s completed at 9:32 PM