Credit Card Fraud Detection Report

By David Uzumaki Ore Ibikunle

15330326

15351216

Date of Submission: 17/12/18

Module: CA4010

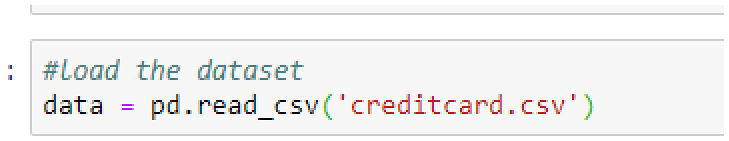
Module Coordinator: Mark Roantree

Contents

Introduction:

For our project we studied a dataset of credit cards to be able to detect fraudulent card payments. To achieve this, we used the following tools:

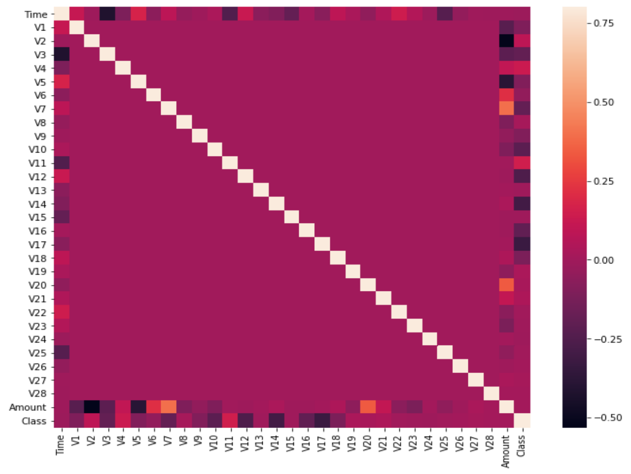
- **Pandas** – is a software library used in Python for data manipulation and analysis. It offers data structures and operations for manipulating numerical tables and time series. We used this to load the CSV file and handle the dataset.



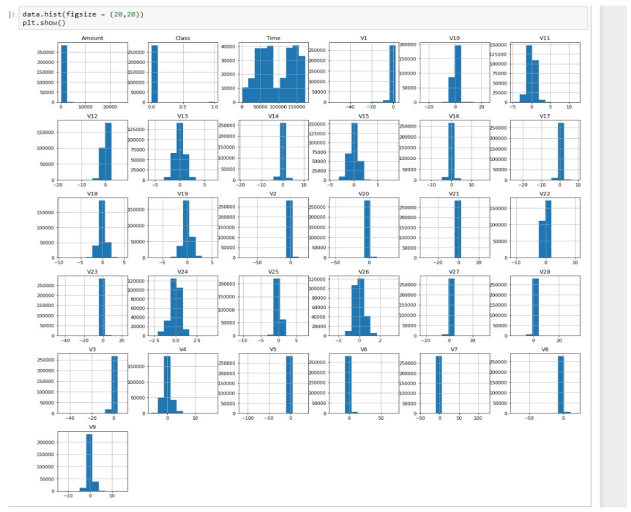
```
#load the dataset
data = pd.read_csv('creditcard.csv')
```

- **Seaborn** – is statistical data visualization used in Python. It provides a high-level interface for drawing attractive and informative statistical graphics. We used this t create the heatmap.

```
sns.heatmap(cormat, vmax = .8, square = True)
plt.show()
```

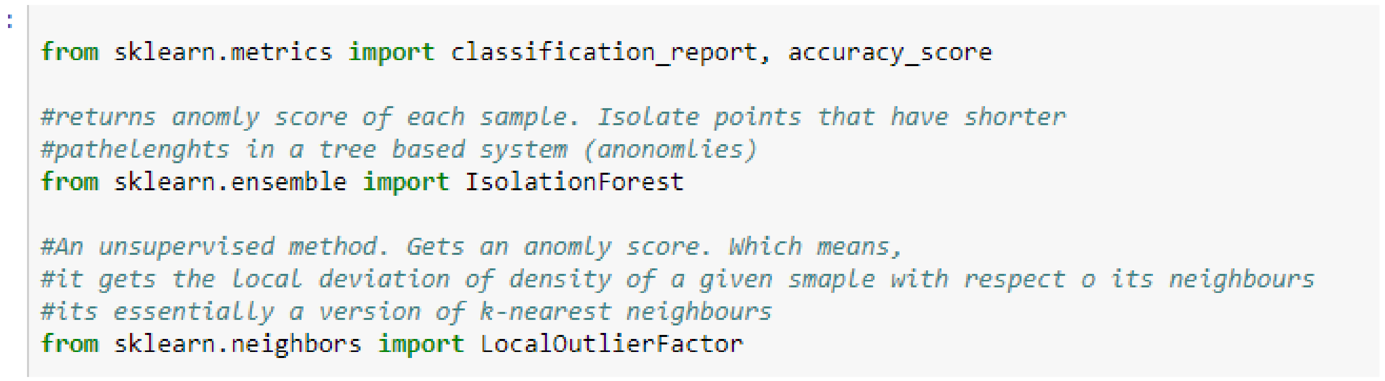


Matplotlib – is a collection of command style functions that make matplotlib work like MATLAB. We used this to graphically display our dataset and to summarise its entries.



```
data.hist(figsize = (20,20))
plt.show()
```

SKLearn – provides a range of supervised and unsupervised learning algorithms vis consistent interface in Python. Basically, it’s a machine learning tool. We used it to score our function and performance.



```
from sklearn.metrics import classification_report, accuracy_score

#returns anomly score of each sample. Isolate points that have shorter
#pathelenghts in a tree based system (anonomlies)
from sklearn.ensemble import IsolationForest

#An unsupervised method. Gets an anomly score. Which means,
#it gets the local deviation of density of a given smaple with respect o its neighbours
#its essentially a version of k-nearest neighbours
from sklearn.neighbors import LocalOutlierFactor
```

We then used Jupiter Notebook because it allowed us to record the code we were typing in and in turn easily save and document the results.

Idea and Dataset Description:

We got our dataset from Kaggle. However, upon commencing the development for the fraud detector we noticed that a lot of the data was censored via a PCA algorithm. PCA is an abbreviation for Principle Component Analysis and it is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities each of which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components. In short, it is an algorithmic way of protecting personal information. However, even though was made private the numbers generated can still be analyzed to provide meaningful context.

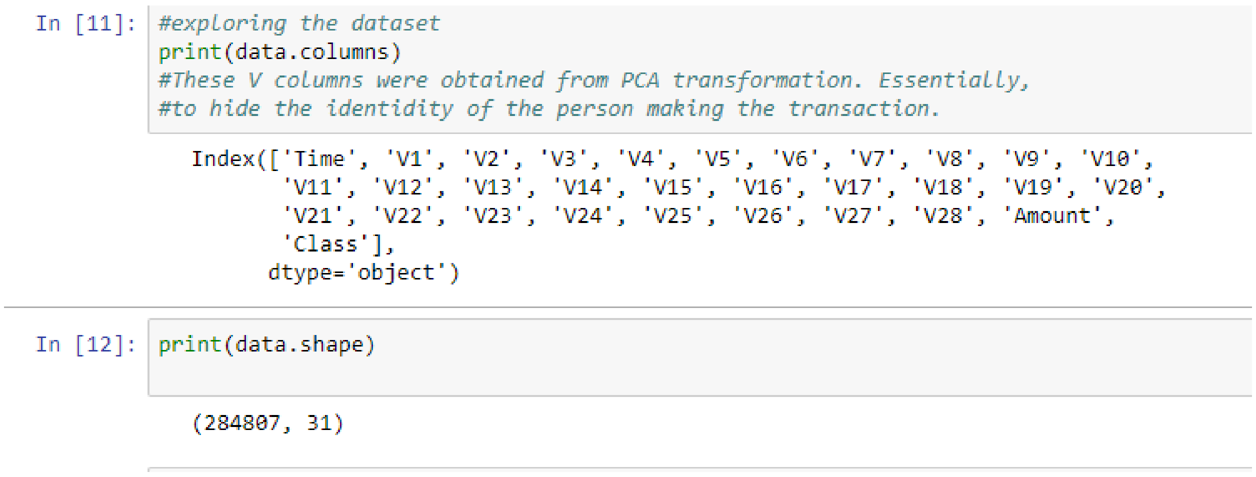
Content:

In our dataset there are transactions made by credit cards from 2013 by European Cardholders. There's a total of over 280,000 entries with 31 columns per entry.

Data Preparation & Exploration

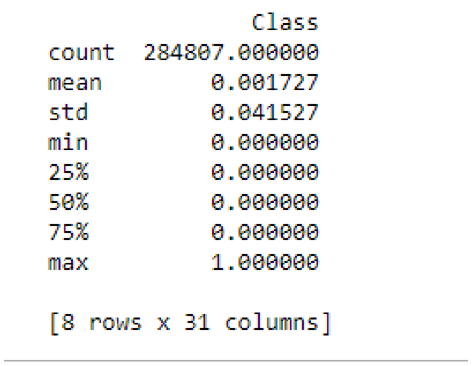
To begin and move forward to the analysis stage we had to explore and prepare the dataset to be able to derive our conclusions.

There was a total of 284807 entries and 31 columns of data which can be seen below.



```
In [11]: #exploring the dataset
         print(data.columns)
         #These V columns were obtained from PCA transformation. Essentially,
         #to hide the identidity of the person making the transaction.

            Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10',
                   'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20',
                   'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount',
                   'Class'],
                  dtype='object')

In [12]: print(data.shape)

            (284807, 31)
```

We were able to derive more accurate answers by using the entire dataset as our systems are powerful enough to handle the processing. The data set description on Kaggle informed us that most of the entries were valid payments. This meant there was a bias in the dataset we had to amend. We first encountered this when we pulled the mean from the dataset:



```
               Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000

[8 rows x 31 columns]
```

As you can see the mean is near 0, indicating there's an incredibly large bias to deal with.

Bias handling (Outlier fraction)

How we handled the bias to prepare for our algorithm is we put a higher emphasis on the bias. We started by splitting the valid and fraudulent payments.

```
#Now we want to determine the number of fraud cases
fraud = data[data['Class'] == 1]
valid = data[data['Class']== 0]
```

We then put emphasis on the fraudulent payments be getting the ratio of the fraud to valid transactions.

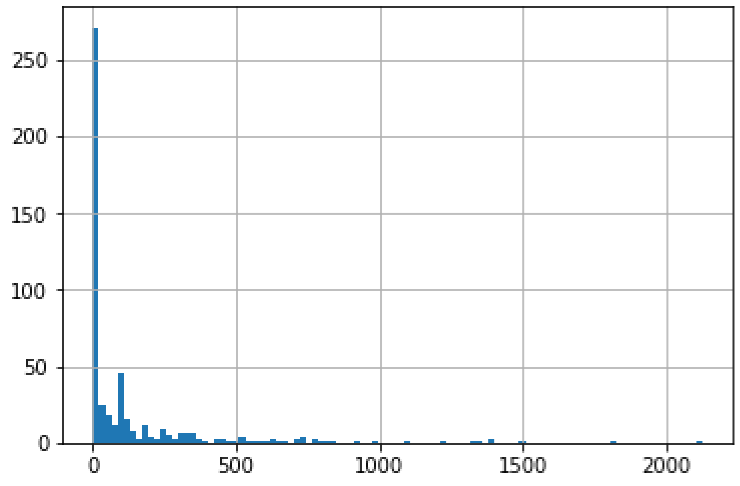
```
#It's important to know an outlier fraction as we don't want to over predict
#or under predict anything.
outlier_fraction = len(fraud) / float(len(valid))
print(outlier_fraction)
```

With that out of the way we then explored the data set further to better understand the dataset and draw some conclusions.

Findings

After handling the bias problem and before tackling any algorithms it was important for us to understand the dataset more. We started this by diving deeper into the fraudulent transactions. Below we plotted the amount column from the only the fraudulent payments.

```
fraud_amount = fraud["Amount"].hist(bins=100);
```



Fraud Amount Transaction Findings

Something that's strange is the median is lower but the mean is higher for the fraud cases. What this suggests is that there are criminals who are high valued and some that focus on withdrawals "below the radar" to avoid detection.

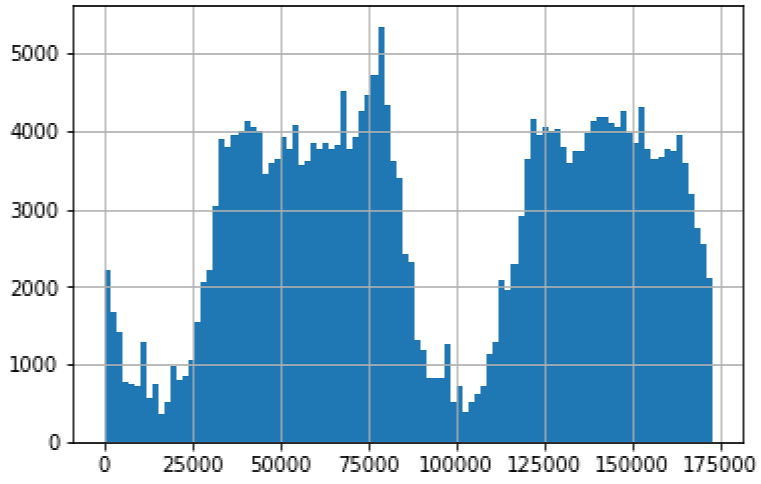
Looking deeper into the fraudulent cases. We can see that the minimum fraudulent transaction was 0 euro and others as low as 68%. This most likely tells us some people test out cards before they spend it on something higher. Because as we can see, the highest fraudulent transaction was at 2125 euro.

Time of Transactions Findings

We can see there's a major drop of transactions in the night compared to the day which is understandable as people sleep. However, it begs the question as to; are the criminal's day or night creatures?

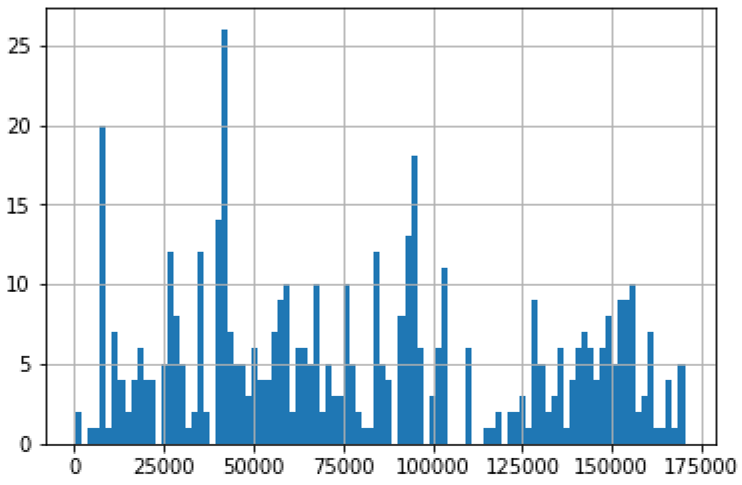
```
data_time = data["Time"].hist(bins=100)
print(data_time)
```

```
AxesSubplot(0.125,0.125;0.775x0.755)
```



And we can see, there are night creatures (at least in this data set):

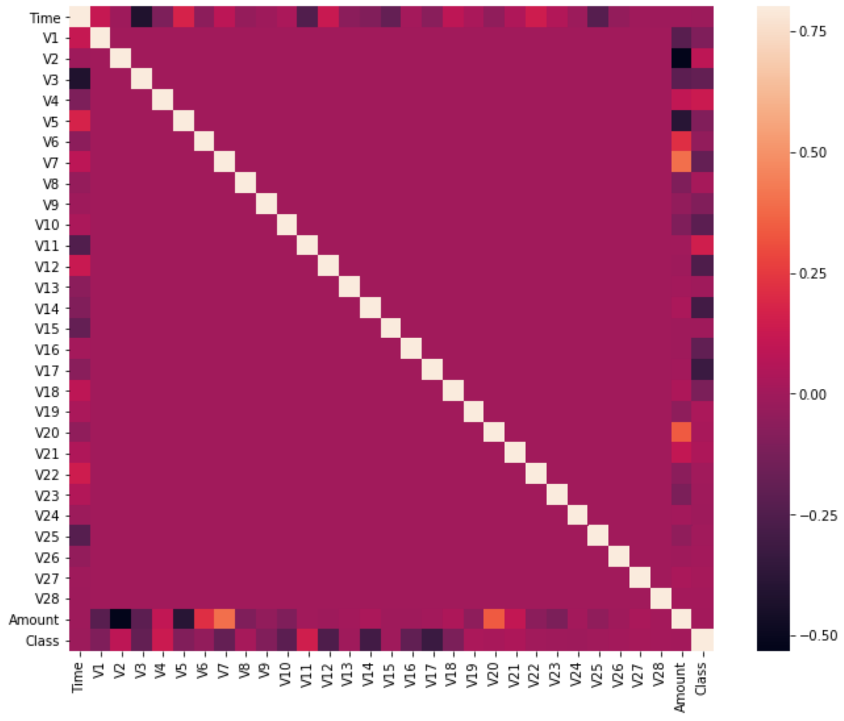
```
]: fraud_time = fraud["Time"].hist(bins=100);
```



Correlation matrix and heatmap findings

The correlation matrix allowed us to define what's important for over classification. So, what do we need and not need?

```
corrmat = data.corr()
fig = plt.figure(figsize = (12,9))

sns.heatmap(corrmat, vmax = .8, square = True)
plt.show()
```



Due to the PCA transformations it becomes difficult to conclude certain results. You can see from the pink/reddish color in the middle that we have. A lot of values are close to 0 so there aren't strong relationships between our **V1 - V28** telling us that it doesn't matter about the location or personal details of the individual making the payment in order to detect a fraudulent payment.

And of course, that is the case, because this dataset has already been processed and we know what's fraudulent and what is not.

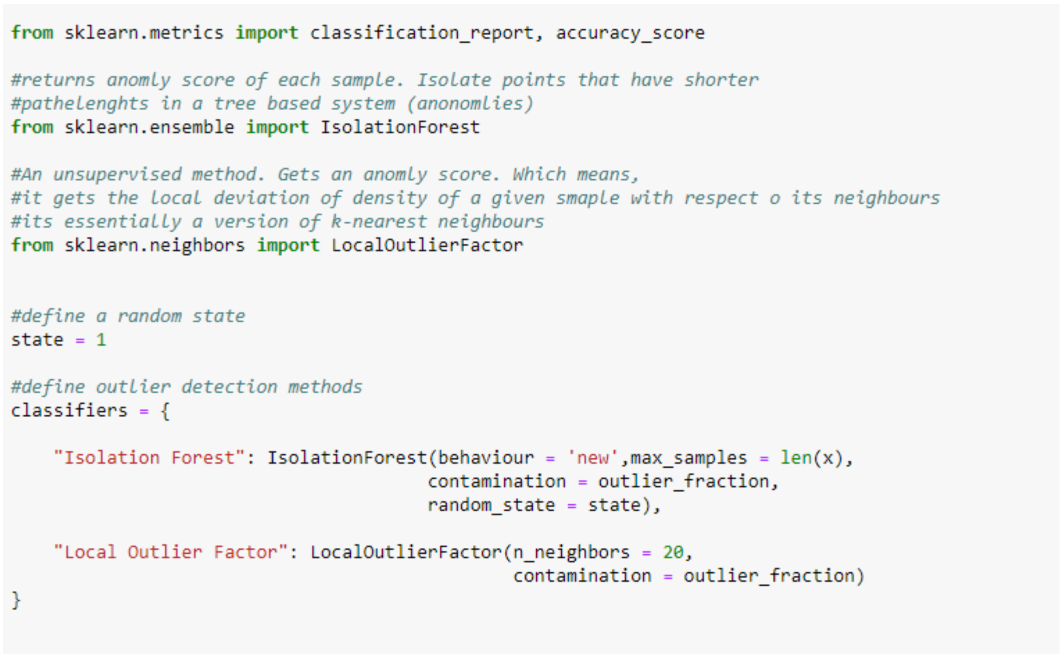
Example: assuming **v17** on the left side is the address of an individual and **v14** on the bottom is, let's say a credit number. It doesn't affect each other in order to determine if a payment is fraudulent. Only because the data has been pre-processed. These however could be other values as **v17** is inversely affecting another **v** value. However, due to privacy protection it becomes hard for us to get a true meaning from what it means. Fortunately, we now know what we need to cut to eliminate over classification.

Algorithm Description

We used 2 algorithms in order to test the performance of our system. A classification report and a variation of k-nearest neighbors. We had to fit the model afterwards to prepare our machine learning model.

Prior that we split our "class" column from the rest of the dataset as that's what only matters. (over classification solution).

As you can see below what we are doing is setting up the classifiers for our functions and taking into account the outlier fraction. Luckily the heavy lifting is done via the sklearn module.



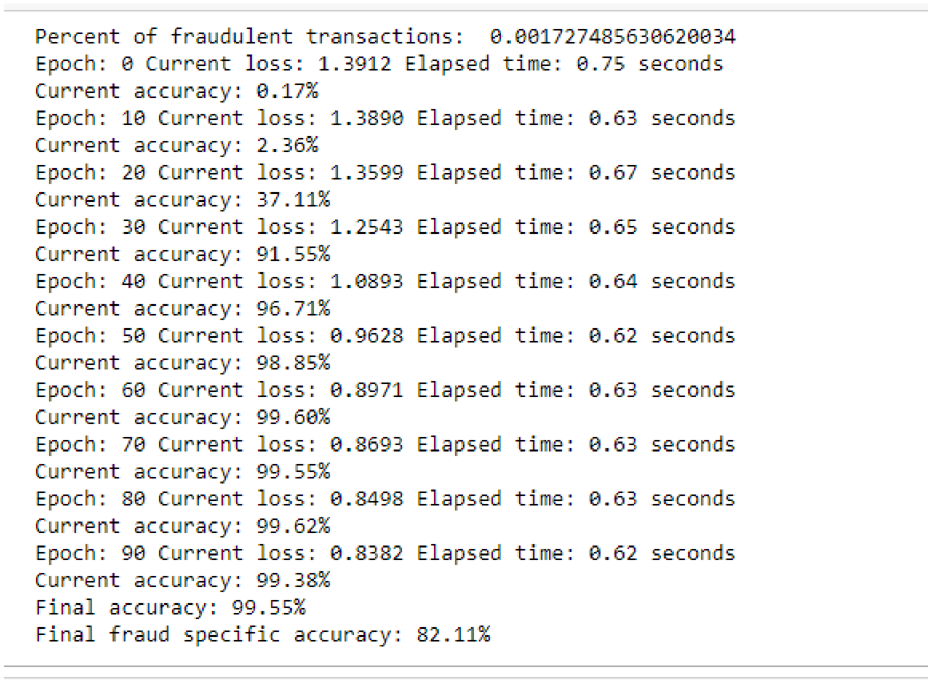
```python
from sklearn.metrics import classification_report, accuracy_score

#returns anomly score of each sample. Isolate points that have shorter
#pathelenghts in a tree based system (anonomlies)
from sklearn.ensemble import IsolationForest

#An unsupervised method. Gets an anomly score. Which means,
#it gets the local deviation of density of a given smaple with respect o its neighbours
#its essentially a version of k-nearest neighbours
from sklearn.neighbors import LocalOutlierFactor


#define a random state
state = 1

#define outlier detection methods
classifiers = {

    "Isolation Forest": IsolationForest(behaviour = 'new',max_samples = len(x),
                                       contamination = outlier_fraction,
                                       random_state = state),

    "Local Outlier Factor": LocalOutlierFactor(n_neighbors = 20,
                                               contamination = outlier_fraction)
}
```

We then had to fit the model and also reshape the predication output as the original output is 1 and -1. So, we assigned 1 to valid payments and -1 to fraud.

Results:

We can see that the program can 100% accurately predict a valid payment it, only 35% of the time can detect a fraudulent payment. What we find here is (even with the limited data we had) is that a single pass analysis isn't good enough to predict a heavily biased dataset. But a multiple epoch system should by putting if we, emphasize the fraud bias at the start so our program can essentially, predict fraudulent transactions better.

However, with some adjustment we made our program run generations at a time. Below is the final accuracy:



```
Percent of fraudulent transactions:  0.001727485630620034
Epoch: 0 Current loss: 1.3912 Elapsed time: 0.75 seconds
Current accuracy: 0.17%
Epoch: 10 Current loss: 1.3890 Elapsed time: 0.63 seconds
Current accuracy: 2.36%
Epoch: 20 Current loss: 1.3599 Elapsed time: 0.67 seconds
Current accuracy: 37.11%
Epoch: 30 Current loss: 1.2543 Elapsed time: 0.65 seconds
Current accuracy: 91.55%
Epoch: 40 Current loss: 1.0893 Elapsed time: 0.64 seconds
Current accuracy: 96.71%
Epoch: 50 Current loss: 0.9628 Elapsed time: 0.62 seconds
Current accuracy: 98.85%
Epoch: 60 Current loss: 0.8971 Elapsed time: 0.63 seconds
Current accuracy: 99.60%
Epoch: 70 Current loss: 0.8693 Elapsed time: 0.63 seconds
Current accuracy: 99.55%
Epoch: 80 Current loss: 0.8498 Elapsed time: 0.63 seconds
Current accuracy: 99.62%
Epoch: 90 Current loss: 0.8382 Elapsed time: 0.62 seconds
Current accuracy: 99.38%
Final accuracy: 99.55%
Final fraud specific accuracy: 82.11%
```

References:

https://seaborn.pydata.org/

https://pandas.pydata.org/

https://matplotlib.org/users/pyplot_tutorial.html

https://machinelearningmastery.com/a-gentle-introduction-to-scikit-learn-a-python-machine-learning-library/

https://scikit-learn.org/

https://www.kaggle.com/samkirkiles/credit-card-fraud/data