Gogoberidze, Nodari

Fraud Detection using MLPs and Autoencoders

A case study in fraud detection using deep learning methods

Problem Statement
Fraudulent activity is present in just about every industry and economic sector. It comes in many shapes and forms, and varies in impact. In finance and banking, financial fraud is rife as bad actors try to game the system to gain monetary rewards. The detection of financial fraud is of significant interest to every financial entity. For this project, I will be concentrating on detecting fraudulent credit card transactions using deep learning methods. Of particular focus is optimizing model performance on a highly imbalanced dataset.

High Level Overview of Steps
In order to detect fraudulent transactions, I make use of two types of deep learning models. Since this is a two class problem, i.e. fraud vs. non-fraud, I first perform a binary classification on the dataset. I construct two variations of this model. The first is the straightforward, fully-connected model. The second variation places class weights on the training data to account for the data imbalance.
The second type of model makes use of autoencoders. At first it seems counter-intuitive to use an unsupervised, or self-supervised, method to train a network on labeled data. However, it is demonstrated that the resulting model performs relatively well for this type of problem. I run predictions on this model using test data the same in size as the Binary Classification model for a direct comparison of the two model types. I then run a second set of predictions with a much larger number of fraudulent data (that was left unused during training).

Dataset
The “Credit Card Fraud Detection” dataset is available form Kaggle. The data contains credit card transactions made over a two day time period. There are 284,807 recorded transactions, with 492 labeled as fraudulent transactions. This is a highly unbalanced dataset, and is the major challenge to be overcome in this project.

Software: Keras (v. 2.2.4), with a Tensorflow (v. 1.13.1) backend; Python (v. 3.6.7).
Computing Environment: All work was done using Google Colaboratory.

Youtube 2-Minute: https://youtu.be/Ukyud5gpa2o
Youtube 15-Minute: https://youtu.be/h-5t6h384K8

Credit Card Data

The dataset is downloaded from Kaggle, and is made available by the Machine Learning Group - ULB. It is titled “Credit Card Fraud Detection - Anonymized credit card transactions labeled as fraudulent or genuine”. The dataset covers European credit card transactions bade in the month of September, 2013, over a two day period. There are 492 cases of fraud out of a total of 284,807 transactions. This means the dataset is highly unbalanced, the positive class (frauds labeled as 1) account for 0.172% of all transactions. In other words, 99.8% of the data are normal transactions.

This means that even without any training, just assuming all transactions are normal will have a 99.8% accuracy, which is great for any neural network. It’s clear that accuracy is not the metric that should be optimized for. Instead, we care about how accurately the model identifies fraudulent data.

This is what is meant by “unbalanced”. This problem, where the negative case vastly outnumbers the positive case, is very common in “anomaly detection”. Fraud detection is a subtype of anomaly detection, where fraudulent transactions are considered anomalous.

Each transaction in the dataset has 31 columns. Since this is labeled data, there is a “Class” column, where 0 represents a Normal transaction, and 1 represents a Fraudulent one. There is a “Time” column, which represents, for each transaction, the amount of time that has passed since the first transaction in the dataset. While time data is included, I will not be taking any direct advantage of it e.g. LSTM cells. The third labeled column is “Amount”, which represents the amount that was transacted.

The remaining 28 columns are V1-V28. This data has undergone a transformation using Principal Component Analysis (PCA). The effect of this is that the transactions have been anonymized. Where before, there was identifying information, it has been transformed so that confidentiality is ensured. The publishers of the data do not disclose what these columns were previously labeled as, again for privacy reasons.

Getting the Data

The data can be downloaded directly from Kaggle here: https://www.kaggle.com/mlg-ulb/creditcardfraud. However, I make use of the Kaggle API client. To use the Kaggle API, an account must be registered. On the accounts page, a new API token is generated by clicking “Create New API Token”. This downloads a file called “kaggle.json”, which should be placed in “~/.kaggle”. Then the Kaggle API client is downloaded using:

```
pip install kaggle
```

The data is downloaded using:

```
kaggle datasets download -d mlg-ulb/creditcardfraud
unzip creditcardfraud.zip
```

Libraries and Imports

Several tools are used for processing the data. Pandas and Numpy is used for data structures. Seaborn and Matplotlib are used for plotting. Scikit-learn is used for preprocessing and analyzing data. Keras is used to build the models and train the data. Here are all of the imports used:

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
from pylab import rcParams
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score
from sklearn.metrics import classification_report, auc, roc_curve
from sklearn.preprocessing import StandardScaler

from keras.models import Model, load_model
from keras.layers import Input, Dense, Dropout
from keras.callbacks import ModelCheckpoint
from keras import regularizers, Sequential
```

Exploring the Data

The data is imported using pandas, as a dataframe.

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

Inspecting the data, the Time, Amount, Class, and V1 - V28 columns can be seen.

```
df.head(n=3)
```

Time	V1	V2	...	V28	Amount	Class
0	-1.359807	-0.072781	...	-0.021053	149.62	0
0	1.191857	0.266151	...	0.014724	2.69	0
1	-1.358354	-1.340163	...	-0.059752	378.66	0

As mentioned, the data has 284,807 transactions, with 31 columns each.

```
df.shape
```
(284807, 31)

The publishes of the data have ensured that the data is clean. The data has no Null values in it.

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

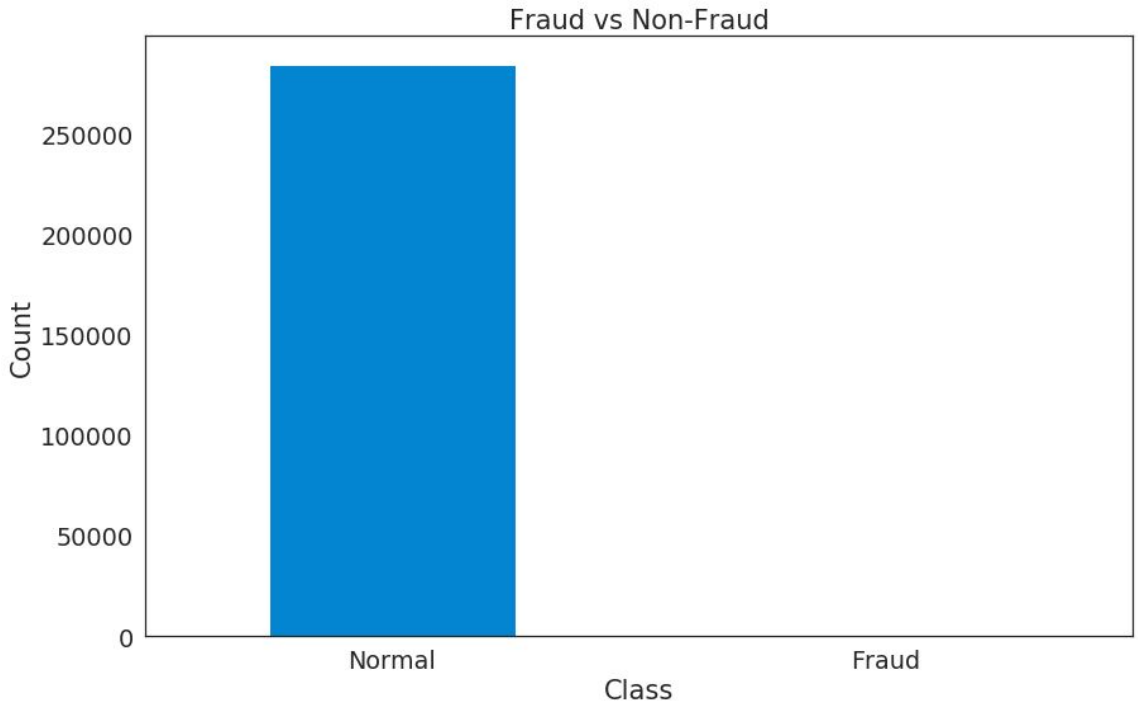
We know how many normal versus fraudulent transactions are in the dataset, but we are able to confirm it.

```
pd.value_counts(df['Class'], sort = True)
```
0 284315
1 492
Name: Class, dtype: int64

From counting the normal transactions versus the fraudulent transactions, 99.8% of the data are normal transactions.

This shows that the dataset is unbalanced. This can be visualized using a histogram.

```
count_classes = pd.value_counts(df['Class'], sort = True)
count_classes.plot(kind = 'bar', rot=0)
plt.xticks(range(2), LABELS)
plt.title("Fraud vs Non-Fraud")
plt.xlabel("Class")
plt.ylabel("Count")
```



Fraud Transactions looks non-existent. It is there, but is dwarfed by Normal Transactions. This is known as an imbalanced class problem. 99.8% is a very good accuracy for any model, so a model that returns "normal" for every transaction would be considered pretty good. Clearly, accuracy is not the metric to strive for in this case.

The metric we want is Accuracy Under the Curve (AUC). The question is the area under which curve. This is discussed later, after several two different curves are shown.

The data has a time column, which represents the time of the given transaction relative to the first transaction. It has a class column (0 for non-fraud, 1 for fraud). It has an amount column, which is the amount of the transaction. It then has 28 other columns, which has undergone Principal Component Analysis (PCA) for privacy concerns. For these 28 columns, there is very low correlation between them. This can be visualized using a correlation matrix heat map.

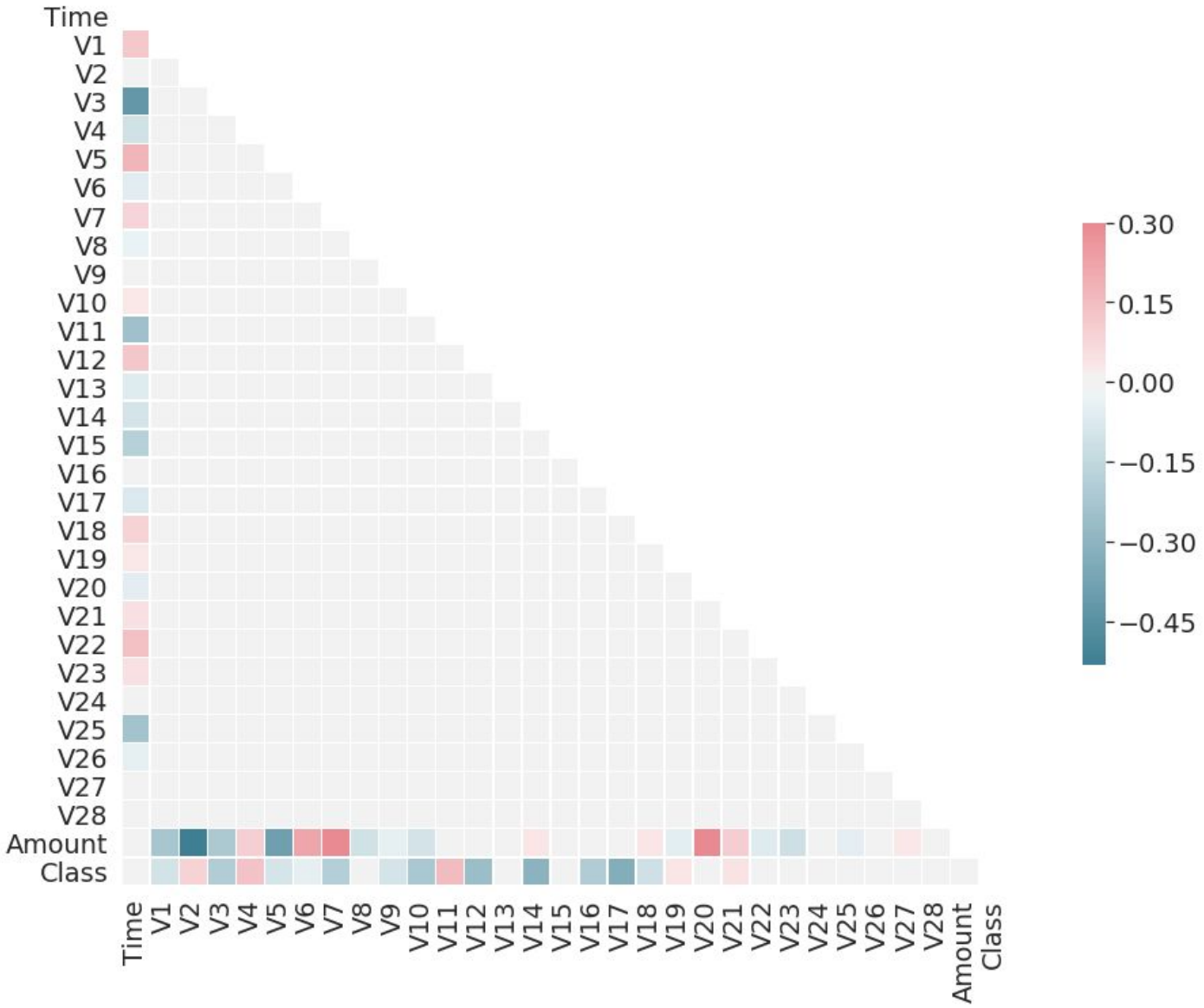
```
corr = df.corr()

mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
f, ax = plt.subplots(figsize=(20, 12))
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
```

```
        square=True, linewidths=.5, cbar_kws={"shrink": .5})
```



It can be seen that there is no correlation between V1-V28. There are correlations to Amount, Class, and Time.

Running the Models

Before anything the data needs to be normalized. The V1-V28 columns are already normalized, but Time and Amount should be as well. We can use scikit-learn to easily do this. We set the Time and Amount columns to have a mean of 0, and a unit standard deviation.

```
df_norm = df
```

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

Binary Classification

The first model built will be a Binary Classifier. This is a standard Feed-Forward Multilayer Perceptron (MLP), using Binary Crossentropy as the loss function. First the data must be split into training and test sets. The split is 80/20, with 80% for training and 20% for testing. Scikit-learn is used to do the split for us.

```
RANDOM_SEED = 314 # used to help randomly select the data points
TEST_PCT = 0.2 # 20% of the data for testing

# split the data 80/20, training to testing
train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT,
random_state=RANDOM_SEED)
```

The input data currently has the class column. We'll seperate that out to use for the labels.

```
train_y = train_x['Class']
train_x = train_x.drop(['Class'], axis=1)

test_y = test_x['Class']
test_x = test_x.drop(['Class'], axis=1)

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

Now for the model. For the hyper parameters, the number of epochs is set to 50, and the batch size to 128.

```
nb_epoch = 50
batch_size = 128
input_dim = train_x.shape[1]  # num of columns, 30
```

The model is a fully connected sequential one. There are 4 dense layers, each with a tanh activation. The first layer is 120 neurons, then 20% dropout. Then a dense layer of 60 neurons followed by another 20% dropout. Then a dense layer of 30 neurons with 20% dropout. Finally, since we want an output between 0 and 1, a dense layer with a single neuron, with a sigmoid activation.

```
model = Sequential()
model.add(Dense(120, input_shape=(input_dim,), activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(60, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(30, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

The model uses Binary Crossentropy for the loss function, and the Adam optimizer.

```
model.compile(metrics=['accuracy'],
              loss='binary_crossentropy',
              optimizer='adam')
```

A checkpoint is defined to save the model.

```
cp = ModelCheckpoint(filepath="mlp_fraud.h5",
                     save_best_only=True,
                     verbose=0)
```

Finally, the model is run.

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

The final epoch will look something like this:

```
Epoch 50/50
227845/227845 [==============================] - 8s 35us/step - loss: 0.0014 - acc:
0.9997 - val_loss: 0.0042 - val_acc: 0.9992
```

The model has a validation accuracy of nearly 100%, which is normally phenomenal. But, as mentioned, the data is highly unbalanced. The accuracy rate is 99.8% just by assuming all of the transactions are classified as "Normal". Instead, the goals is to predict probabilities for inputs, and define a threshold on which we assign those probabilities to their respective class.

By default that threshold may be 0.5. If the probability (the output of the final sigmoid function) is less than half, assign to 0 (normal), if greater, assign to 1 (fraud). But that threshold may be lowered or raised, depending on several criteria.
When making predictions for a binary, or two-class, classification problem, there are two types of errors one can make:

- **Type 1 Error**, or a false positive.
- **Type 2 Error,** or a false negative.

In this case, predicting that a given transaction is fraudulent, when in fact it is not, is a Type 1 error. Predicting a given transaction is normal, when it is really fraudulent, is a Type 2 error.

False Positives (FP) are contrasted with **True Positives (TP)**, which would mean a fraudulent transaction was correctly predicted. **False Negatives (FN)** are contrasted with **True Negative (TN)**, which means a transaction was correctly predicted as non-fraudulent, or normal. In summary:

- TP - correctly identifying a transaction as fraud
- FP - incorrectly identifying a transaction as fraud
- TN - correctly identifying a transaction as normal
- FN - incorrectly identifying a transaction as normal

With this in mind we can use rations of True/False Positive/Negative to see how well the model is performing.

If not already in memory, we can load the model back in.

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

Then we can run our test data in to make predictions.

```
preds_y = model.predict(test_x).flatten()
```

It's useful to have the labels and the predictions in a single labeled data structure. We can use a pandas dataframe for this, and inspect the data.

```
error_df = pd.DataFrame({'Predictions': preds_y,
                         'Actual': test_y})
error_df.describe()
```

	Predictions	Actual
count	56962	56962

mean	0.001824	0.002019
std	0.039109	0.044887
min	0.000028	0
25%	0.000028	0
50%	0.000029	0
75%	0.000031	0
max	0.999776	1

It's not terribly informative. We can see most of our predictions hover close to 0. As always, this is because the data is highly imbalanced.

As mentioned, overall model accuracy is not a good metric. We can use Type 1 and Type 2 errors as metrics, but how exactly? There are typically two curves used.

ROC Curve

In order to see how well the model is doing with regards to Type 1 and Type 2 errors, we cans see how well the predictions are made against some common ratios:

- The **True Positive Rate**, or **Sensitivity** is *(TP)/(TP + FN).*
- The **False Positive Rate** is *(FP)/(FP + TN)*.
    - **Specificity** is *(TN)/(TN + FP)* or *(1 - False Positive Rate)*.

To graph the results of the model's prediction with respect to these ratios, the **Receiver Operating Characteristic (ROC)** curve is used. The ROC curve shows the tradeoff between true positive vs false positive rates, for different threshold values.

We can obtain the False Positive Rate, the True Positive Rate, and the Thresholds using scikit-learn.

```
fpr, tpr, thresholds = roc_curve(error_df.Actual, error_df.Predictions)
```

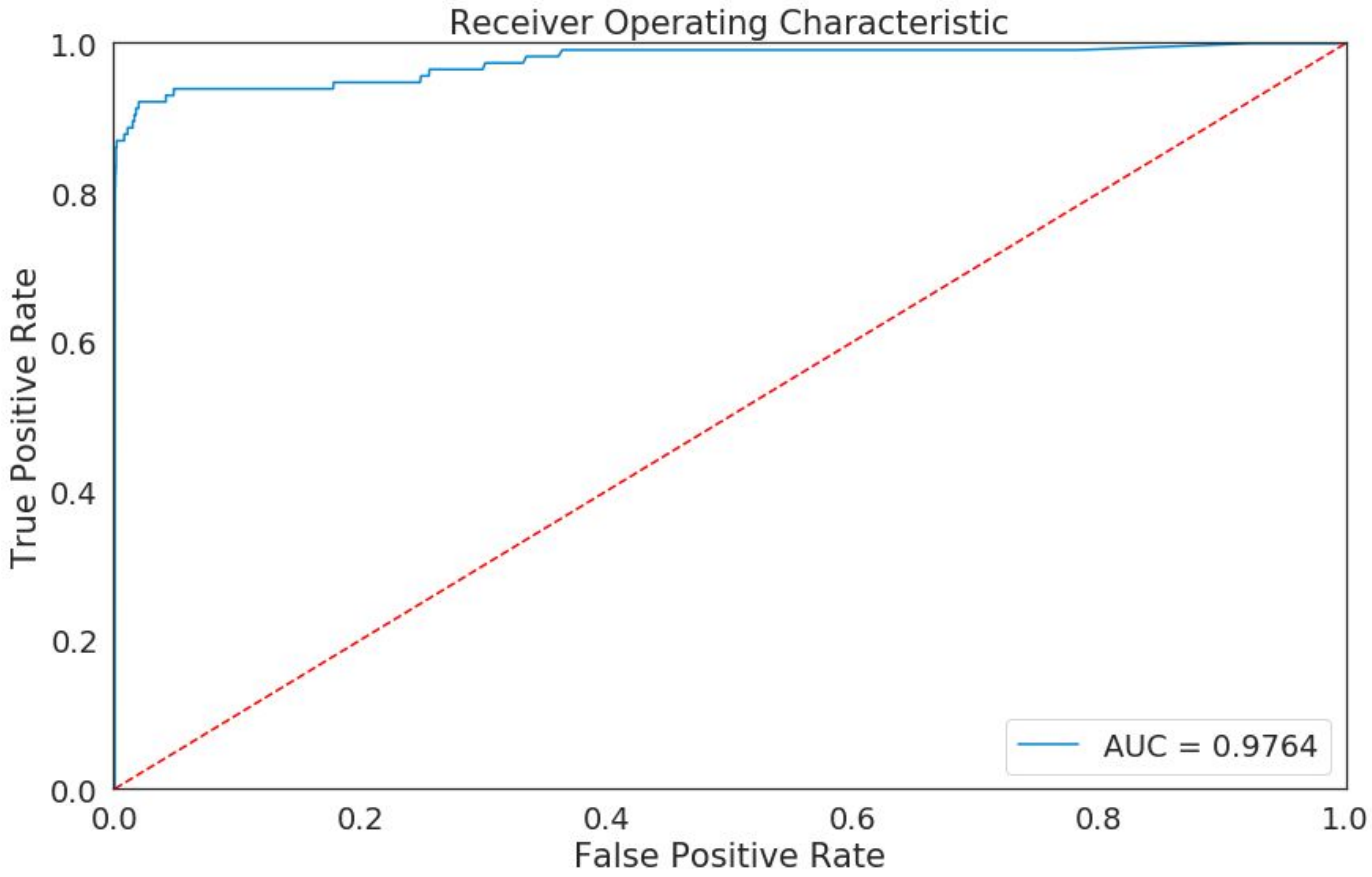
We will plot the curve using these returned values. We can also calculate the Area Under the Curve (AUC).

```
roc_auc = auc(fpr, tpr)
```

We can use Matplotlib to plot the resulting curve.

```
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();
```



Normally, this is the ideal shape we want. Skillful models will have the AUC very high, and perfect models will have it at 100%. We have an AUC of 97.64%, which is very good. However, in for the given dataset the shape of this curve is unsurprising. This very optimistic looking curve is a result of the large data imbalance, and doesn't help us much.

The reason that this curve is unhelpful is that the very large number of class 0 (normal) transactions means we are less interested in the skill of the model at predicting class 0 correctly, e.g. high True Negatives.

Precision-Recall Curve

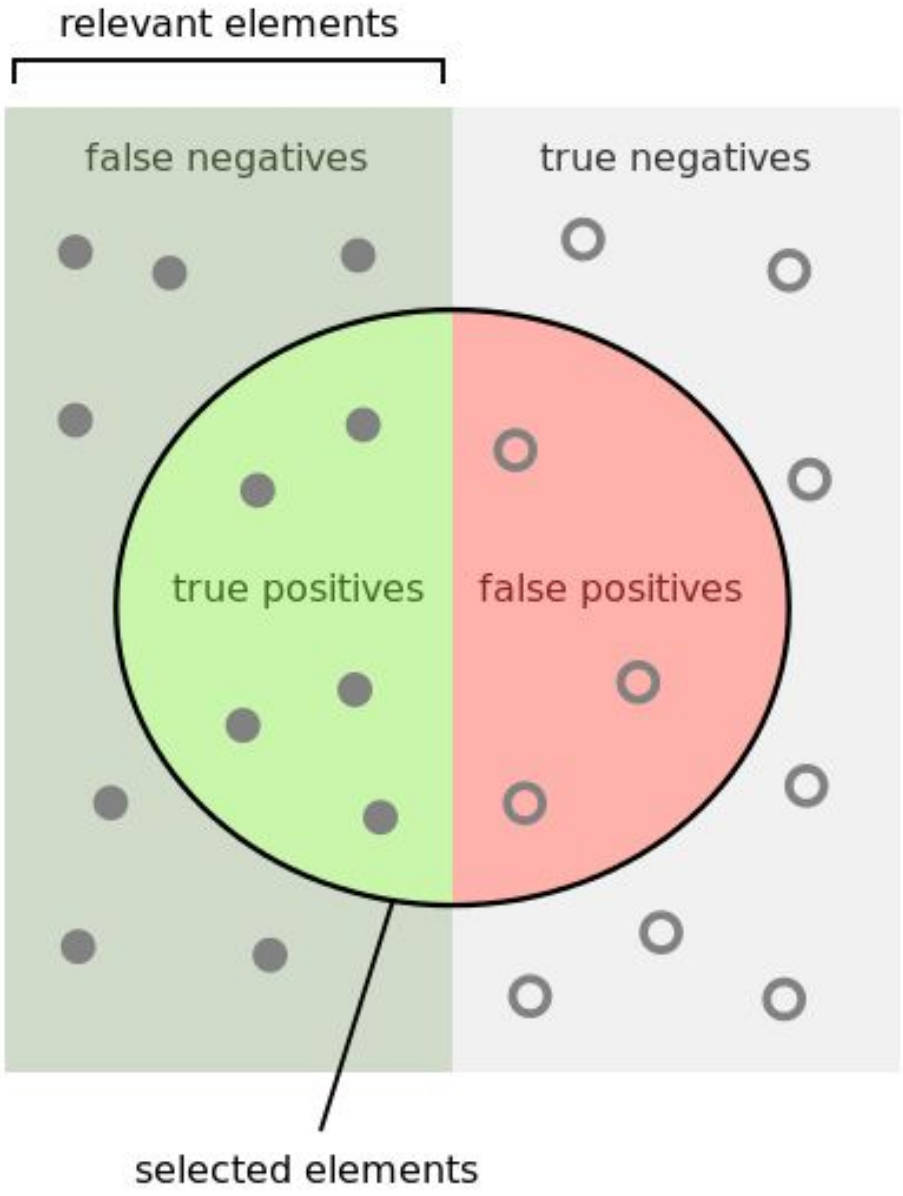
The second common type of curve used to understand the error rate is the **Precision-Recall Curve**. While mapping a curve based on True Positive Rate vs False Positive Rate is unhelpful,

there is a combination of rations that can be used which are informative in the case of highly imbalanced data:

- **Precision** = *(TP)/(TP + FP)*
- **Recall** = *(TP)/(TP + FN)*

Precision is also known as **Positive Predictive Power**. Recall is the same as sensitivity, i.e. the True Positive Rate.

Wikipedia provides a helpful visual.



How many selected items are relevant?

$$\text{Precision} = \frac{\text{true positives}}{\text{selected elements}}$$

How many relevant items are selected?

$$\text{Recall} = \frac{\text{true positives}}{\text{relevant elements}}$$

Note that the visual is not “to scale” in regards to our dataset. The visual is more appropriate for a perfectly balanced dataset. For the credit card dataset, the left side of the figure is much smaller than the right side.

The balance between Precision and Recall is useful for imbalanced datasets. Since neither Precision nor Recall uses True Negatives (the majority of the dataset), the combination of the two becomes very informative.

For detecting fraudulent events, we want Recall to be as close to 100% as possible. However, there is always a trade off between Recall and Precision (though not necessarily 1-to-1). Higher Recall (identifying more fraudulent events), means lower Precision (incorrectly flagging normal events as fraudulent). The threshold mentioned above, defines where we make this trade off.

To visualize these metrics, the Precision-Recall Curve is used. This curve summarizes the trade-offs between Precision and Recall, for different thresholds.

The **No-Skill Line** on a Precision-Recall curve is the number of positive cases (fraud) divided by the number of total cases (positive and negative). For a perfectly balanced dataset (equal number of positive and negative), this is 0.5. A skillful model will have a curve that rests above this point.

```
num_fraud = error_df.Actual[error_df.Actual == 1].count()
num_normal = error_df.Actual[error_df.Actual == 0].count()
no_skill_level = num_fraud/(num_fraud + num_normal)

print(num_fraud, num_normal, no_skill_level)
(115, 56847, 0.0020188897861732383)
```

We see the No-Skill Level is very low for this dataset.

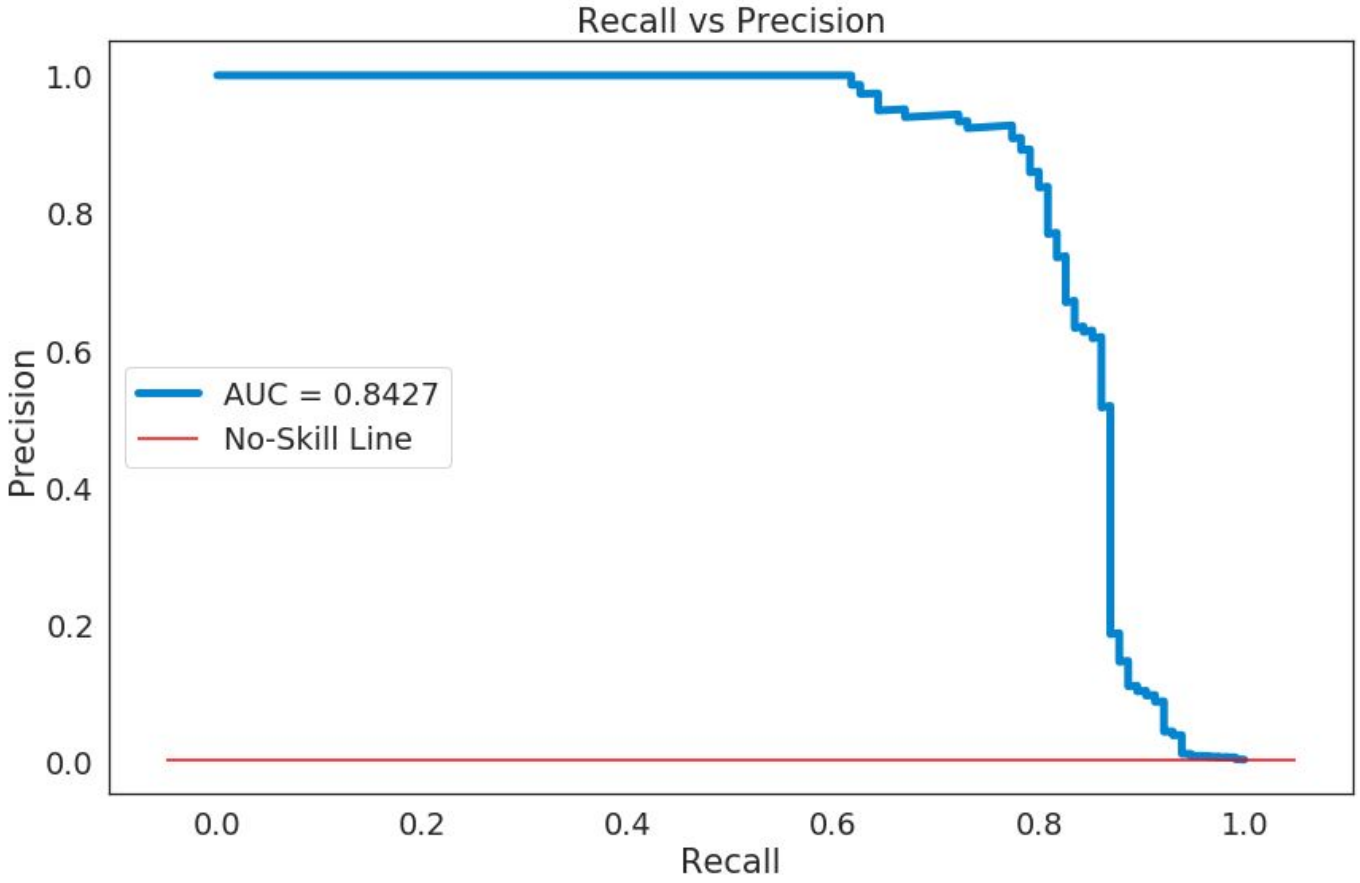
To get the Precision, Recall, and different Thresholds we use scikit-learn. We will also get the AUC.

```
precision_rt, recall_rt, threshold_rt = precision_recall_curve(error_df.Actual,
error_df.Predictions)
pr_auc = auc(recall_rt, precision_rt)
```

We can plot the resulting curve using Matplotlib, making sure to include the AUC and the No-Skill Line.

```
fig, ax = plt.subplots()
ax.plot(recall_rt, precision_rt, linewidth=5, label='AUC = %0.4f'% pr_auc)
```

```
ax.hlines(no_skill_level, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label='No-Skill Line')
ax.legend(loc='center left')
plt.title('Recall vs Precision')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()
```



The curve sits comfortably over the no-skill line for most threshold levels. Here AUC is important. The closer it is to 1 (100%), the better the model is.

Again, the goal is to have a high Recall so that as many fraudulent events are captured as possible. However the curve shows the trade off that is made: for higher values of Recall, the Precision goes down. The only way to have a near perfect Recall is to have near zero Precision, which is the same as claiming that all transactions in the dataset are fraudulent. You'll capture all the fraud events, but at the expense of incorrectly identifying all of the normal events as fraudulent.

Finally we have good metrics to test our model against. We want as high Recall as possible while maintaining relatively high Precision. Therefore we want the Precision-Recall AUC to be

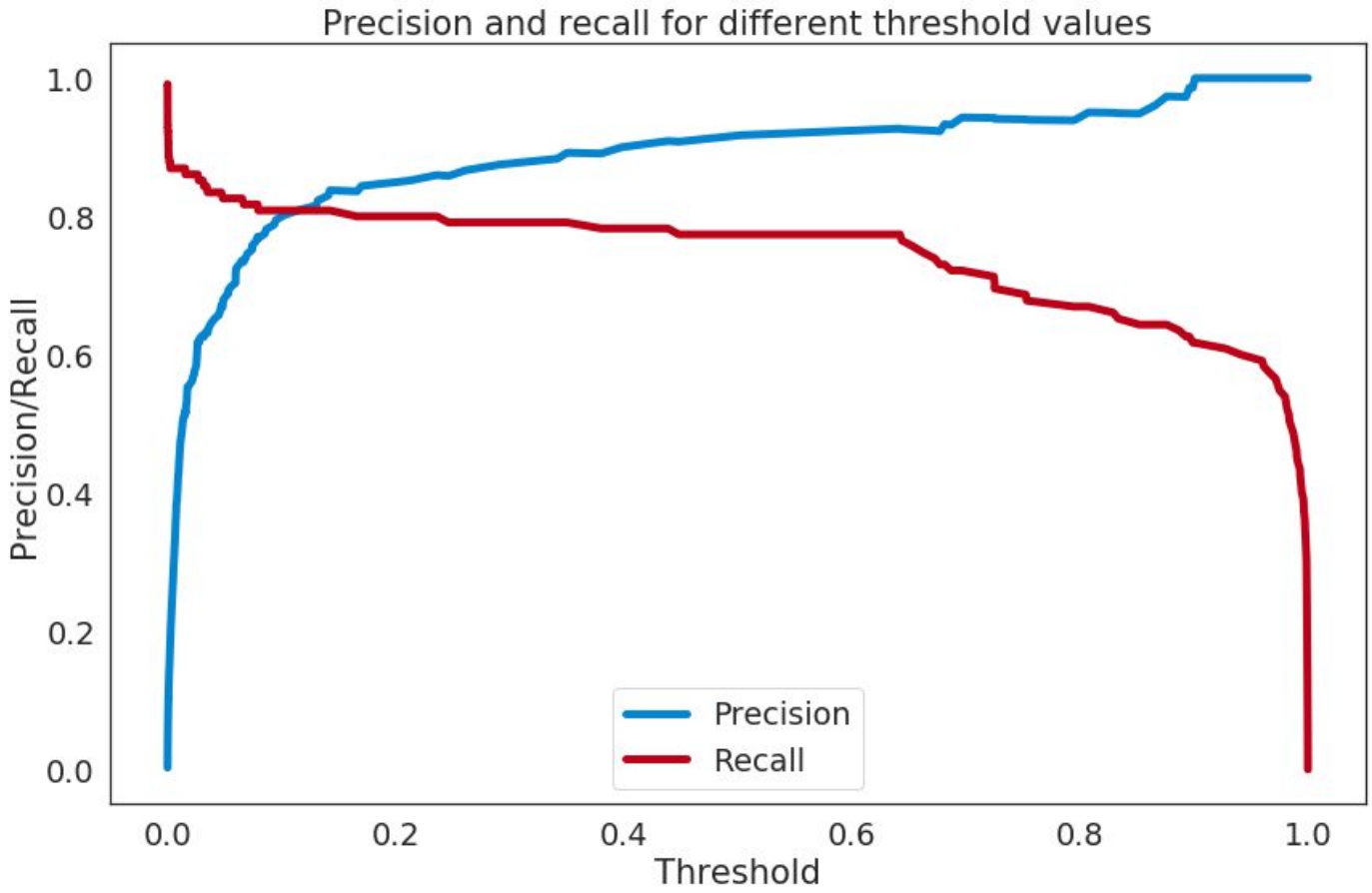
as high as possible. The next step is to define where we make the tradeoff between Recall and Precision by setting a threshold.

Defining a Threshold

In order to find good threshold values, we can plot Precision and Recall directly against different threshold values.

Threshold values were returned by precision_recall_curve() function, so we can plot the precision values and the recall values against those.

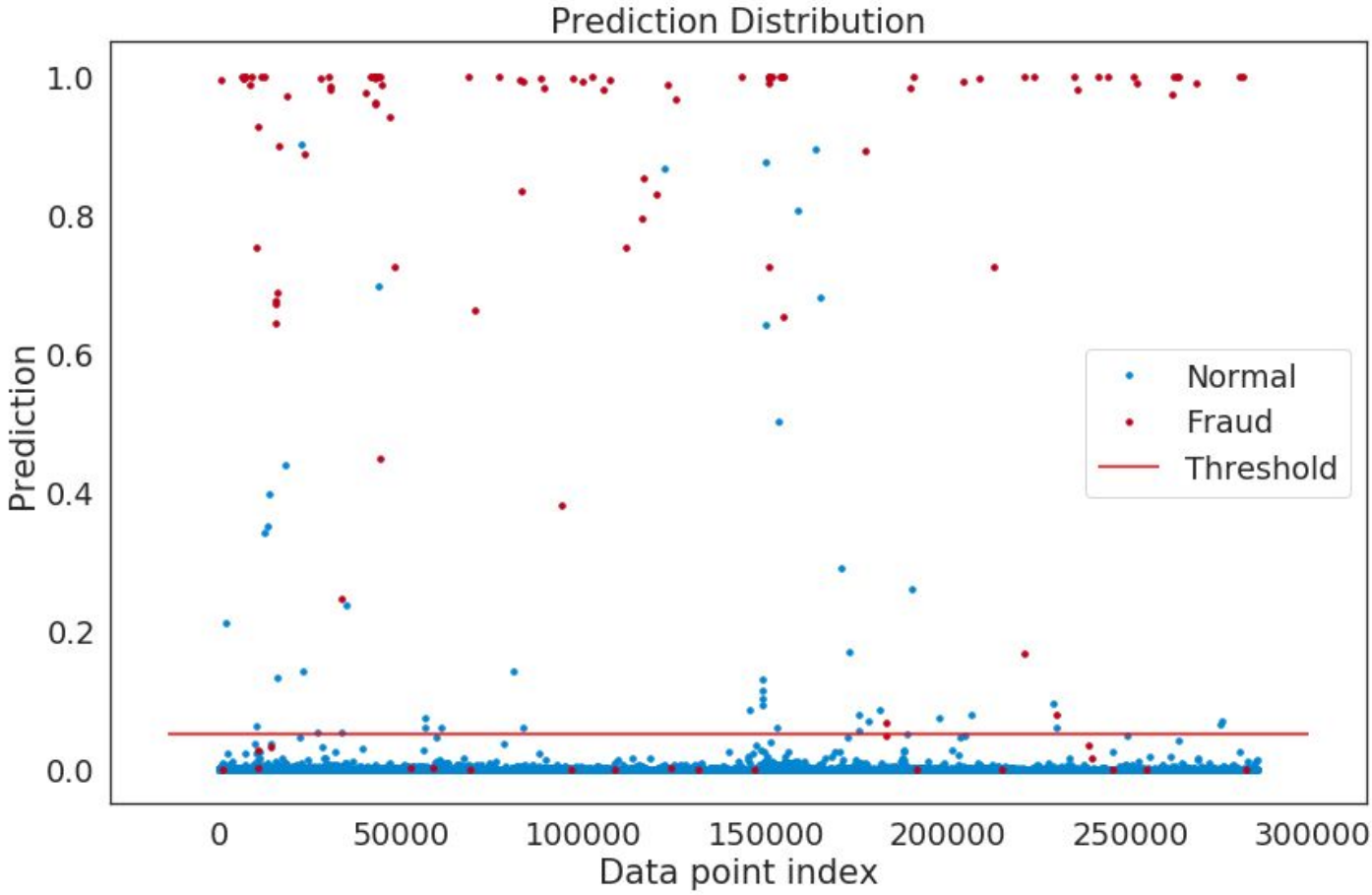
```
plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=5)
plt.plot(threshold_rt, recall_rt[1:], label="Recall",linewidth=5)
plt.title('Precision and recall for different threshold values')
plt.xlabel('Threshold')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()
```



For both high Precision and high Recall, a low threshold is needed. 0.05 seems to be a good threshold. We can plot the distribution of the predictions, the probabilities the model assigned, the threshold level, and contrast actual classes (normal vs fraud).

```
threshold_fixed = 0.05
groups = error_df.groupby('Actual')
fig, ax = plt.subplots()

for name, group in groups:
    ax.plot(group.index, group.Predictions, 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("Prediction Distribution")
plt.ylabel("Prediction")
plt.xlabel("Data point index")
plt.show();
```



All of the Normal points (in blue) above the threshold (red line) are false positives. All of the Fraud points (in read) above the line are true positives. We can see that if we lower the threshold much more, to capture the False Negatives, we risk far too many false positives. We can also see that raising the threshold to reduce the False Positives will dramatically reduce the number of correctly identified Fraud transactions.

Final Tally

So how well is the model performing?

First we convert our predictions into binary class values based on the threshold we set.

```
preds_y_binary = [1 if e > threshold_fixed else 0 for e in error_df.Predictions]
```

Precision and Recall are easy to calculate for any given threshold. We can use a nice convenience function from scikit-learn.

```
print(classification_report(test_y, preds_y_binary))
```

	precision	recall	f1-score	support
0 (normal)	1	1	1	56847
1 (fraud)	0.68	0.83	0.75	115
micro avg	1	1	1	56962
macro avg	0.84	0.91	0.87	56962
weighted avg	1	1	1	56962

We can see that we have Precision of 68% and Recall of 83%.

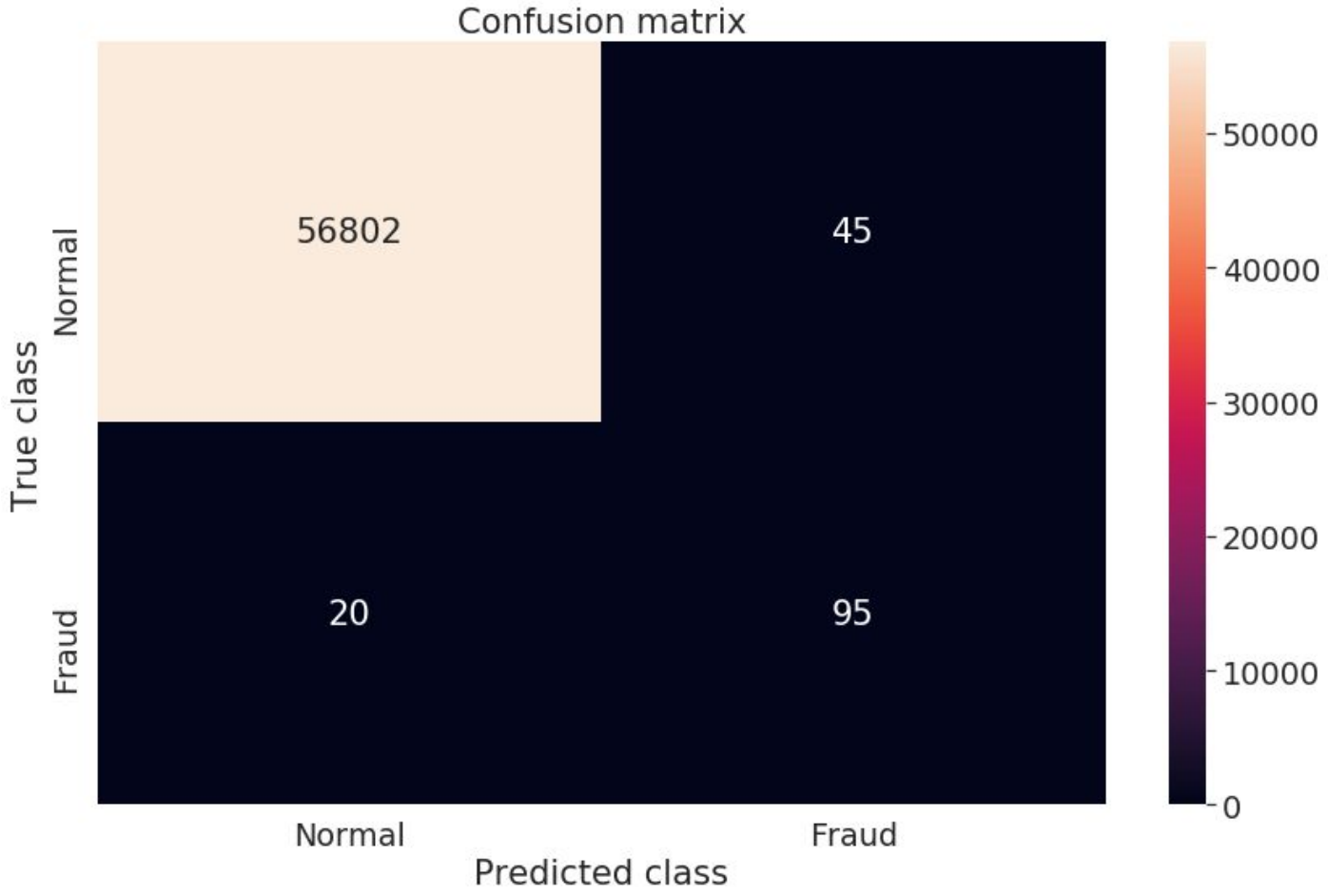
F1 is an overall measure of a model's accuracy that combines precision and recall *(Precision X Recall) / (Precision + Recall)*. That means a good F1 score means low False Positives and low False Negatives. The F1 score is 75%, which is decently good.

Finally we can plot a Confusion Matrix, which shows the exact numerical values for all combinations of True/False-Positive/Negative.

```
conf_matrix = confusion_matrix(test_y, preds_y_binary)
```

```
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()
```



95 fraud cases are correctly identified, 20 are missed, 45 are mis-identified as fraud, and 56,802 are correctly categorized as normal. In summary:

- TP - 95
- FN - 20
- FP - 45
- TN - 56,802

Not bad.

Autoencoder Model

The point of using autoencoders is to use the input data as its own label, and come up with some useful intermediate representation (encoding). Then when running a prediction, and

feeding in input data that does not fit well to the intermediate representation, then output signal that it does not fit well. This is known as **Reconstruction Error**.

For this reason we drop the fraudulent data from the training, in order to come up with an encoding for "normal" (non-fraudulent) transactions. After the model is trained, the hope is that feeding in fraudulent data will easily be flagged due to its high Reconstruction Error.

First, create the training and testing data, similar to before but with one big difference.

```
# split the data 80/20, training to testing
train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT,
random_state=RANDOM_SEED)
```

The difference this time, is we're going to take out all of the Fraud data from the training set. As stated, the goal is to come up with a good encoding for *normal* transaction data, and then test the reconstruction error against that.

```
# take out the fraudulent transactions from training data
remaining_fraud = train_x[train_x.Class == 1]
# keep only normal transaction data in training set
train_x = train_x[train_x.Class == 0]

# drop the class column (labels) from training set
train_x = train_x.drop(['Class'], axis=1)

# create labels for test data
test_y = test_x['Class']
# drop the labels from test data itself
test_x = test_x.drop(['Class'], axis=1)

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

The test data has the same counts for normal and fraud transactions as before. This way we can run directo comparisons against the previous model. However since we've set aside 80% of the fraud data from the training set, it would be a shame to let it go to waste. After making comparisons, we'll run a second round of predictions with *all* of the fraud data. For now we'll create some variables to hold the extended testing set, with all fraud data.

```
# take out labels from remaining fraud data (taken out from training set)
remaining_fraud_y = remaining_fraud['Class']
remaining_fraud_x = remaining_fraud.drop(['Class'], axis=1).values
```

```
# extend test data and label with additional fraud data
test_x_extended = np.concatenate((remaining_fraud_x, test_x), axis=0)
test_y_extended = pd.concat([remaining_fraud_y, test_y])
```

Setup and Train the Autoencoder

We'll train this model for a couple more epochs than the previous, setting it at 100. The batch size is the same. We'll go down to 14 neurons, and then 7.

```
nb_epoch = 100
batch_size = 128
input_dim = train_x.shape[1] # num of columns, 30
encoding_dim = 14
hidden_dim = int(encoding_dim / 2) # i.e. 7
learning_rate = 1e-7
```

The encoder will use an L1 regularizer. The decoder mirrors the encoder. So the model will be 30 -> 14 -> 7 -> 7 -> 14 -> 30.

```
input_layer = Input(shape=(input_dim, ))
encoder = Dense(encoding_dim, activation="tanh",
activity_regularizer=regularizers.l1(learning_rate))(input_layer)
encoder = Dense(hidden_dim, activation="relu")(encoder)
decoder = Dense(hidden_dim, activation='tanh')(encoder)
decoder = Dense(input_dim, activation='relu')(decoder)
autoencoder = Model(inputs=input_layer, outputs=decoder)
```

I did try different parameters, but they had no significant impact on performance, so this simple model was as performant but not needlessly complex.

This time we will use Mean Squared Error as the loss function, and stick with the Adam optimizer.

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

Create a checkpoint to save the model.

```
cp = ModelCheckpoint(filepath="autoencoder_fraud.h5",
                               save_best_only=True,
                               verbose=0)
```

Run the model.

```
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]).history
```

The final epoch looks something like this:
Epoch 100/100
227468/227468 [==============================] - 6s 27us/step - loss: 0.6901 - acc: 0.7192 - val_loss: 0.7244 - val_acc: 0.7201

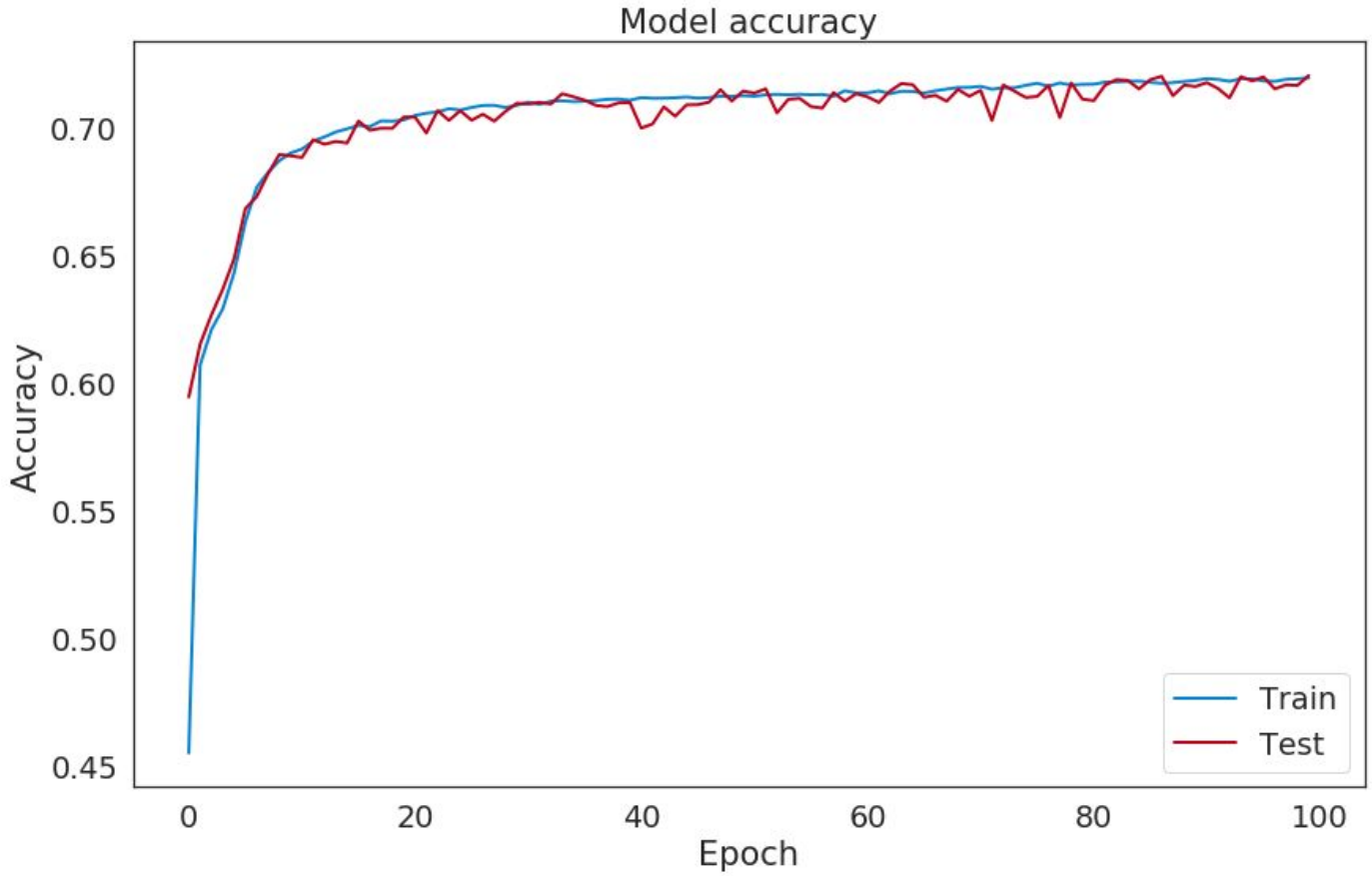
Autoencoder Results

We can load the model back in if not saved in memory.

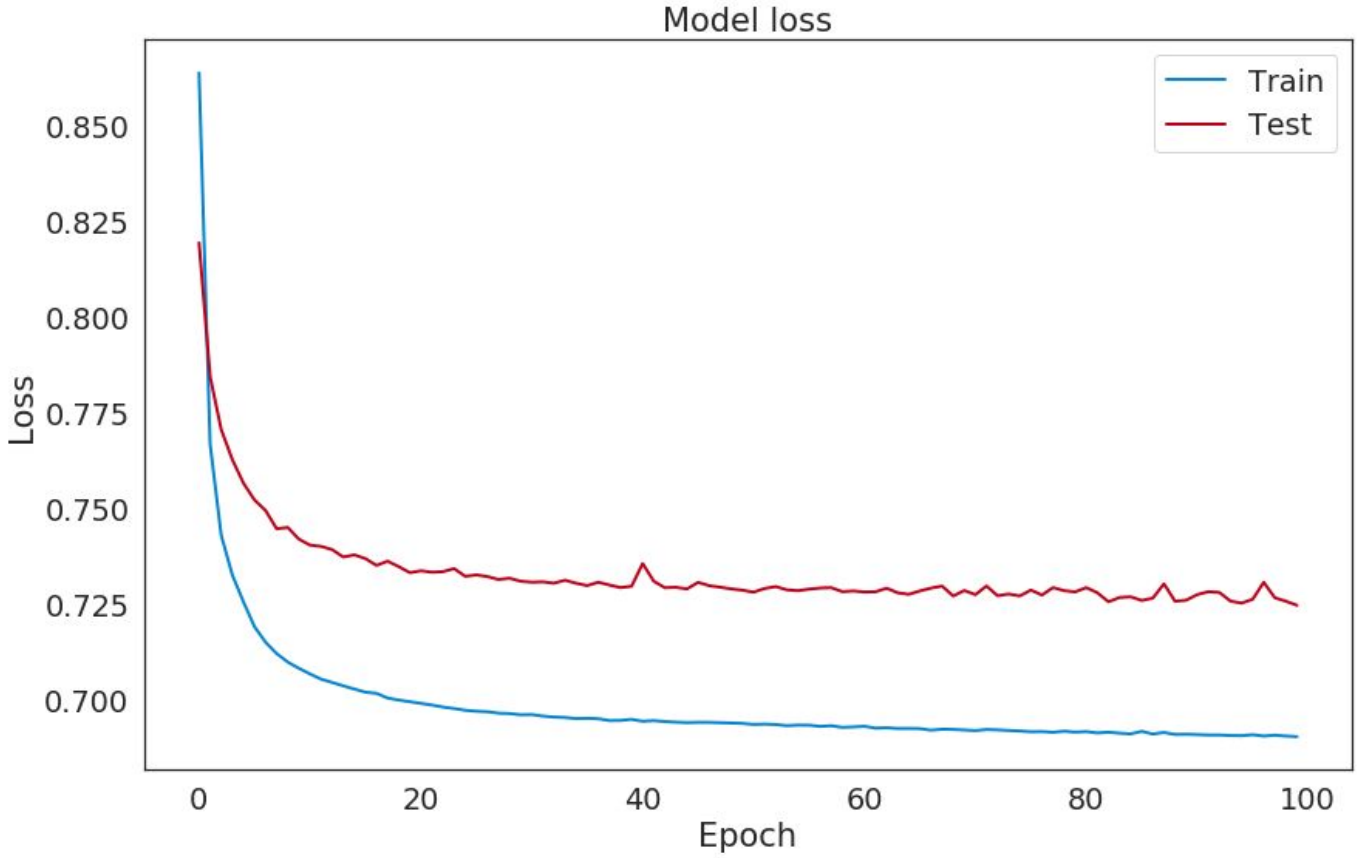
```
autoencoder = load_model('autoencoder_fraud.h5')
```

First we can look at the history. We can see the accuracy of the training and testing data are very closely aligned, and flatten out near 70%.

```
plt.plot(history['acc'], linewidth=2, label='Train')
plt.plot(history['val_acc'], linewidth=2, label='Test')
plt.legend(loc='lower right')
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```


Model accuracy
Accuracy
0.70
0.65
0.60
0.55
0.50
0.45
0
20
40
60
80
100
Epoch
Train
Test

Mean squared error is near 70% for the training and 72% for the test data.



Reconstruction Error

Running the predictions on the test data, we can get the reconstruction error.

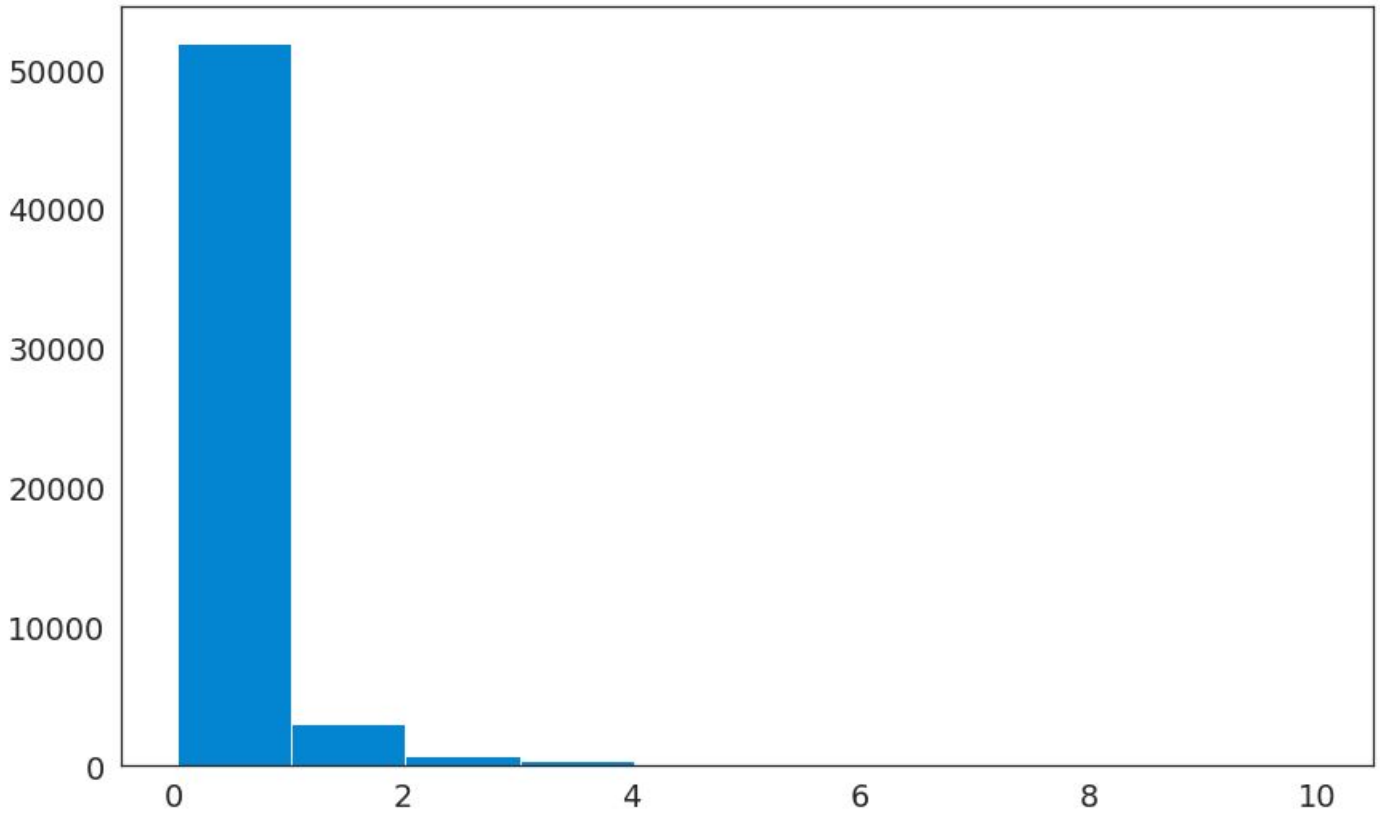
```
preds = autoencoder.predict(test_x)
mse = np.mean(np.power(test_x - preds, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse,
                         'Actual': test_y})
error_df.describe()
```

	Reconstruction_error	Actual
count	56962	56962
mean	0.724297	0.002019
std	3.140024	0.044887
min	0.040535	0
25%	0.246993	0
50%	0.386249	0

75%	0.61612	0
max	199.174374	1

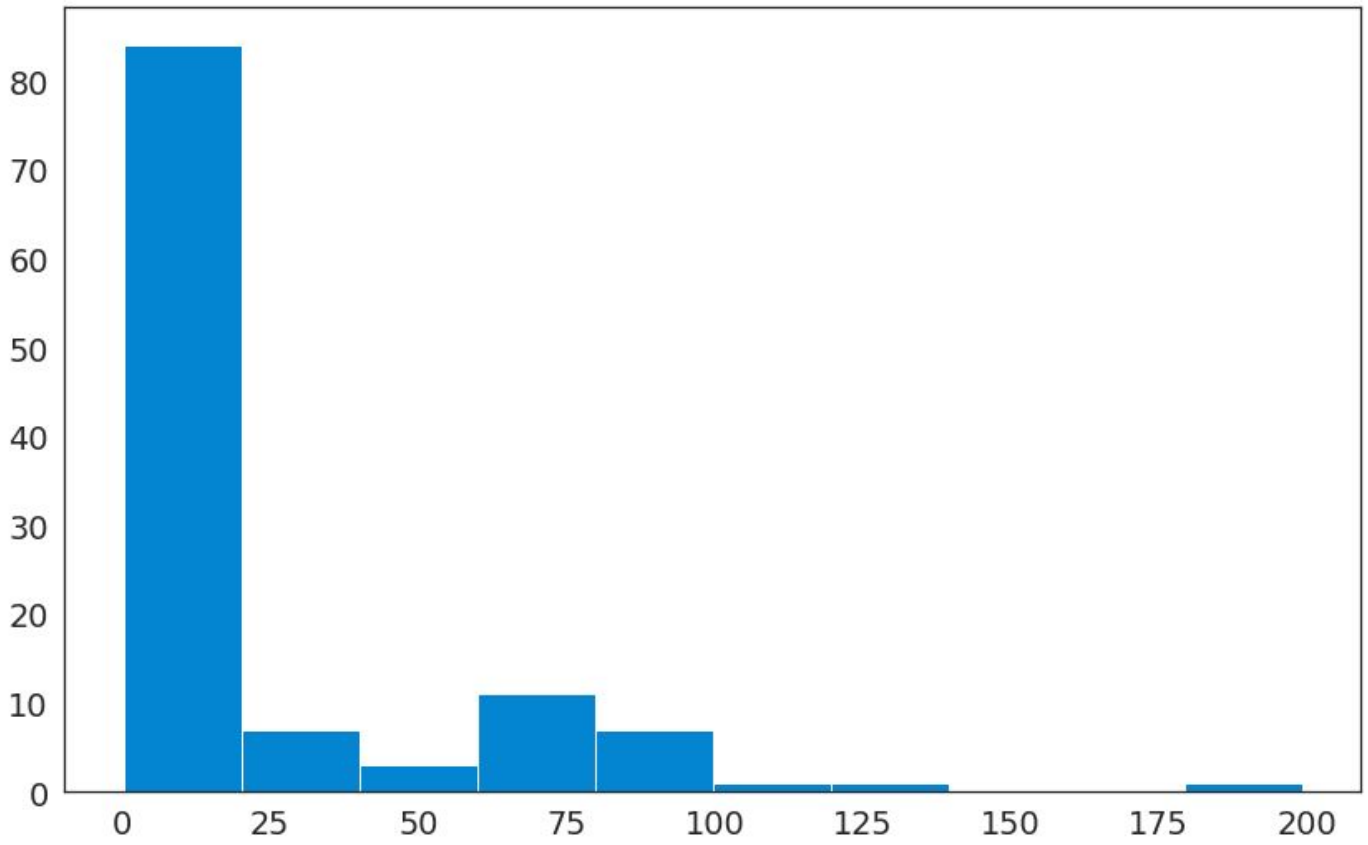
Perhaps some histograms would be more informative. For reconstruction error of normal transactions (without fraud), we can see most of the predictions are near zero.

```
fig = plt.figure()
ax = fig.add_subplot(111)
normal_error_df = error_df[(error_df['Actual']== 0) & (error_df['Reconstruction_error'] < 10)]
_ = ax.hist(normal_error_df.Reconstruction_error.values, bins=10)
```



On the other hand, for reconstruction error of fraud transactions, we can see the predictions are far more spread out. In simple terms, high reconstruction error values tend to mean they are fraud events.

```
fig = plt.figure()
ax = fig.add_subplot(111)
fraud_error_df = error_df[error_df['Actual'] == 1]
_ = ax.hist(fraud_error_df.Reconstruction_error.values, bins=10)
```



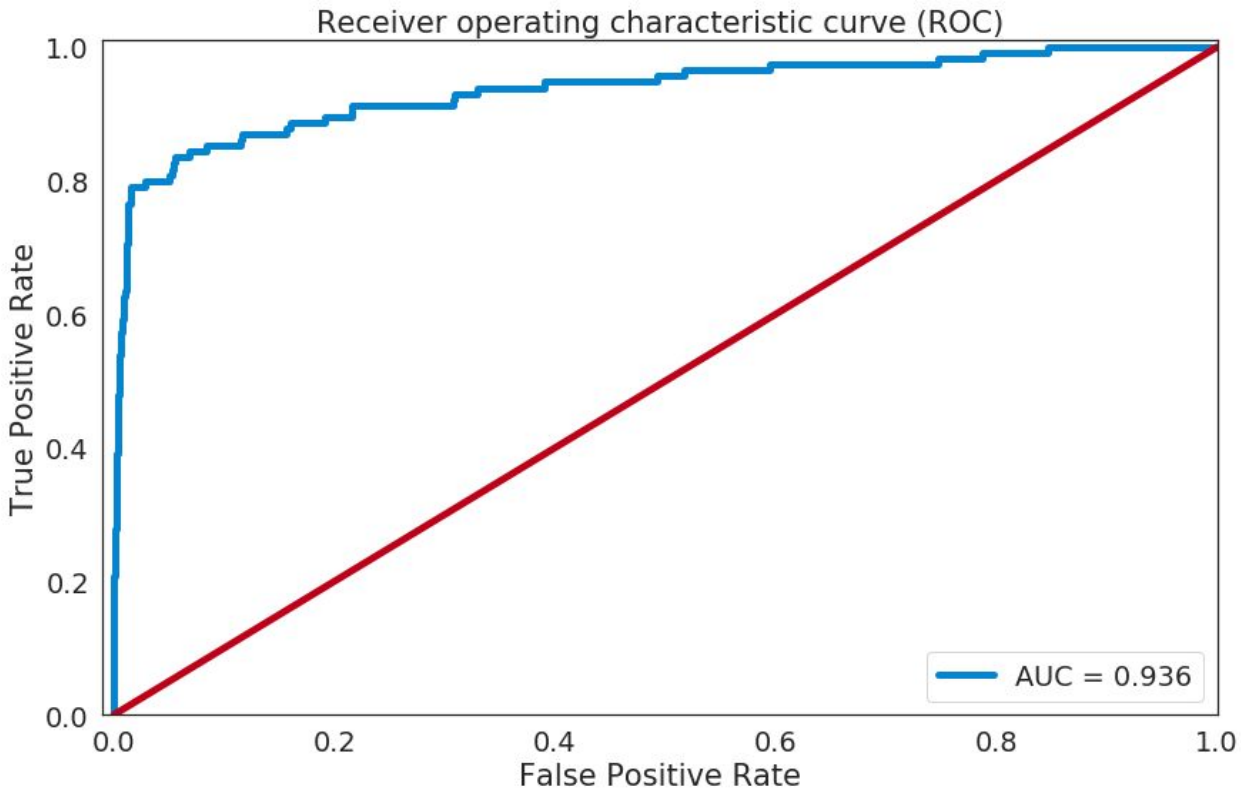
ROC Curve

Although we know that the ROC Curve is not informative for the dataset, we can still take a look. Plotting the ROC curve we see again that it looks very optimistic. But it's the same dataset as before, which is highly unbalanced, so it is not very useful.

```
false_pos_rate, true_pos_rate, thresholds = roc_curve(error_df.Actual,
error_df.Reconstruction_error)
roc_auc = auc(false_pos_rate, true_pos_rate,)

plt.plot(false_pos_rate, true_pos_rate, linewidth=5, label='AUC = %0.3f'% roc_auc)
plt.plot([0,1],[0,1], linewidth=5)

plt.xlim([-0.01, 1])
plt.ylim([0, 1.01])
plt.legend(loc='lower right')
plt.title('Receiver operating characteristic curve (ROC)')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Precision - Recall

What we really care about is Precision and Recall. We will again plot the Precision-Recall curve. First we find the skill level, which is the same as before, since it's the same amount of data as before.

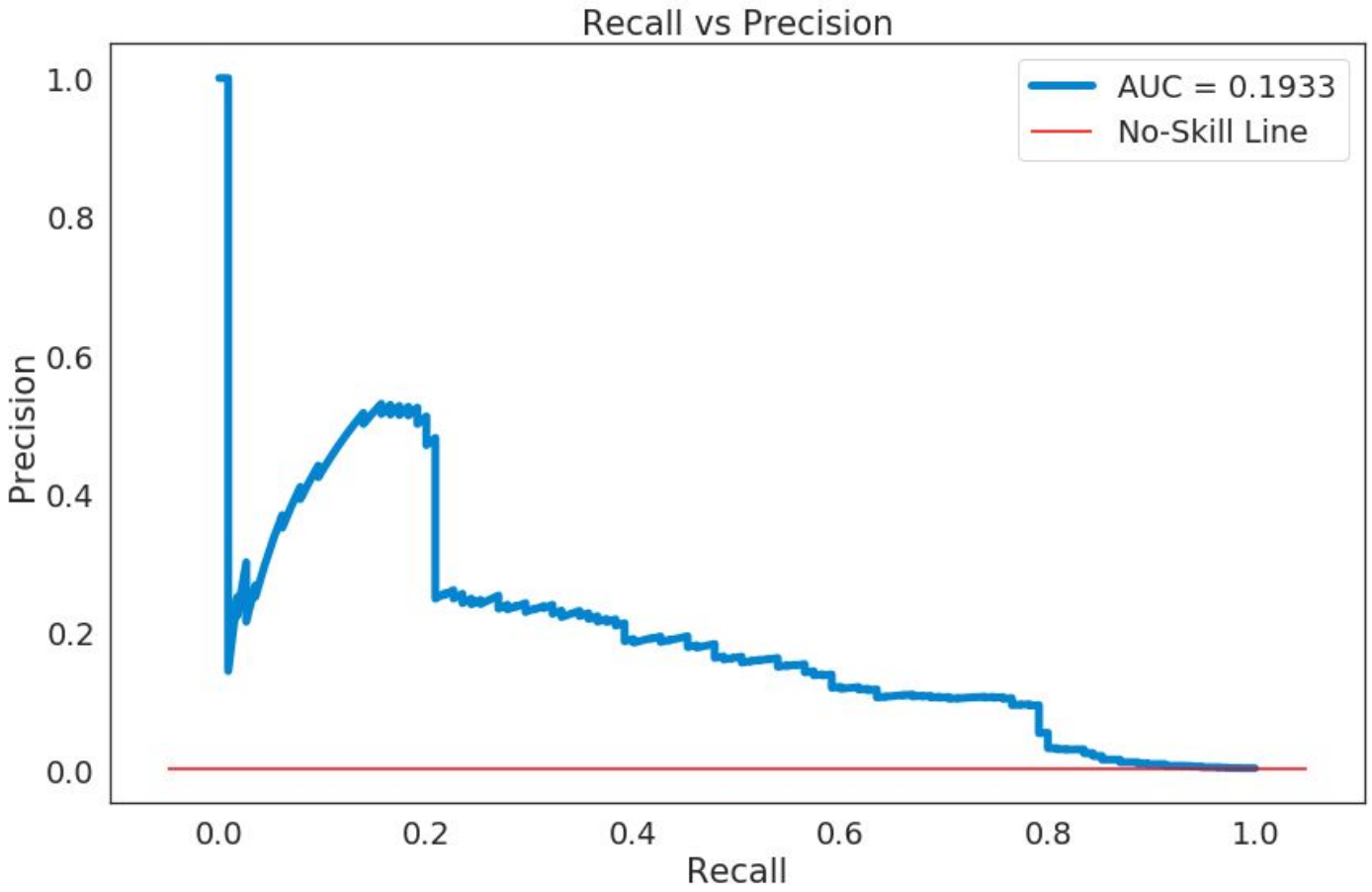
```
num_fraud = error_df.Actual[error_df.Actual == 1].count()
num_normal = error_df.Actual[error_df.Actual == 0].count()
no_skill_level = num_fraud/(num_fraud + num_normal)
```

```
print(num_fraud, num_normal, no_skill_level)
(115, 56847, 0.0020188897861732383)
```

We calculate the precision, recall, and threshold values, as well as the AUC. Then we plot them.

```
precision_rt, recall_rt, threshold_rt = precision_recall_curve(error_df.Actual,
error_df.Reconstruction_error)
pr_auc = auc(recall_rt, precision_rt)
fig, ax = plt.subplots()
ax.plot(recall_rt, precision_rt, linewidth=5, label='AUC = %0.4f'% pr_auc)
ax.hlines(no_skill_level, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label='No-Skill
Line')
```

```
ax.legend(loc='upper right')
plt.title('Recall vs Precision')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()
```



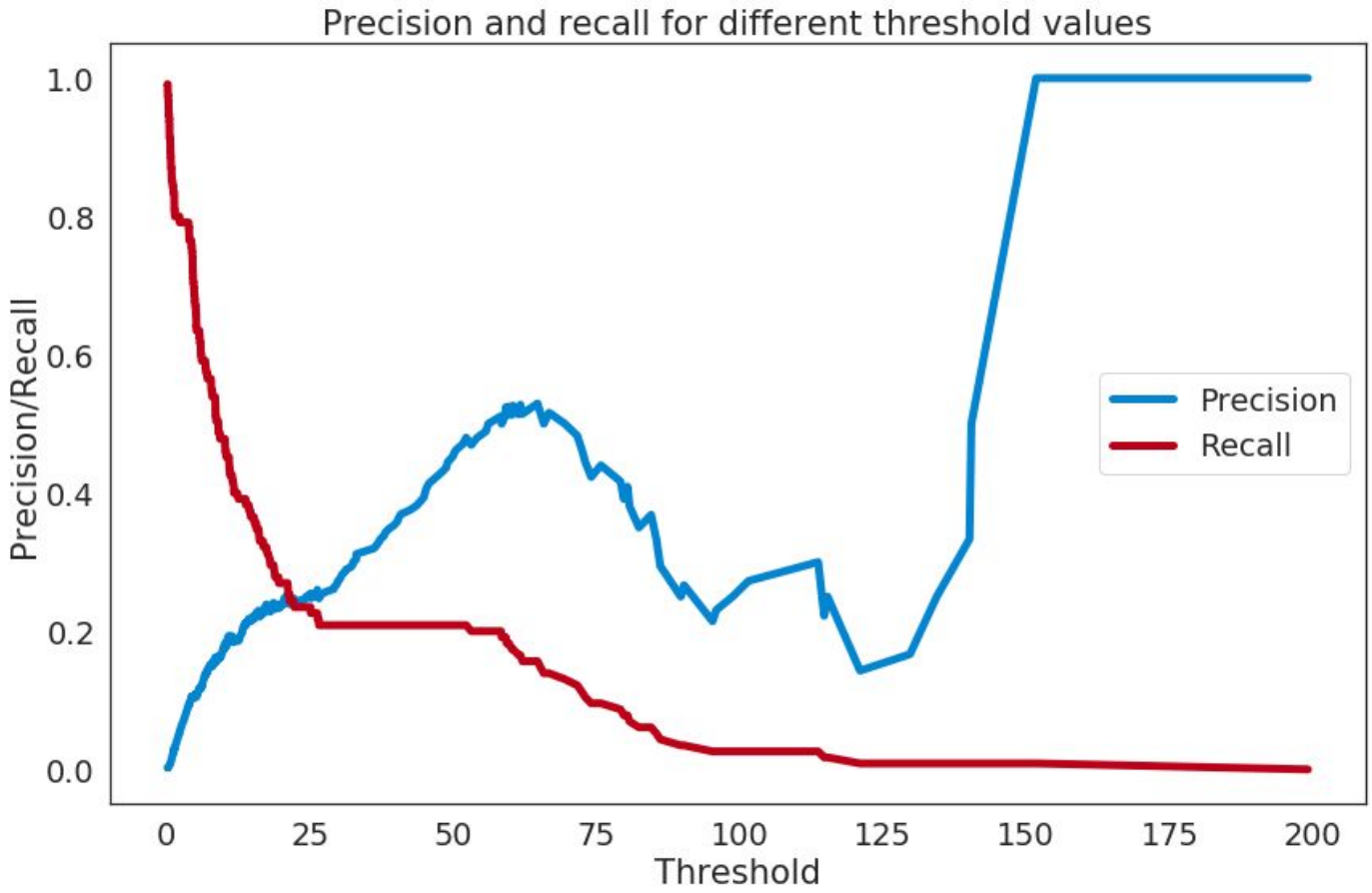
While the model is skilled (above the no-skill line), the AUC is much lower than before. Previously, for the logistic regression model, the AUC was 0.8427. Now, for the Autoencoder model, it's 0.1933. We can see that Precision drops quickly, but then rises again as Recall is improved. However for Recall starting at 0.2 and above, it drops again and keeps going down.

Defining a Reconstruction Error Threshold

The Precision-Recall vs threshold plot looks different as well. This time we are not calculating probabilities, but reconstruction error, so the thresholds are no longer between 0 and 1, they are different values for reconstruction error.

```
plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=5)
plt.plot(threshold_rt, recall_rt[1:], label="Recall",linewidth=5)
```

```
plt.title('Precision and recall for different threshold values')
plt.xlabel('Threshold')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()
```



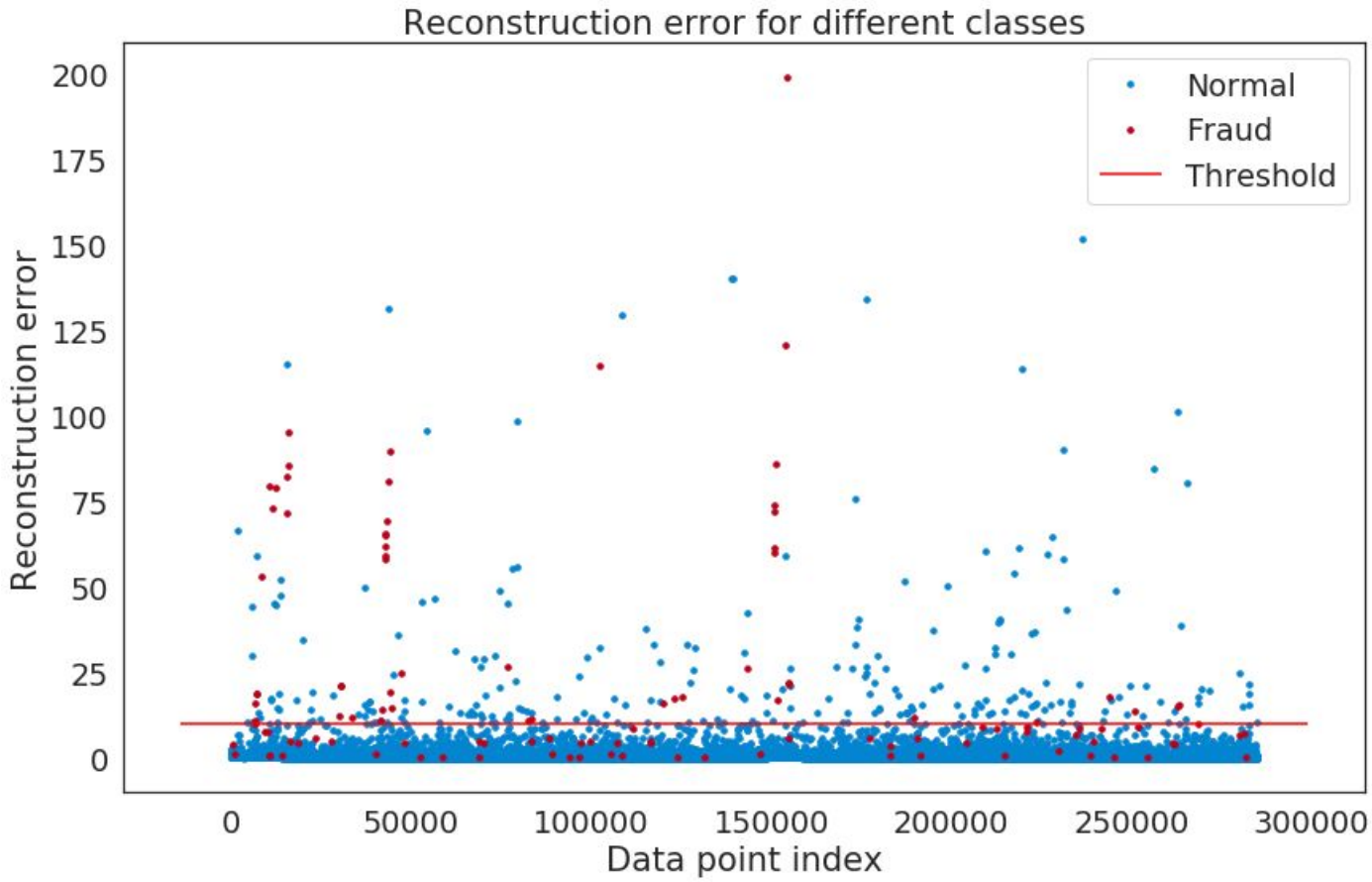
We can see the trade-off between Precision and Recall is more dramatic than before. We want high Recall, so will need to choose a low threshold, which means we'll have to settle for a low Precision as well.

Choosing a threshold of 10, we can plot the Reconstruction Error for different classes, against the threshold.

```
threshold_fixed = 10
groups = error_df.groupby('Actual')
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 different classes")
plt.ylabel("Reconstruction error")
plt.xlabel("Data point index")
plt.show();
```



It's clear that while the majority of normal transactions have low reconstruction error, there are enough that have higher than average that it's very difficult to both eliminate False Positives and correctly identify True Positives. We can't lower the threshold to capture more frauds without letting in a deluge of False Positives.

Final Tally

As before we can convert predictions to class assignments (zeros and ones) by comparing each value against the threshold.

```
preds_y_binary = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error]
print(classification_report(test_y, preds_y_binary))
```

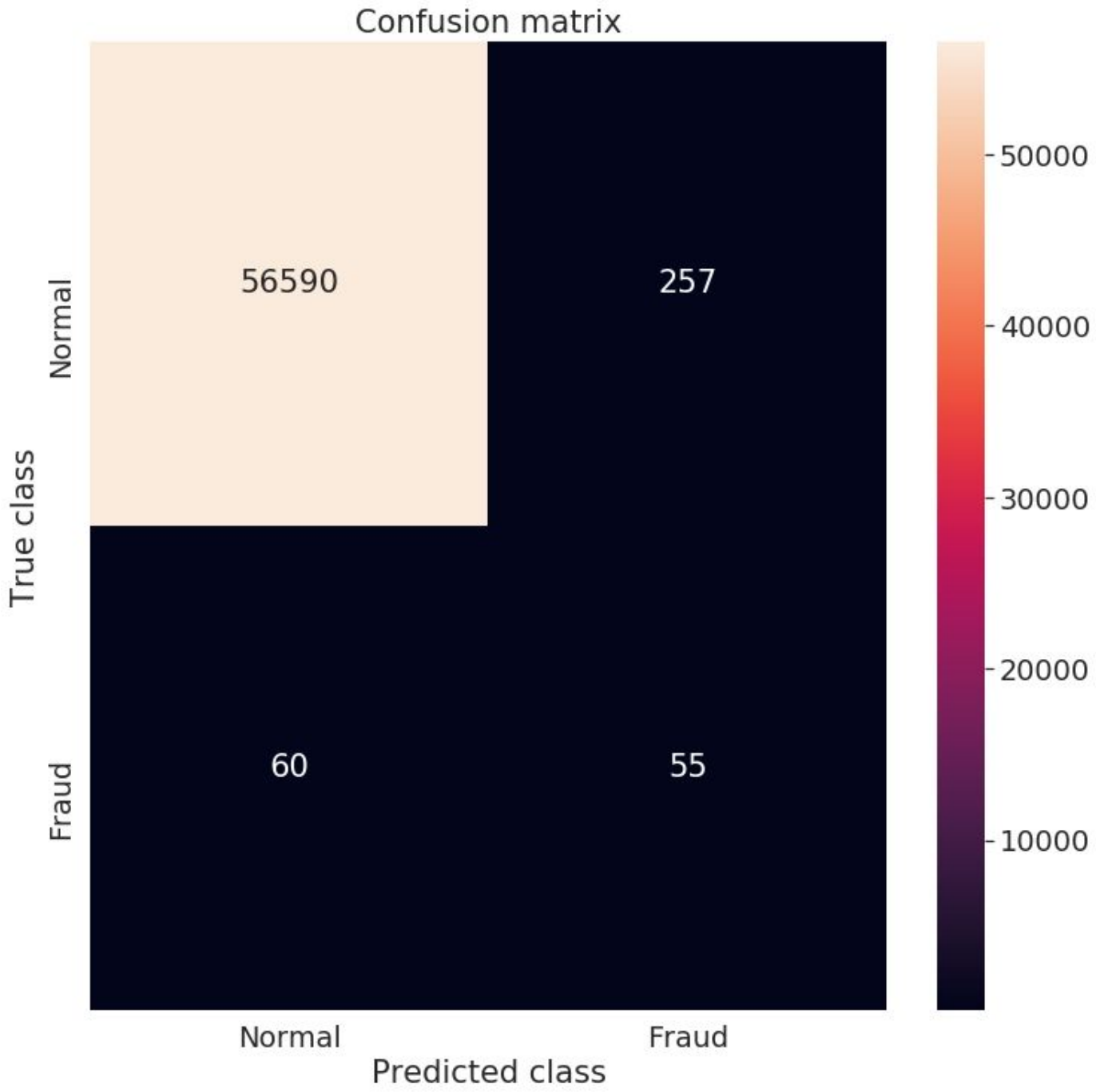
	precision	recall	f1-score	support
0	1	1	1	56847
1	0.18	0.48	0.26	115
micro avg	0.99	0.99	0.99	56962
macro avg	0.59	0.74	0.63	56962
weighted avg	1	0.99	1	56962

With the threshold set at 10, Precision is 18% and Recall is 48%. The F1 score is 26%.

Finally we plot a Confusion Matrix.

```
pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]
conf_matrix = confusion_matrix(error_df.Actual, pred_y)

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()
```



We now have:

- True Positives: 55
- False Negatives: 60
- False Positives: 253
- True Positives: 56,594

Therefore, as shown above by the classification report:

- Precision = 55 / (55 + 253) = 0.17857 = 18%

- Recall = 55 / (55 + 50) = 0.4783 = 48%

Previously we had Recall of 83% and Precision of 68%.

So it seems the Autoencoder, while much better than an unskilled model, does not beat the more straightforward supervised learning approach.

Using the extended test data

One thing remains. When training the Autoencoder, 80% of the fraud data was put aside, since it was not needed for training. For a direct comparison against the previous model, it was important to have the test data be the same size. But now, we can reinject the fraud data that was put aside, into the test data, and redo the predictions to see how the autoencoder model does with additional fraud data points to identify.

```
preds = autoencoder.predict(test_x_extended)
mse = np.mean(np.power(test_x_extended - preds, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse,
                         'Actual': test_y_extended})
```

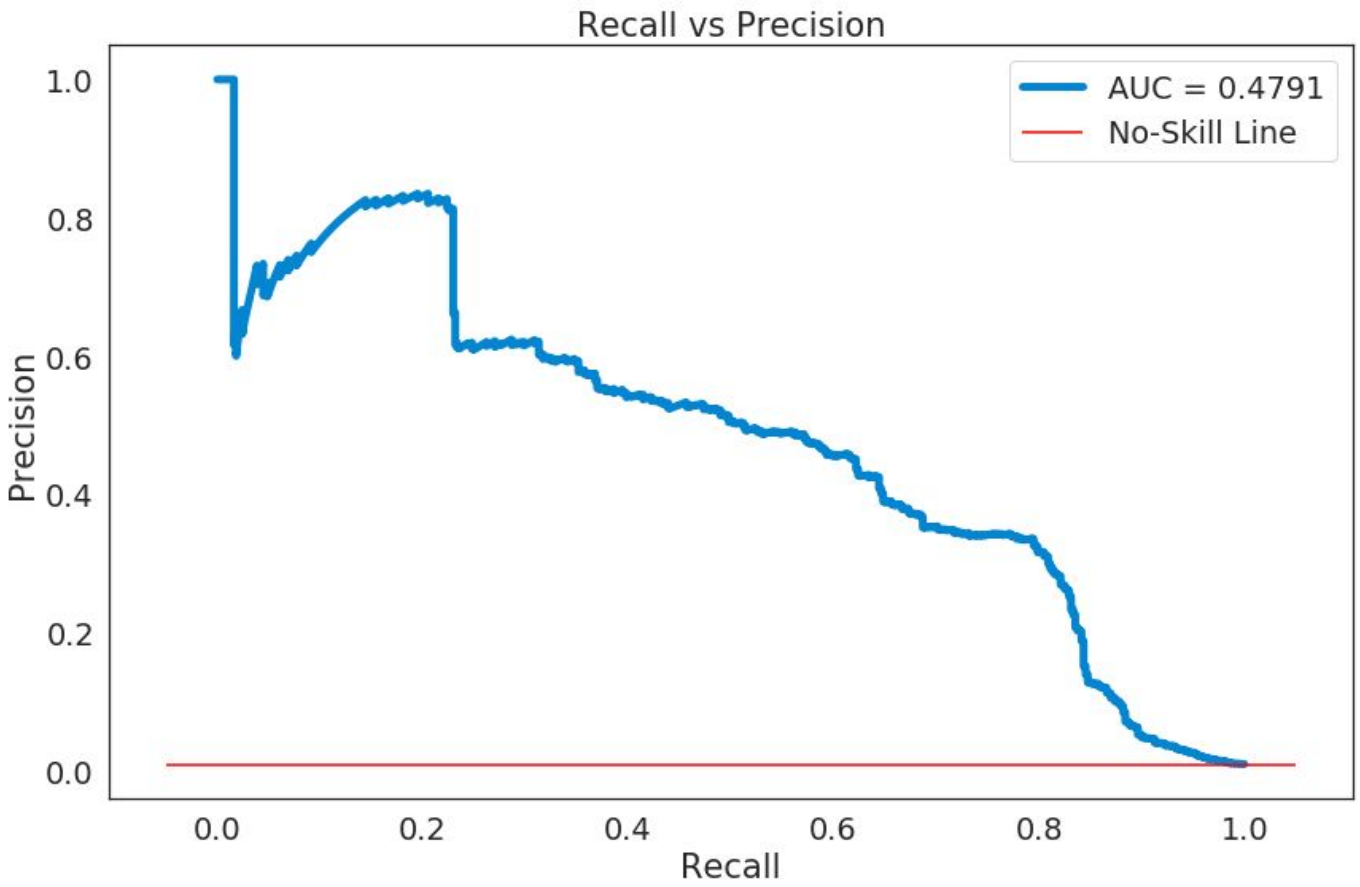
Now we have all 492 fraud data points as part of the predictions:

```
num_fraud = error_df.Actual[error_df.Actual == 1].count()
num_normal = error_df.Actual[error_df.Actual == 0].count()
no_skill_level = num_fraud/(num_fraud + num_normal)

print(num_fraud, num_normal, no_skill_level)
```

We'll skip straight to the more relevant Precision-Recall plot.

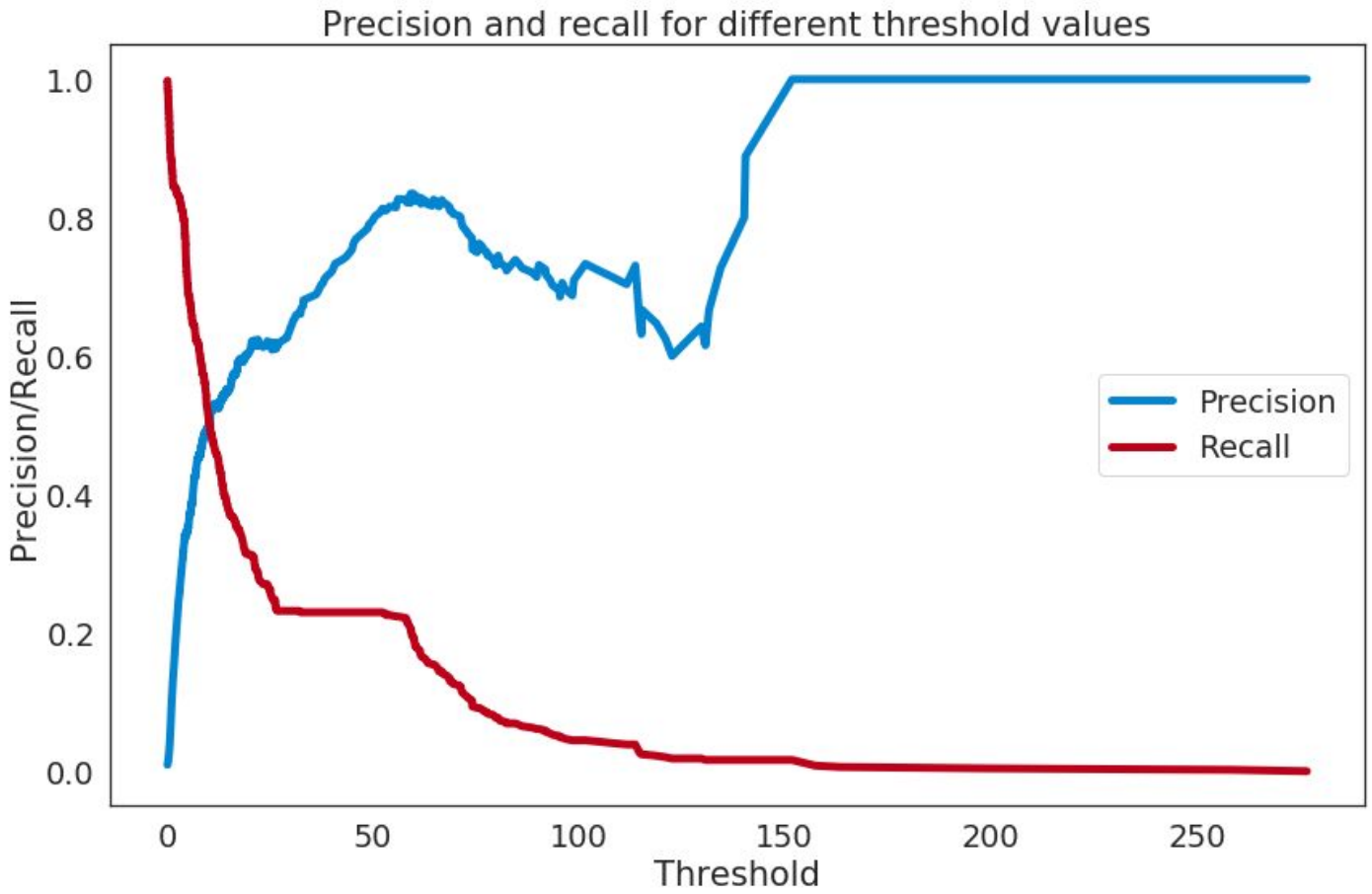
```
precision_rt, recall_rt, threshold_rt = precision_recall_curve(error_df.Actual,
error_df.Reconstruction_error)
pr_auc = auc(recall_rt, precision_rt)
fig, ax = plt.subplots()
ax.plot(recall_rt, precision_rt, linewidth=5, label='AUC = %0.4f'% pr_auc)
ax.hlines(no_skill_level, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label='No-Skill
Line')
ax.legend(loc='upper right')
plt.title('Recall vs Precision')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()
```



The Recall-Precision curve looks the same, but AUC has doubled. This is because although the shape is similar, the curve has shifted upwards, meaning we get a little more Precision bang, for our Recall buck.

To get a sense of the Precision and Recall against threshold values, we'll plot the usual graph.

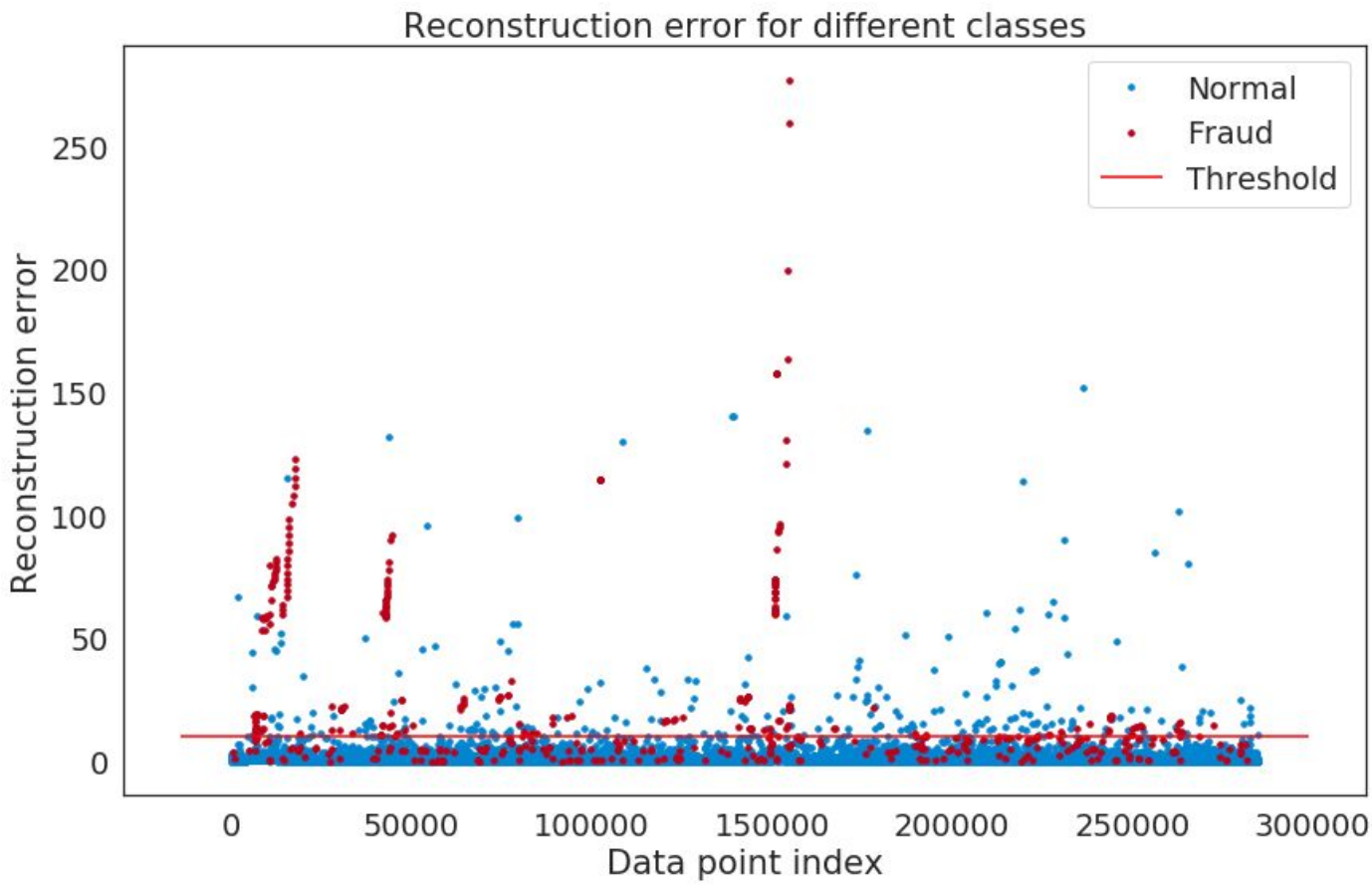
```
plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=5)
plt.plot(threshold_rt, recall_rt[1:], label="Recall",linewidth=5)
plt.title('Precision and recall for different threshold values')
plt.xlabel('Threshold')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()
```



Again, the two curves look the same as before, but now we have a bit more Precision for higher Recall values. We can't afford a higher threshold, but Recall should be slightly better.

```
threshold_fixed = 10
groups = error_df.groupby('Actual')
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 different classes")
plt.ylabel("Reconstruction error")
plt.xlabel("Data point index")
plt.show();
```

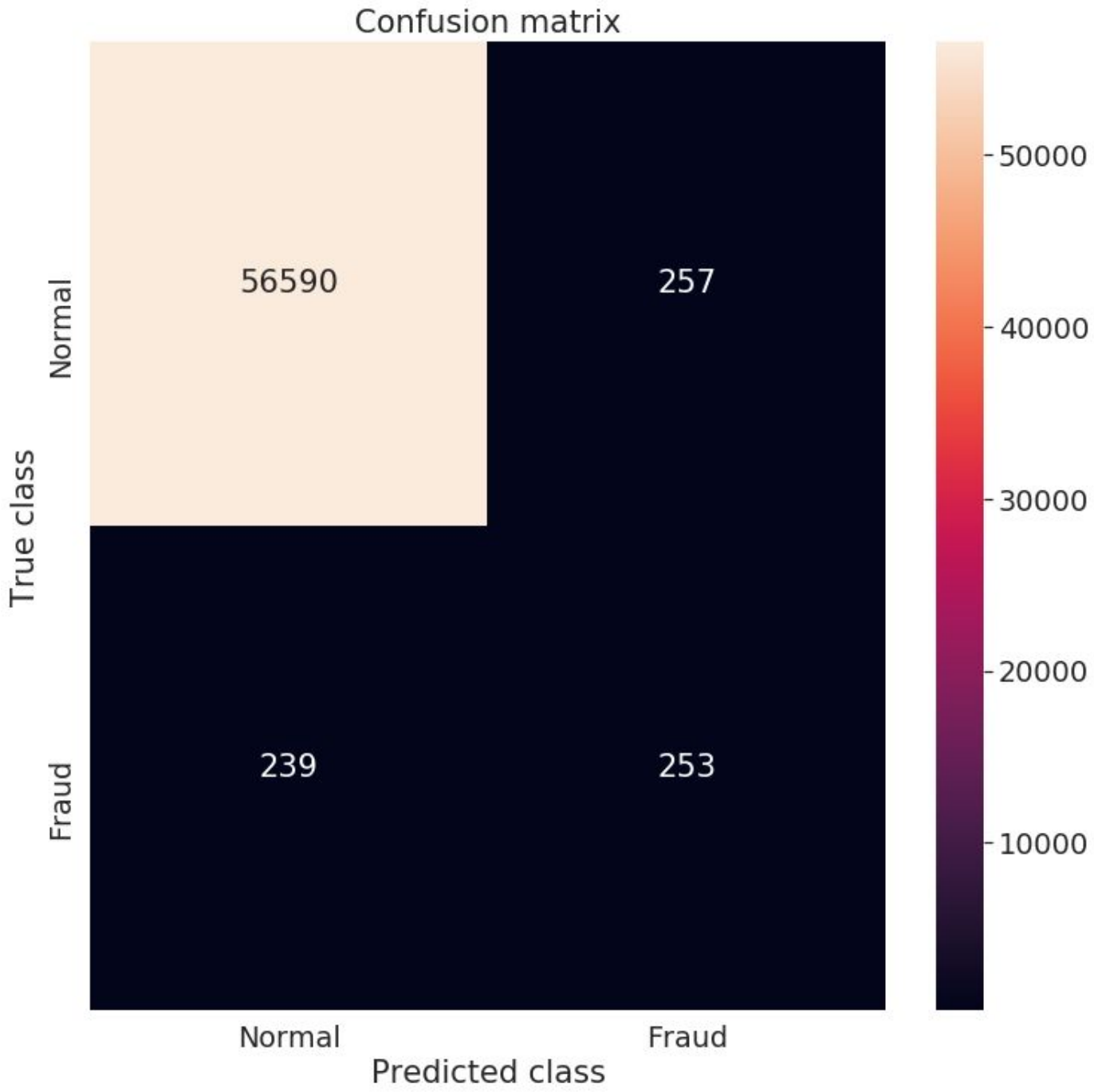


There's a lot more fraud events visible. We also see there are some clusters, well above the threshold. But again, as before, there's a lot of fraud event sitting at the same level as normal events, so we can't lower the threshold without letting in False Positives.

Here's the Confusion Matrix.

```
pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]
conf_matrix = confusion_matrix(error_df.Actual, pred_y)

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()
```



As always, we convert the predictions to classes based on the thresholds. Then we can get the Recall, Precision, and F1 scores.

```
preds_y_binary = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error]

print(classification_report(test_y_extended, preds_y_binary))
```

	precision	recall	f1-score	support
0	1	1	1	56847
1	0.5	0.51	0.5	492
micro avg	0.99	0.99	0.99	57339
macro avg	0.75	0.75	0.75	57339
weighted avg	0.99	0.99	0.99	57339

Recall has improved from 48% to 51%, and Precision has improved from 18% to 50%! However, the binary classification model still had much better Recall at 84% and Precision at 68%.

Binary Classification with weighted classes

There is another approach to tackling the issue on an imbalanced dataset.

The traditional approach is over-sampling or under-sampling. Over-sampling would be taking the minority class (fraud data) and replicating it a bunch until the datasets are balanced. The risk with this approach is that it leads towards over-fitting.

Under-Sampling is to cut out a bunch of the majority class (normal transactions) until the classes are balanced. The problem with this approach is that there is a dramatic loss of information, so the model does not generalize as well as it otherwise would.

Another method, similar in spirit but different in detail is weighting the classes during training. Keras's `fit()` function has a parameter called `class_weights`. This lets you assign a weight to each of your classes.

This a way of handling imbalanced classes. Instead of tackling the problem on sampling side with over-sampling the minority class (fraud data), this parameter assists in taking bigger steps during optimization by giving more weight to the loss of the minority class, resulting in a similar outcome.

Setup and Train

Split the testing and training exactly as before:

```
RANDOM_SEED = 314
```

```
TEST_PCT = 0.2

train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT,
random_state=RANDOM_SEED)

train_y = train_x['Class']
train_x = train_x.drop(['Class'], axis=1)

test_y = test_x['Class']
test_x = test_x.drop(['Class'], axis=1)

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

Setup the exact same model as before, with the same parameters.

```
nb_epoch = 50
batch_size = 128
input_dim = train_x.shape[1] #num of columns, 30

model = Sequential()
model.add(Dense(120, input_shape=(input_dim,), activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(60, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(30, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

This time however we will introduce a dictionary of weights. We will assign labels of 0 (Normal Transactions) a weight of 1. We will assign labels of 1 (Fraudulent Transactions) with a weight of 50. This will heavily favor (50:1) large step sizes when training on fraud data.

The class weights are stored in a simple Python dictionary.

```
class_weight = {0 : 1., 1: 50.}
```

The compilation parameters are the same.

```
model.compile(metrics=['accuracy'],
              loss='binary_crossentropy',
              optimizer='adam')
```

We'll setup the checkpoint as before, with a different name.

```
cp = ModelCheckpoint(filepath="mlp_class-weighted_fraud.h5",
                     save_best_only=True,
                     verbose=0)
```

Finally run the model almost exactly as before, but this time pass in the class weights to the class_weight argument.

```
history = model.fit(train_x, train_y,
                    epochs=nb_epoch,
                    batch_size=batch_size,
                    class_weight=class_weight, # Pass in the class weights, 50:1
                    shuffle=True,
                    validation_data=(test_x, test_y),
                    verbose=1,
                    callbacks=[cp]).history
```

Load the model.

```
model = load_model('mlp_class-weighted_fraud.h5')
```

Get the predictions.

```
preds_y = model.predict(test_x).flatten()
```

Setup the dataframe.

```
error_df = pd.DataFrame({'Predictions': preds_y,
                         'Actual': test_y})
```

Recall vs Precision

We'll skip the ROC curve, since it is not useful, and go straight to the Recall vs Precision curve.

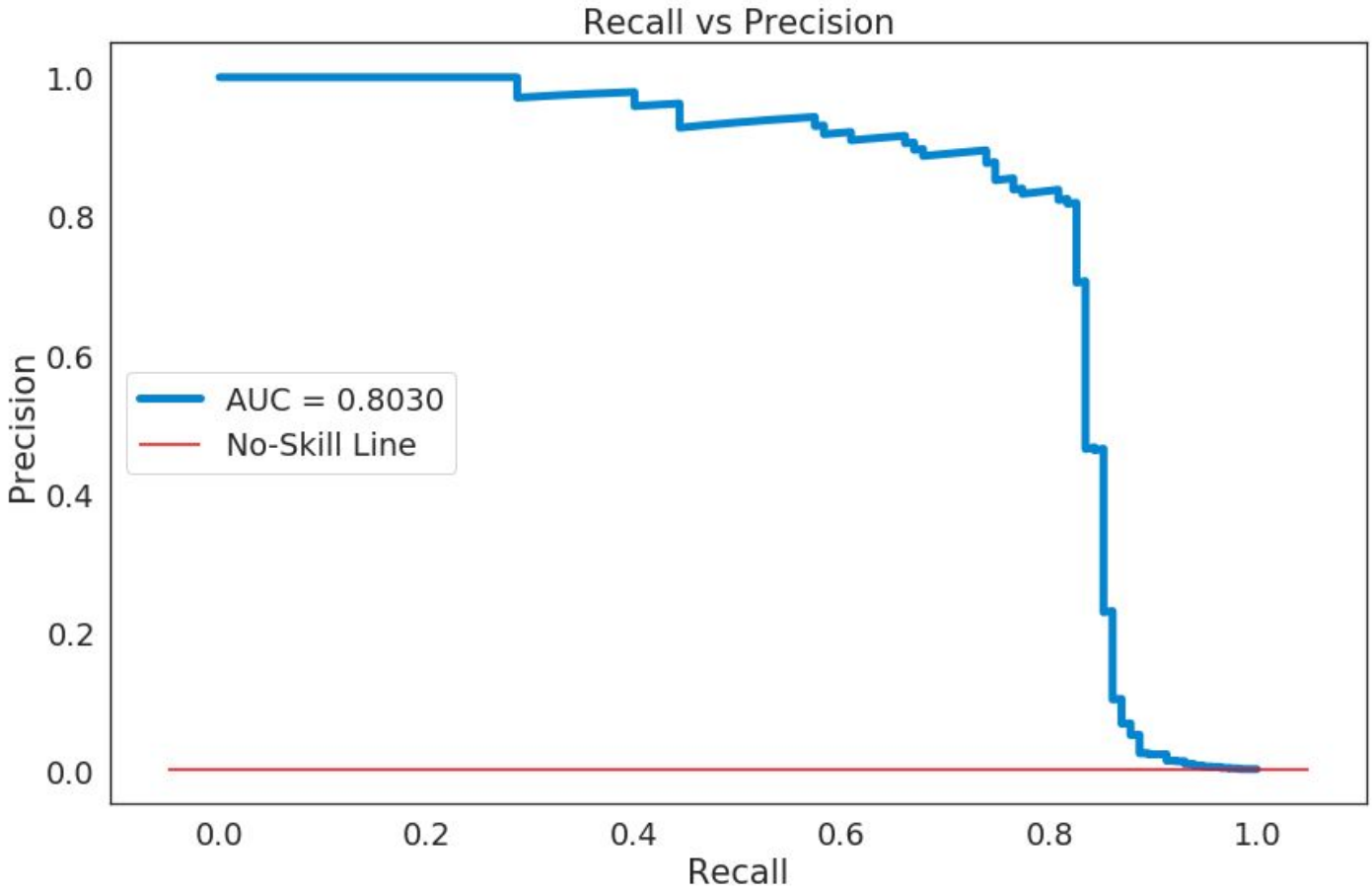
Set the No-Skill Level.

```
num_fraud = error_df.Actual[error_df.Actual == 1].count()
num_normal = error_df.Actual[error_df.Actual == 0].count()
no_skill_level = num_fraud/(num_fraud + num_normal)
```

```
print(num_fraud, num_normal, no_skill_level)
(115, 56847, 0.0020188897861732383)
```

And plot the Precision-Recall Curve.

```
precision_rt, recall_rt, threshold_rt = precision_recall_curve(error_df.Actual,
error_df.Predictions)
pr_auc = auc(recall_rt, precision_rt)
fig, ax = plt.subplots()
ax.plot(recall_rt, precision_rt, linewidth=5, label='AUC = %0.4f'% pr_auc)
ax.hlines(no_skill_level, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label='No-Skill
Line')
ax.legend(loc='center left')
plt.title('Recall vs Precision')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()
```

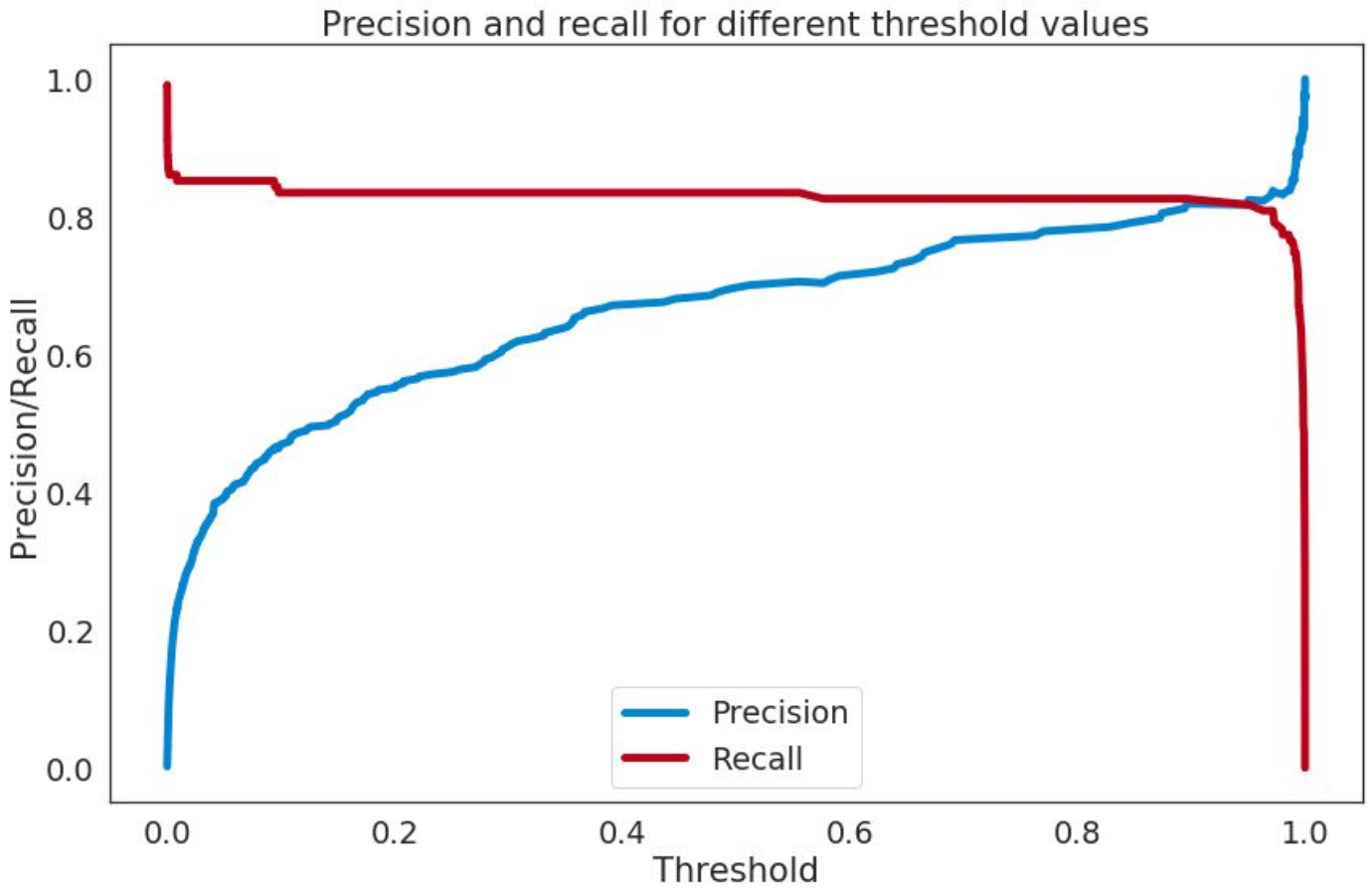


The results are somewhat interesting. The AUC is smaller than before 80% vs. 83.3%. Higher AUC is better, so this doesn't look ideal on first glance.

Threshold values

Now we'll look at the Recall and Precision for different threshold values.

```
plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=5)
plt.plot(threshold_rt, recall_rt[1:], label="Recall",linewidth=5)
plt.title('Precision and recall for different threshold values')
plt.xlabel('Threshold')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()
```

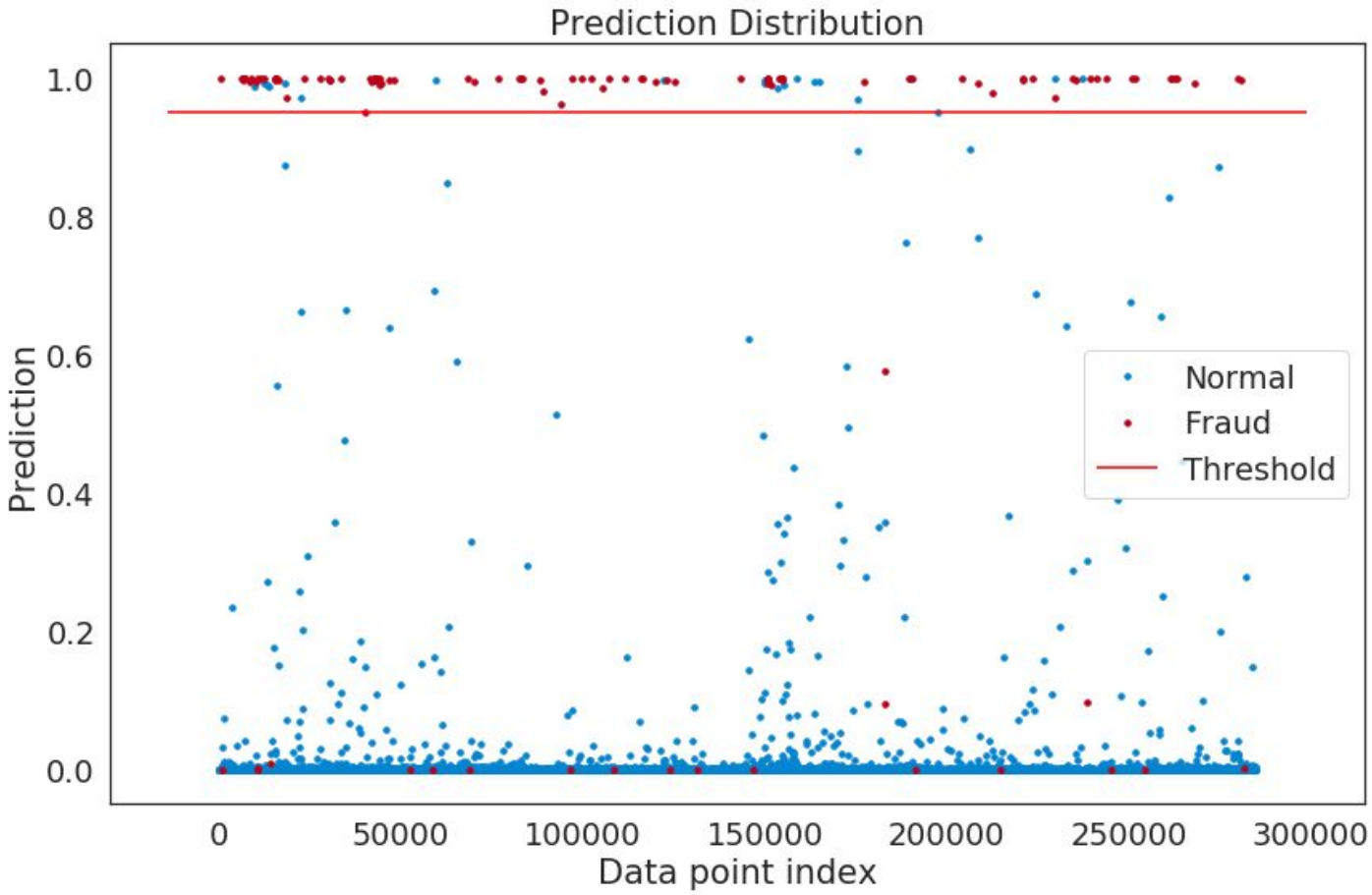


The results of applying class weights are much more apparent here. The results are significantly different than before. As we increase the threshold, for most values, we don't have much loss in Recall but we have a steadily increasing Precision. It seems as though the probabilities close to 1 are mostly True Positives.

We can look at the distribution to visualize. The threshold can be raised pretty high.

```
threshold_fixed = 0.95
groups = error_df.groupby('Actual')
fig, ax = plt.subplots()

for name, group in groups:
    ax.plot(group.index, group.Predictions, 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("Prediction Distribution")
plt.ylabel("Prediction")
plt.xlabel("Data point index")
plt.show();
```



The majority of fraud events have bubbled up to the top. This is why we don't lose much Recall as we raise the Threshold. Also The Normal events are scattered, but mostly distributed towards the lower probabilities. This is why Precision goes up steadily as we raise the threshold.

Final Tally

Now for the numbers.

Finally get the predicted classes.

```
preds_y_binary = [1 if e > threshold_fixed else 0 for e in error_df.Predictions]
```

```
print(classification_report(test_y, preds_y_binary))
```

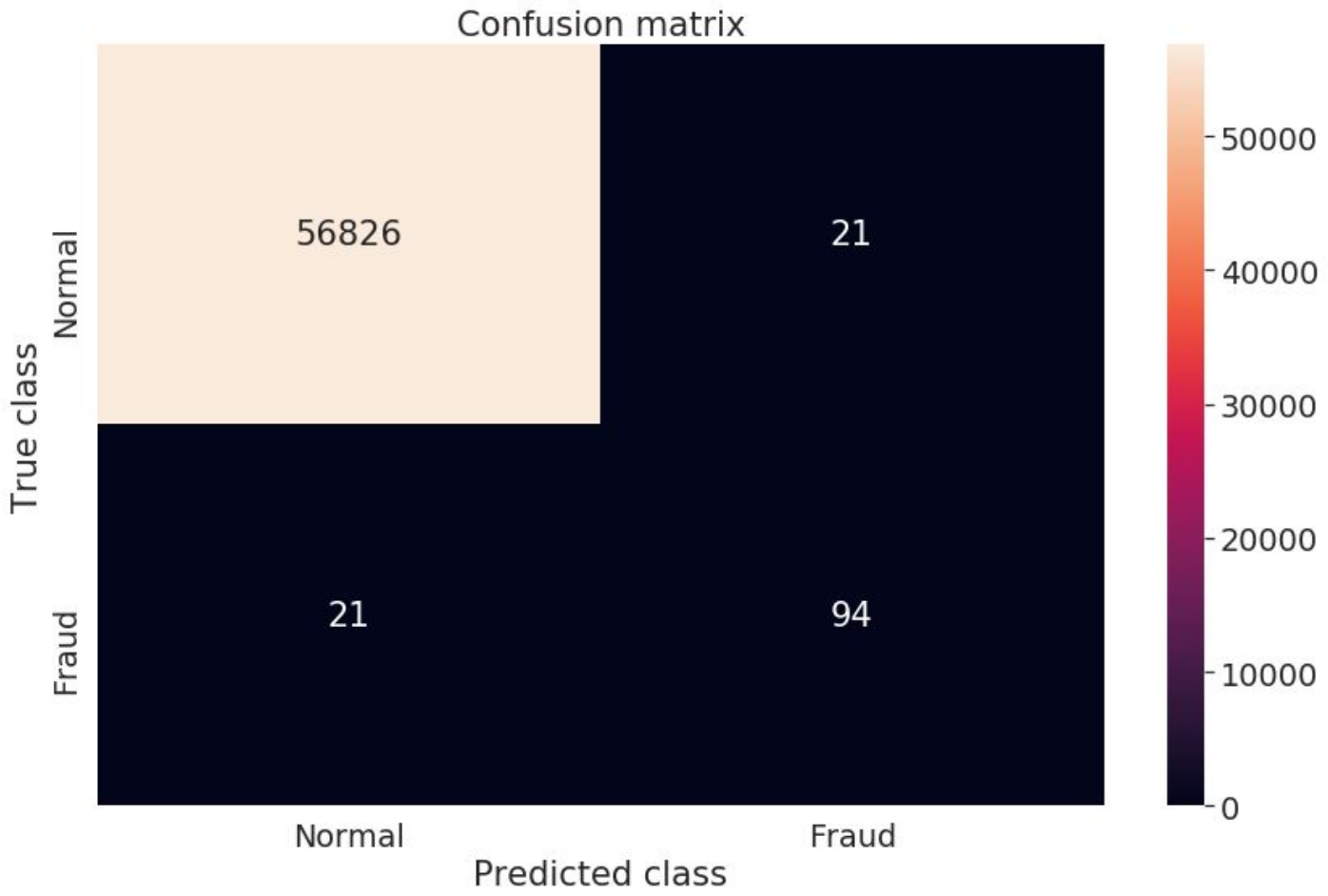
	precision	recall	f1-score	support
0	1	1	1	56847
1	0.82	0.82	0.82	115
micro avg	1	1	1	56962
macro avg	0.91	0.91	0.91	56962
weighted avg	1	1	1	56962

Recall and Precision or both quite good, at 82%.

The Confusion Matrix shows the numerical breakdown.

```
conf_matrix = confusion_matrix(test_y, preds_y_binary)
```

```
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()
```



Previously (without the weighted classes) we had:

- TP - 95
- FN - 20
- FP - 45
- TN - 56,802
- Precision - 68%
- Recall - 83%
- F1 Score - 75%

Now (with weighted classes) we have:

- TP - 94
- FN - 21
- FP - 21
- TN - 56,826
- Precision - 82%
- Recall - 82%
- F1 Score - 82%

Therefore, with weighted classes, we're not any better at identifying Fraud events, and we miss just as many. However, we're better about not misidentifying Normal transactions as Fraud (we have less False Positives). So overall, this is the best model for identifying fraud relative to all the two previous ones.

Youtube Videos

2-Minute:
15-Minute:

Sources

Data

- Credit Card Fraud Detection - Anonymized credit card transactions labeled as fraudulent or genuine: https://www.kaggle.com/mlg-ulb/creditcardfraud

Articles/Blogs

- Fraud Detection Using Autoencoders in Keras with a TensorFlow Backend - David Ellison, PhD: https://www.datascience.com/blog/fraud-detection-with-tensorflow
- Credit Card Fraud Detection using Autoencoders in Keras - Venelin Valkov: https://medium.com/@curiousily/credit-card-fraud-detection-using-autoencoders-in-keras-tensorflow-for-hackers-part-vii-20e0c85301bd
- How and When to Use ROC Curves and Precision-Recall Curves for Classification in Python - Jason Brownlee: https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/
- Simple Approach for Handling Imbalanced Dataset - Credit Card Fraud Detection: http://www.coderschool.vn/blog/simple-approach-for-handling-imbalanced-dataset-credit-card-fraud-detection/
- Precision and Recall: https://en.wikipedia.org/wiki/Precision_and_recall
- From Logistic Regression in SciKit-Learn to Deep Learning with TensorFlow: A fraud detection case study - Matthias Groncki, Part I: https://ipythonquant.wordpress.com/2018/05/08/from-logistic-regression-in-scikit-learn-to-deep-learning-with-tensorflow-a-fraud-detection-case-study-part-i/
- What to do when the labels are skewed: tackling the Accuracy Paradox - Mark Ryan: https://medium.com/@markryan_69718/what-to-do-when-the-labels-are-skewed-tackling-the-accuracy-paradox-7b72bd4a9ccc