case-study-3-1

May 12, 2025

1 Case Study: Fraud detection using Autoencoders in Keras

1.1 General Setup

In [6]:
```
# Import required packages
# matplotlib inline
import pandas as pd
import numpy as np
from scipy import stats
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score, classification_report, auc, roc_curve
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.preprocessing import StandardScaler
from pylab import rcParams
from keras.models import Model, load_model
from keras.layers import Input, Dense
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras import regularizers
```

In [7]:
```
#set random seed and percentage of test data
RANDOM_SEED = 314 #used to help randomly select the data points
TEST_PCT = 0.2 # 20% of the data
```

In [8]:
```
#set up graphic style (color scheme from xkcd.com)
rcParams['figure.figsize'] = 14, 8.7 # Golden Mean
LABELS = ["Normal","Fraud"]

col_list = ["cerulean","scarlet"]# https://xkcd.com/color/rgb/
sns.set(style='white', font_scale=1.75)
sns.set_palette(sns.xkcd_palette(col_list))
```

1.2 Description of the "Credit Card Fraud Detection" Dataset (Kaggle)

The "Credit Card Fraud Detection" dataset from Kaggle is a widely used benchmark for evaluating fraud detection systems. It contains transactions made by European cardholders over two days in September 2013. The dataset includes **284,807 transactions**, of which **492 are fraudulent**, resulting in a highly **imbalanced dataset** (only about 0.172% fraud cases).

Key Features: - **Time**: The number of seconds elapsed between the first transaction in the dataset and the current transaction. - **V1 to V28**: These are the result of a **PCA (Principal Component Analysis) transformation** applied to the original features for confidentiality reasons. The original features (like merchant name, location, etc.) are not included to protect sensitive information. - **Amount**: The transaction amount, which can be useful for modeling. - **Class**: The target variable. It takes the value **1** in case of fraud and **0** otherwise.

Important Characteristics: - **Strong class imbalance**: Most machine learning models must be adapted or tuned (e.g., via resampling techniques or specialized loss functions) to handle the rare positive (fraud) cases. - **PCA-transformed features**: No direct interpretation of the individual V1–V28 components is possible, but patterns can still be learned by models. - **Real-world relevance**: Despite anonymization, the dataset reflects the challenges of detecting fraud in real transaction flows.

Common Uses: - Testing anomaly detection algorithms. - Experimenting with sampling strategies (e.g., SMOTE, undersampling). - Benchmarking classification models, especially under imbalanced conditions. - Studying precision-recall tradeoffs rather than focusing only on accuracy.

1.3 Import and Check Data

```
In [9]: df = pd.read_csv("C:/DATA/creditcard.csv")
```

```
In [10]: df.head(5)
```

```
Out[10]:    Time        V1        V2        V3        V4        V5        V6        V7  \
         0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
         1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
         2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
         3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
         4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

                 V8        V9  …       V21       V22       V23       V24       V25  \
         0  0.098698  0.363787  … -0.018307  0.277838 -0.110474  0.066928  0.128539
         1  0.085102 -0.255425  … -0.225775 -0.638672  0.101288 -0.339846  0.167170
         2  0.247676 -1.514654  …  0.247998  0.771679  0.909412 -0.689281 -0.327642
         3  0.377436 -1.387024  … -0.108300  0.005274 -0.190321 -1.175575  0.647376
         4 -0.270533  0.817739  … -0.009431  0.798278 -0.137458  0.141267 -0.206010

                V26       V27       V28  Amount  Class
         0 -0.189115  0.133558 -0.021053  149.62      0
         1  0.125895 -0.008983  0.014724    2.69      0
         2 -0.139097 -0.055353 -0.059752  378.66      0
         3 -0.221929  0.062723  0.061458  123.50      0
         4  0.502292  0.219422  0.215153   69.99      0

         [5 rows x 31 columns]
```

```
In [11]: df.shape
```

```
Out[11]: (284807, 31)
```

```
In [12]: # No of null values in dataset
         df.isnull().values.sum()
```

```
Out[12]: 0
```

Indeed the data seems to be cleaned and loaded as we expect. Now we want to check if we have the expected number of normal and fraudulent rows of data. We will simply pull the "Class" column and count the number of normal (0) and fraud (1) rows.

```
In [13]: pd.Series(df['Class']).value_counts(sort=True)
```

```
Out[13]: Class
         0    284315
         1       492
         Name: count, dtype: int64
```

The counts are as expected (284,315 normal transactions and 492 fraud transactions). As is typical in fraud and anomaly detection in general, this is a very unbalanced dataset.

We will cut up the dataset into two data frames, one for normal transactions and the other for fraud

```
In [14]: normal_df = df[df['Class']==0]
         fraud_df  = df[df['Class']==1]
```

Summary Statistics of the Transaction Amount Data

```
In [15]: normal_df.Amount.describe()
```

```
Out[15]: 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
```

```
In [16]: fraud_df.Amount.describe()
```

```
Out[16]: 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
```

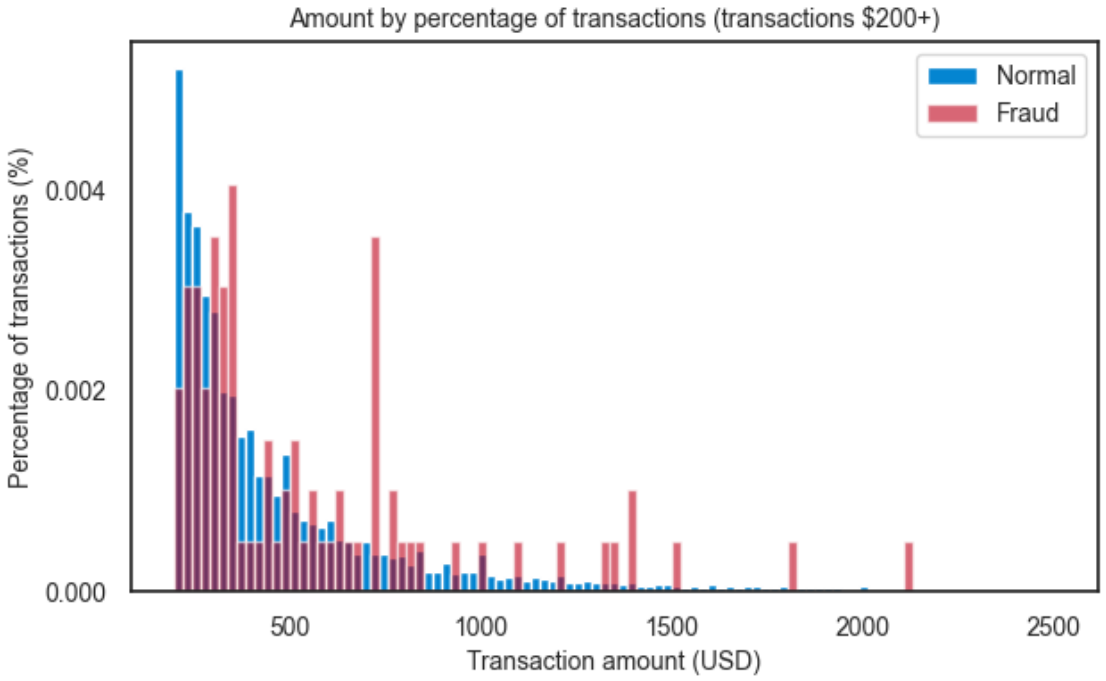
Although the mean is a little higher in the fraud transactions, it is certainly within a standard deviation and so is unlikely to be easy to discriminate in a highly precise manner between the classes with pure statistical methods.

1.4 Visual Exploration of the Transaction Amount Data

In anomaly detection datasets it is common to have the areas of interest "washed out" by abundant data. In this dataset, a lot of low-value transactions that will be generally uninteresting (buying cups of coffee, lunches, etc). This abundant data is likely to wash out the rest of the data, so looking at transactions which are $200+

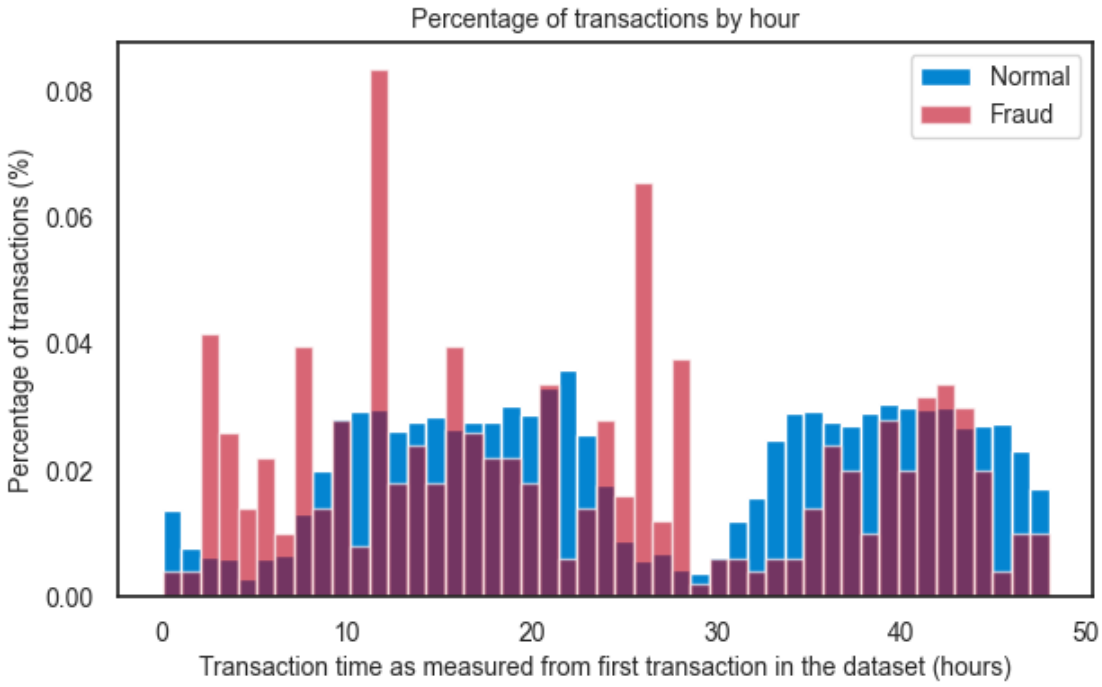
```
In [17]: # Define bins for the histogram:
         # Create 100 evenly spaced values between $200 and $2500.
         # These bins will define the ranges for transaction amounts in the histograms.
         bins = np.linspace(200, 2500, 100)
```

```
# Set the figure size for the plot
plt.figure(figsize=(7, 4))

# Plot histogram for normal (non-fraudulent) transactions:
# - 'Amount' column from the normal transactions dataframe (normal_df).
# - Use the previously defined bins.
# - 'alpha=1' means fully opaque.
# - 'density=True' normalizes the histogram so the total area equals 1 (useful for comparing d
# - 'label' sets the label that will appear in the legend.
plt.hist(normal_df.Amount, bins, alpha=1, density=True, label='Normal')

# Plot histogram for fraudulent transactions:
# - 'Amount' column from the fraud transactions dataframe (fraud_df).
# - Same bins as above for direct comparison.
# - 'alpha=0.6' makes this plot slightly transparent (easier to compare overlapping areas).
# - 'density=True' again normalizes the histogram.
# - 'label' identifies the curve as representing fraud cases.
plt.hist(fraud_df.Amount, bins, alpha=0.6, density=True, label='Fraud')

# Add a legend in the upper right corner to differentiate 'Normal' and 'Fraud' plots
plt.legend(loc='upper right', fontsize=10)
# Set the title of the plot
plt.title("Amount by percentage of transactions (transactions \$200+)", fontsize=10)
# Set the x-axis label
plt.xlabel("Transaction amount (USD)", fontsize=10)
# Set the y-axis label
plt.ylabel("Percentage of transactions (%)", fontsize=10)
plt.xticks(fontsize=10)    # X-axis ticks font size
plt.yticks(fontsize=10)    # Y-axis ticks font size
# Display the plot
plt.show()
```



Since the fraud cases are relatively few in number compared to bin size, we see the data looks predictably more variable. In the long tail, especially, we are likely observing only a single fraud transaction. ***It would be hard to differentiate fraud from normal transactions by transaction amount alone.***

```
In [18]: bins = np.linspace(0, 48, 48) #48 hours
         plt.figure(figsize=(7, 4))
         plt.hist((normal_df.Time/(60*60)), bins, alpha=1, density=True, label='Normal')
         plt.hist((fraud_df.Time/(60*60)), bins, alpha=0.6, density=True, label='Fraud')
         plt.legend(loc='upper right', fontsize=10)
         plt.title("Percentage of transactions by hour", fontsize=10)
         plt.xlabel("Transaction time as measured from first transaction in the dataset (hours)", fonts
         plt.ylabel("Percentage of transactions (%)", fontsize=10);
         # plt.hist((df.Time/(60*60)),bins)
         plt.xticks(fontsize=10)    # X-axis ticks font size
         plt.yticks(fontsize=10)    # Y-axis ticks font size

         plt.show()
```



Hour "zero" corresponds to the hour the first transaction happened and not necessarily 12-1am. Given the heavy decrease in normal transactions from hours 1 to 8 and again roughly at hours 24 to 32, we can assume those time correspond to nighttime for this dataset. If this is true, fraud tends to occur at higher rates during the night. Statistical tests could be used to give evidence for this fact, but are not in the scope of this article. Again, however, ***the potential time offset between normal and fraud transactions is not enough to make a simple, precise classifier***.

1.5 Using Autoencoder

1.5.1 Normalize and Scale Data¶

Both time and amount have very different magnitudes, which will likely result in the large magnitude value "washing out" the small magnitude value. It is therefore common to scale the data to similar magnitudes. As most of the data (other than 'time' and 'amount') result from the product of a PCA analysis. The PCA done on the dataset transformed it into standard-normal form. We will do the same to the 'time' and 'amount' columns.

```
In [19]: df_norm = df.copy()
         # Reshape your data either using array.reshape(-1, 1) if your data has a single feature
         # or array.reshape(1, -1) if it contains a single sample.
         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))
```

1.5.2 Dividing Training and Test Set

Now we split the data into training and testing sets according to the percentage and with a random seed we wrote at the beginning of the code.

```
In [20]: train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT, random_state=RANDOM_SEED)
         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 set
         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
```

1.6 Creating The Model

1.6.1 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, we are encoding 30 columns into 14 dimensions so we are expecting high-level features to be represented by roughly two columns ($30/14 = 2.1$). Of those high-level features, we are 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 [21]: nb_epoch      = 50
         batch_size    = 128
         input_dim     = train_x.shape[1] #num of columns, 30
         encoding_dim  = 18
         hidden_dim1   = 10 #int(encoding_dim / 2) #i.e. 7
         hidden_dim2   = 6
         learning_rate = 1e-7
```

```
In [22]: # This returns a tensor
         input_layer = Input(shape=(input_dim, ))

         # a layer instance is callable on a tensor, and returns a tensor
```

```
        # Dense implements the operation: output = activation(dot(input, kernel) + bias), where
        # activation is the element-wise activation function passed as the activation argument,
        # kernel is a weights matrix created by the layer, and
        # bias is a bias vector created by the layer (only applicable if use_bias is True).

        # activity_regularizer: Regularizer function applied to the output of the layer
        encoder = Dense(encoding_dim, activation="tanh",
                        activity_regularizer=regularizers.l1(learning_rate))(input_layer)
        encoder = Dense(hidden_dim1, activation="elu")(encoder)
        encoder = Dense(hidden_dim2, activation="tanh")(encoder)
        decoder = Dense(hidden_dim2, activation='elu')(encoder)
        decoder = Dense(hidden_dim1, activation='tanh')(decoder)
        decoder = Dense(input_dim, activation='elu')(decoder)

        # This creates a model that includes
        # the Input layer and four Dense layers
        autoencoder = Model(inputs=input_layer, outputs=decoder)
```

In [23]:
```
        # Configure the learning process, by compiling the model
        autoencoder.compile(optimizer='adam',
                            metrics=['accuracy'],
                            loss='mean_squared_error')
```

In [24]:
```
        # Saving the model
        cp = ModelCheckpoint(filepath="autoencoder_fraud.keras",
                             save_best_only=True,
                             verbose=0)
```

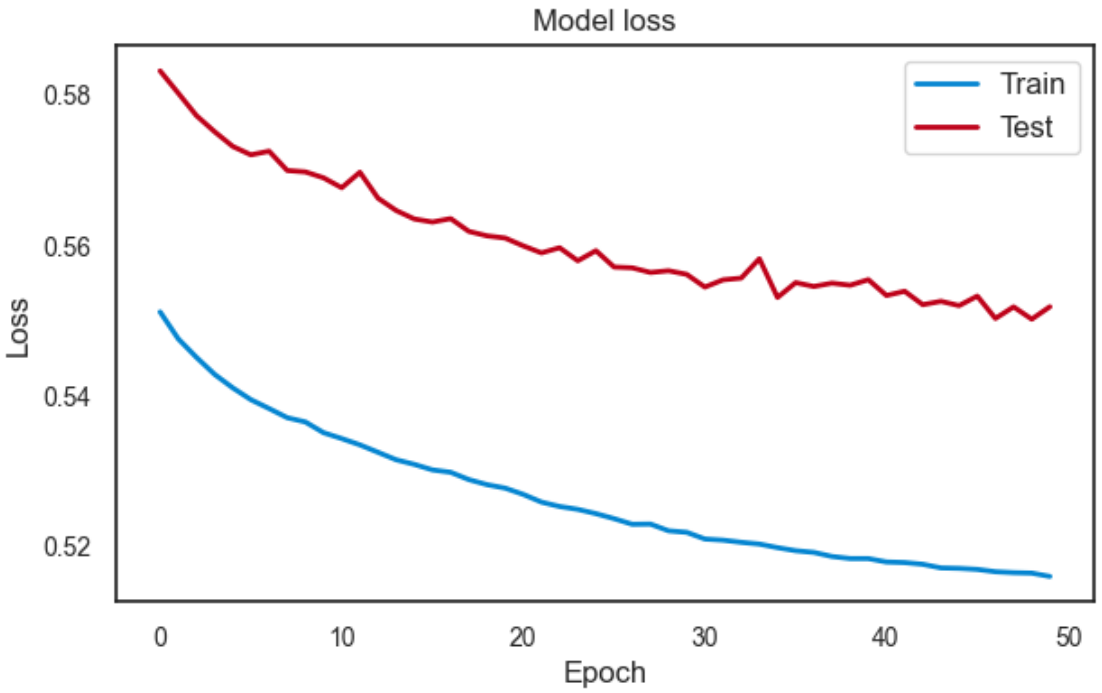
In [25]:
```
        # TensorBoard basic visualizations.
        # This callback writes a log for TensorBoard,
        # which allows you to visualize dynamic graphs of your training and test metrics
        tb = TensorBoard(log_dir='./logs',
                         histogram_freq=0,
                         write_graph=True,
                         write_images=True)
```

In [27]:
```
        # Starts training
        # autoencoder: same training(x) and target data(y)
        # validation_data: tuple (x_val, y_val) on which
        # to evaluate the loss and any model metrics at the end of each epoch.

        # History.history attribute is a record of training loss values
        # and metrics values at successive epochs.
        history = autoencoder.fit(x=train_x, y=train_x,
                                  epochs=nb_epoch,
                                  batch_size=batch_size,
                                  shuffle=True,
                                  validation_data=(test_x, test_x),
                                  verbose=0,
                                  callbacks=[cp, tb]).history
```

In [51]:
```
        autoencoder = load_model('autoencoder_fraud.keras')
```

In [30]:
```
        plt.figure(figsize=(7, 4))
        plt.plot(history['loss'], linewidth=2, label='Train')
```

```
plt.plot(history['val_loss'], linewidth=2, label='Test')
plt.legend(loc='upper right', fontsize=12)
plt.title('Model loss', fontsize=12)
plt.ylabel('Loss', fontsize=12)
plt.xlabel('Epoch', fontsize=12)
#plt.ylim(ymin=0.70,ymax=1)
plt.xticks(fontsize=10)   # X-axis ticks font size
plt.yticks(fontsize=10)   # Y-axis ticks font size
plt.show()
```



In [31]:
```
# Step 1: Use the trained autoencoder to predict (reconstruct) the test set
# This passes the input 'test_x' through the full autoencoder (encoder + decoder)
# and produces a reconstructed version of each input sample
test_x_predictions = autoencoder.predict(test_x)

# Step 2: Calculate the reconstruction error (Mean Squared Error) for each sample
# - Subtract the reconstructed inputs from the original inputs
# - Square the difference element-wise
# - Compute the mean error across all features (axis=1 means row-wise for each sample)
mse = np.mean(np.power(test_x - test_x_predictions, 2), axis=1)

# Step 3: Create a DataFrame to collect reconstruction errors and the true labels
# - 'Reconstruction_error' column stores the computed MSE for each sample
# - 'True_class' column stores the true label for each sample (0 = normal, 1 = fraud)
error_df = pd.DataFrame({
    'Reconstruction_error': mse,
    'True_class': test_y
})
```

```
# Step 4: Display descriptive statistics of the reconstruction errors and labels
# - This includes count, mean, standard deviation, min, max, and percentiles
# - Helps to understand the overall distribution of errors for both normal and fraudulent tran
error_df.describe()
```

```
1781/1781            3s 2ms/step
```

Out[31]:

	Reconstruction_error	True_class
count	56962.000000	56962.000000
mean	0.551697	0.002019
std	3.069215	0.044887
min	0.027368	0.000000
25%	0.145310	0.000000
50%	0.243181	0.000000
75%	0.413759	0.000000
max	189.518412	1.000000

Most normal transactions are reconstructed with low error (around 0.2–0.4). - A few transactions show very large reconstruction errors (up to ~188), likely corresponding to fraudulent activities. - This indicates that the autoencoder learned the structure of normal transactions well and struggles with outliers (fraud cases), which ***is exactly the intended behavior***.

To turn these reconstruction errors into actual predictions:

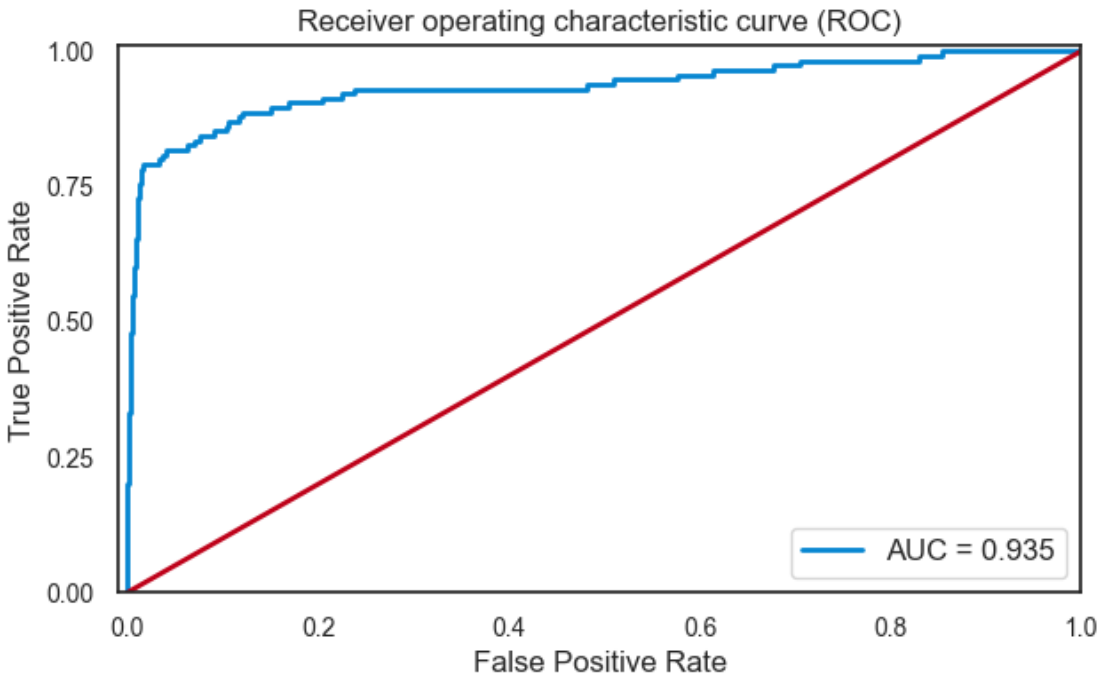
We would choose a threshold: if Reconstruction_error > threshold, classify as fraud.

Threshold choice affects precision and recall — For this reason we usually plot the Precision-Recall curve or ROC curve to tune it properly.

1.6.2 ROC Curve Check

Receiver operating characteristic curves are an expected output of most binary classifiers. Since we have an imbalanced data set they are somewhat less useful. Why? Because you can generate a pretty good-looking curve by just simply guessing everything is the normal case because there are so proportionally few cases of fraud.

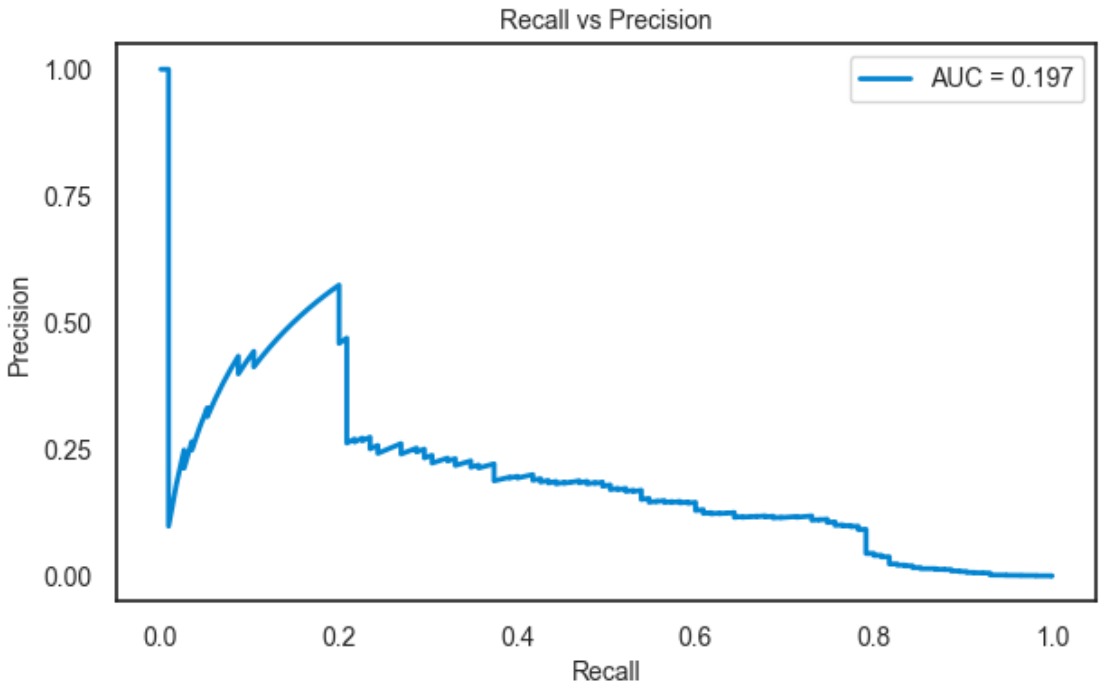
```
In [32]: false_pos_rate, true_pos_rate, thresholds = roc_curve(error_df.True_class,
                                                              error_df.Reconstruction_error)
         roc_auc = auc(false_pos_rate, true_pos_rate,)

         plt.figure(figsize=(7, 4))
         plt.plot(false_pos_rate, true_pos_rate, linewidth=2, label='AUC = %0.3f'% roc_auc)
         plt.plot([0,1],[0,1], linewidth=2)

         plt.xlim([-0.01, 1])
         plt.ylim([0, 1.01])
         plt.legend(loc='lower right', fontsize=12)
         plt.title('Receiver operating characteristic curve (ROC)', fontsize=12)
         plt.ylabel('True Positive Rate', fontsize=12)
         plt.xlabel('False Positive Rate', fontsize=12)
         plt.xticks(fontsize=10)   # X-axis ticks font size
         plt.yticks(fontsize=10)   # Y-axis ticks font size

         plt.show()
```



1.6.3 Recall vs. Precision Thresholding

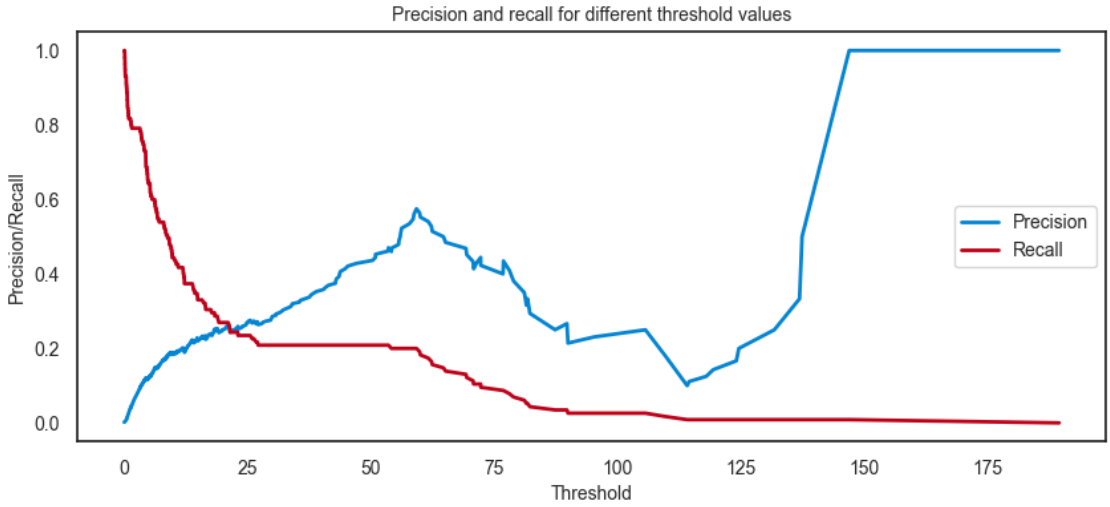
Now let's look at recall vs. precision to see the trade-off between the two.

In [33]:
```
# calculates precision/recall using reconstruction error as the decision function
# returns:
# precision_rt: Precision values such that element i is the precision of predictions with
#               score >= thresholds[i] and the last element is 1.
# recall_rt: Decreasing recall values such that element i is the recall of predictions with
#           score >= thresholds[i] and the last element is 0.
# threshold_rt = Increasing thresholds on the decision function used to compute
#                precision and recall.
precision_rt, recall_rt, threshold_rt = precision_recall_curve(error_df.True_class,
                                                               error_df.Reconstruction_error)

pr_auc = auc(recall_rt, precision_rt,)

plt.figure(figsize=(7, 4))
plt.plot(recall_rt, precision_rt, linewidth=2, label='AUC = %0.3f'% pr_auc)
plt.legend(loc='upper right', fontsize=10)
plt.title('Recall vs Precision', fontsize=10)
plt.xlabel('Recall', fontsize=10)
plt.ylabel('Precision', fontsize=10)
plt.xticks(fontsize=10)   # X-axis ticks font size
plt.yticks(fontsize=10)   # Y-axis ticks font size

plt.show()
```



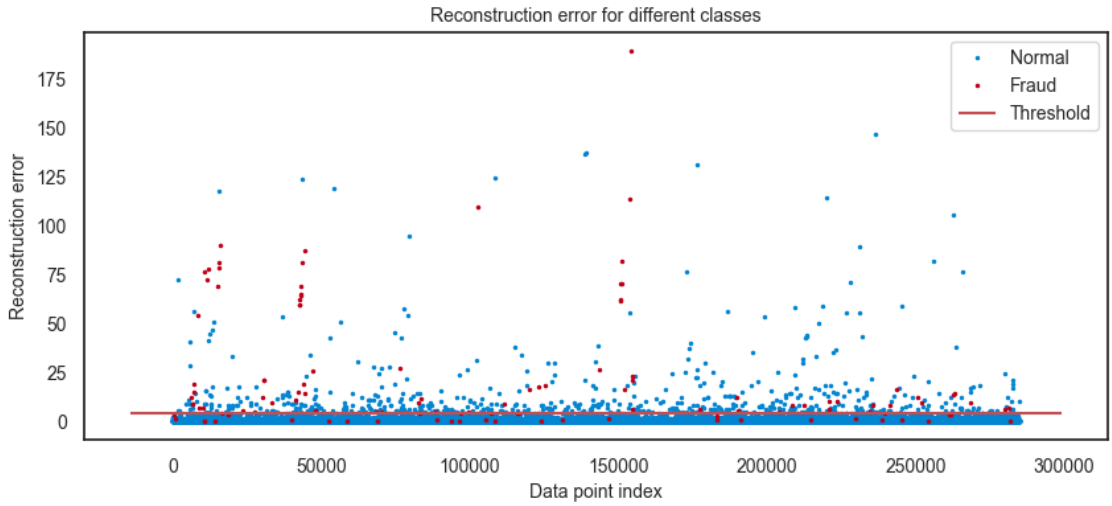
Precision and recall are the eternal tradeoff in data science, so at some point you have to draw an arbitrary line, or a threshold. Where this line will be drawn is essentially a business decision. In this case, you are trading off the cost between missing a fraudulent transaction and the cost of falsely flagging the transaction as a fraudulent even when it is not.

```
In [50]: plt.figure(figsize=(10, 4))
         plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=2)
         plt.plot(threshold_rt, recall_rt[1:]   , label="Recall"   ,linewidth=2)

         plt.title('Precision and recall for different threshold values', fontsize=10)
         plt.xlabel('Threshold', fontsize=10)
         plt.ylabel('Precision/Recall', fontsize=10)
         plt.tick_params(axis='both', labelsize=10)
         plt.legend(fontsize=10)

         plt.show()
```



Reconstruction Error vs Threshold Check

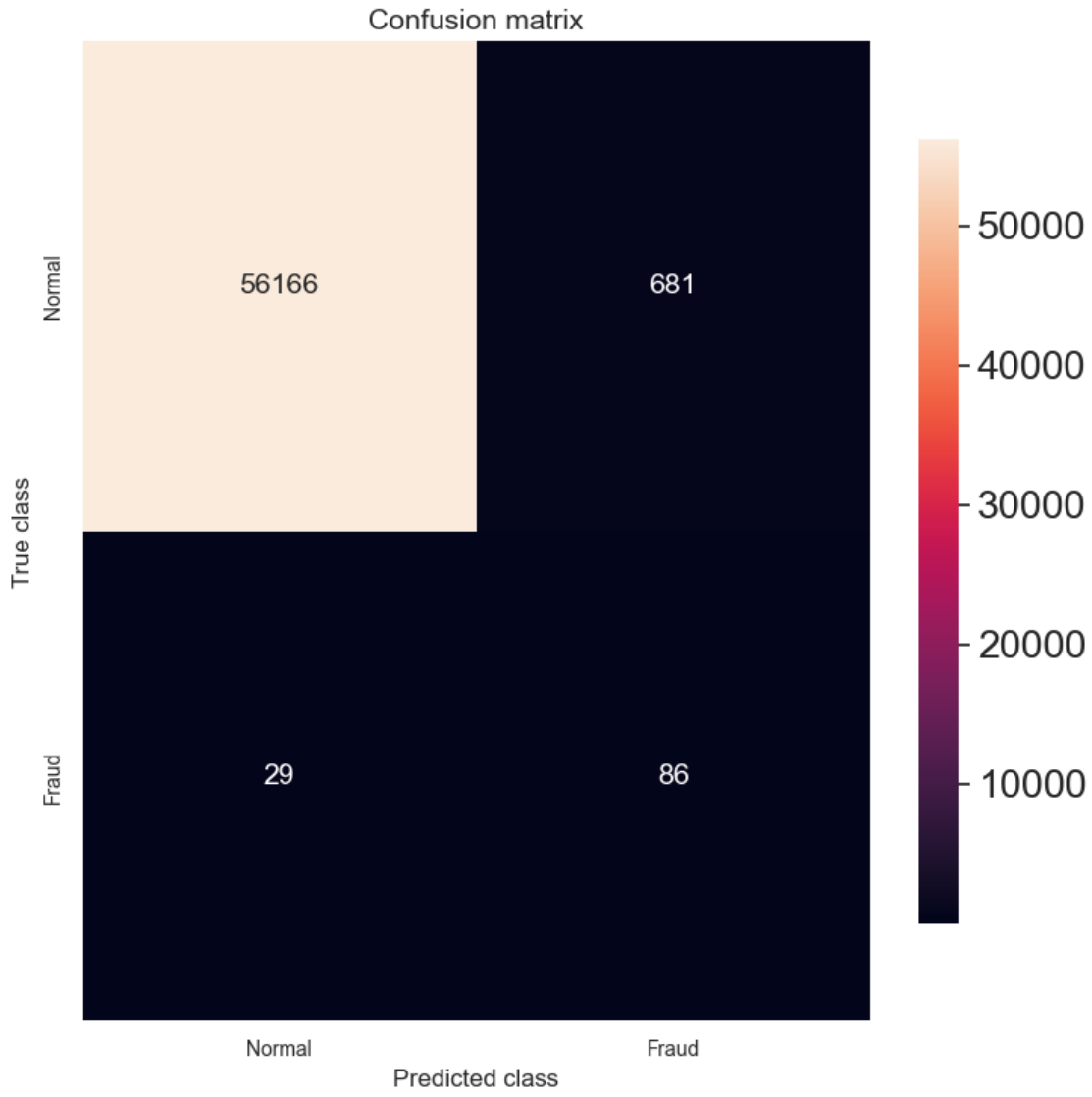
```
In [46]: threshold_fixed = 4
         groups = error_df.groupby('True_class')
         fig, ax = plt.subplots(figsize=(10, 4))

         for name, group in groups:
             ax.plot(group.index, group.Reconstruction_error, marker='o', ms=1.5, 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')
         # Set tick font size
         ax.tick_params(axis='both', labelsize=10)
         ax.legend(fontsize=10)
         plt.title("Reconstruction error for different classes", fontsize=10)
         plt.ylabel("Reconstruction error", fontsize=10)
         plt.xlabel("Data point index", fontsize=10)
         plt.show();
```



Confusion Matrix

Finally, we take a look at a traditional confusion matrix for the 20% of the data we randomly held back in the testing set. Here I really take a look at the ratio of detected fraud cases to false positives.

In [58]:
```
# As with autoencoders, it is the assumption that fraud or anomalies will suffer
# from a detectably high reconstruction error, predicting class as 1 (Fraud)
# if reconstruction error is greater than threshold
pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]
conf_matrix = confusion_matrix(error_df.True_class, pred_y)

plt.figure(figsize=(8, 8))
sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d", annot_kws
plt.title("Confusion matrix", fontsize=14)
plt.ylabel('True class', fontsize=12)
plt.xlabel('Predicted class', fontsize=12)

# Change tick label sizes
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)

plt.tight_layout()
plt.show()
```



1.7 Connection with Anomaly Detection

The method we are using — **autoencoders trained only (almost) on normal transactions** — is **a classic anomaly detection approach**.

Key idea of anomaly detection:

- **Train only on normal data**.
- **Detect deviations** (anomalies) by identifying examples that **do not fit** the normal data structure.

In this case:

- The **autoencoder** learns how normal transactions behave — meaning it knows how to compress and reconstruct them with low error.
- **Fraudulent transactions** represent **anomalies**: they differ in subtle (or not-so-subtle) ways from normal patterns.

- Thus, frauds are **badly reconstructed**, resulting in **high reconstruction errors**.

Reconstruction error acts like an anomaly score:

- **Low reconstruction error** → very likely a **normal** transaction.
- **High reconstruction error** → likely an **anomaly** (fraud).

Excellent observation — you are absolutely right to point that out!
A native speaker would phrase it something like:
In this example, we're not really training only on normal data

- **Training Data (`train_x`)**:
  It is **not filtered** to include only normal (non-fraudulent) transactions.
- Therefore, the **autoencoder is trained on a mix** of **mostly normal** transactions and a **tiny number** of **fraudulent** transactions (because frauds are so rare, only ~0.2%).
- **In practice**, this means:
  - The autoencoder mostly learns to reconstruct **normal patterns**.
  - A very few frauds are seen during training, but because they are statistically very rare and different, the autoencoder **cannot learn to reconstruct them well** anyway.
- Thus, even though the training set includes some frauds, the model behaves **almost as if trained only on normal data**.

Why does it still work?

- The **fraud samples are so rare** that they barely affect the overall learning process.
- The autoencoder mainly optimizes for the majority class (normal transactions).
- **Fraud transactions are statistical outliers**: the model finds them "strange" and reconstructs them poorly, causing **high reconstruction errors**.

1.8 Conclusions

The results show that **autoencoders can be a very effective tool for detecting fraud**, even in highly imbalanced datasets like credit card transactions.
Specifically:

- **High AUC (Area Under the ROC Curve)**:
  The reported AUC is close to **0.93**, indicating that the autoencoder can **strongly separate** fraudulent transactions from normal ones based only on reconstruction error.
- **Good Precision and Recall trade-off**:
  Despite the severe class imbalance (fraud cases being extremely rare), the autoencoder maintained **reasonable recall** (detecting a good proportion of frauds) without sacrificing too much precision (keeping false alarms relatively low).
- **Threshold tuning is critical**:
  The choice of the reconstruction error threshold directly impacts performance. Proper tuning is necessary to balance **false positives** (normal transactions incorrectly flagged) and **false negatives** (fraud cases missed).
- **Anomaly detection capability**:
  Since the autoencoder was trained only on normal transactions, it naturally **models the "normal behavior"**, making it particularly suited for **unsupervised** or **semi-supervised** fraud detection, where fraud examples are rare or even unknown during training.

- **Limitations**:
  While promising, autoencoders are **not perfect**. There is still a trade-off between catching more frauds and avoiding false positives. Moreover, they may struggle if fraudulent transactions closely mimic normal ones (i.e., are not true "outliers").

1.9 Reference and Credits

D. Surana, "***Fraud detection using Autoencoders in Keras***", Kaggle.com

In [ ]: