Fraud

August 29, 2022

1 Fraud detection with machine learning

We are going to analyze the fraud detection bank dataset (https://www.kaggle.com/code/dcw8161/fraud-detection-bank-dataset/data), and see if we can make a model that can reliably detect fraud. We start be importing the data

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix

df = pd.read_csv("fraud_detection_bank_dataset.csv")



print(df.head())
```

```
   Unnamed: 0  col_0  col_1  col_2  col_3  col_4  col_5  col_6  col_7  col_8  \
0           0      9   1354      0     18      0      1      7      9      0
1           1      0    239      0      1      0      1      0      0      0
2           2      0    260      0      4      0      3      6      0      0
3           3     17    682      0      1      0      0      8     17      0
4           4      1    540      0      2      0      1      7      1      0

   …  col_103  col_104  col_105  col_106  col_107  col_108  col_109  \
0  …        0        0        0        1        1        0        0
1  …        0        1        0        0        0        0        0
2  …        0        0        0        1        1        0        0
3  …        0        1        0        1        1        0        0
4  …        0        0        0        1        1        0        0

   col_110  col_111  targets
0        0       49        1
1        0       55        1
2        0       56        1
3        0       65        1
4        0      175        1
```

```
[5 rows x 114 columns]
```

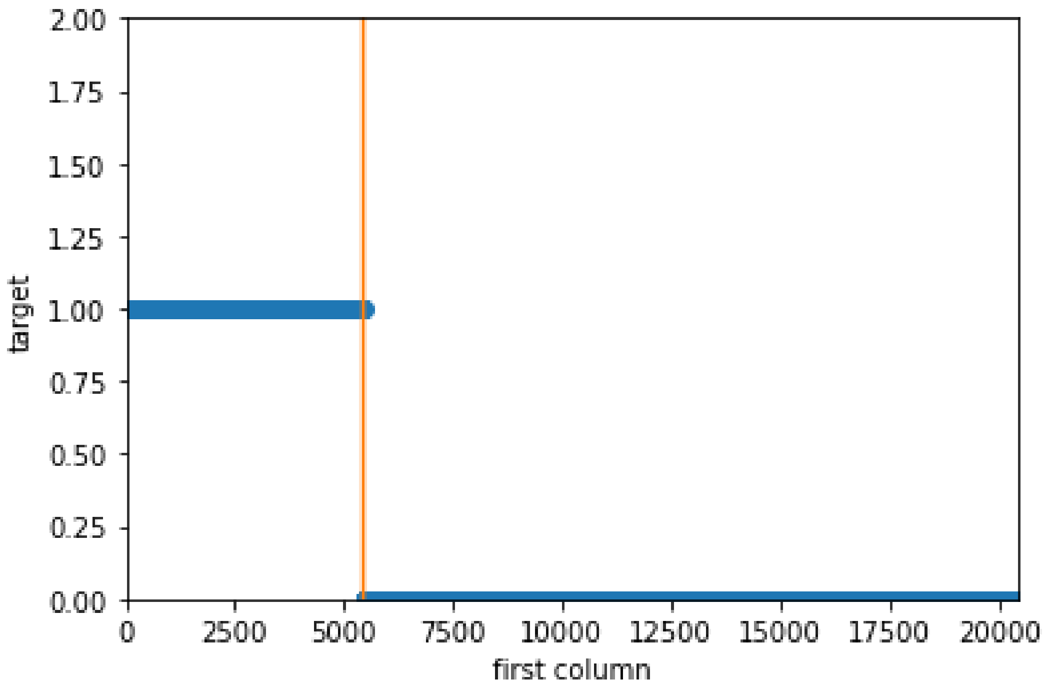
It is a bit unfortunate that we actually don't know what values the columns represent, as this is not explicitly explained by the columns or on Kaggle. But we soldier on. We want to drop the feature called Unnamed: 0 because it just counts the numer of rows. The rows are also arranged such that the n first rows (let's say the total number of rows are m) are targets with value $v_i = 1, \quad i \leq n$, and the remaining values are $v_i = 0, \quad n < i \leq m$. This makes it so that the first column can perfectly predict the targets. I will demonstrate this, but first let's seperate the targets from the main table

```
[2]: targets = df["targets"]
     first_column = df["Unnamed: 0"]
     df.drop("targets", inplace = True, axis = 1)
     df.drop('Unnamed: 0', inplace = True, axis = 1)
     X = pd.DataFrame.to_numpy(df)
     Y = np.asarray([val for val in targets])
```

```
[3]: dividing_line = np.zeros(len(targets))
     index = 0

     for element in targets:
         if element != 1:
             print(index)
             dividing_line[:] = index
             break
         else:
             index += 1

     plt.plot(first_column, targets, "o")
     plt.plot(dividing_line, first_column)
     plt.axis([0,len(targets),0,2])
     plt.xlabel("first column")
     plt.ylabel("target")
```

```
5438
```

```
[3]: Text(0, 0.5, 'target')
```



As we see, we now have a classification model with 100% accuracy, that can be perfectly described by the following function

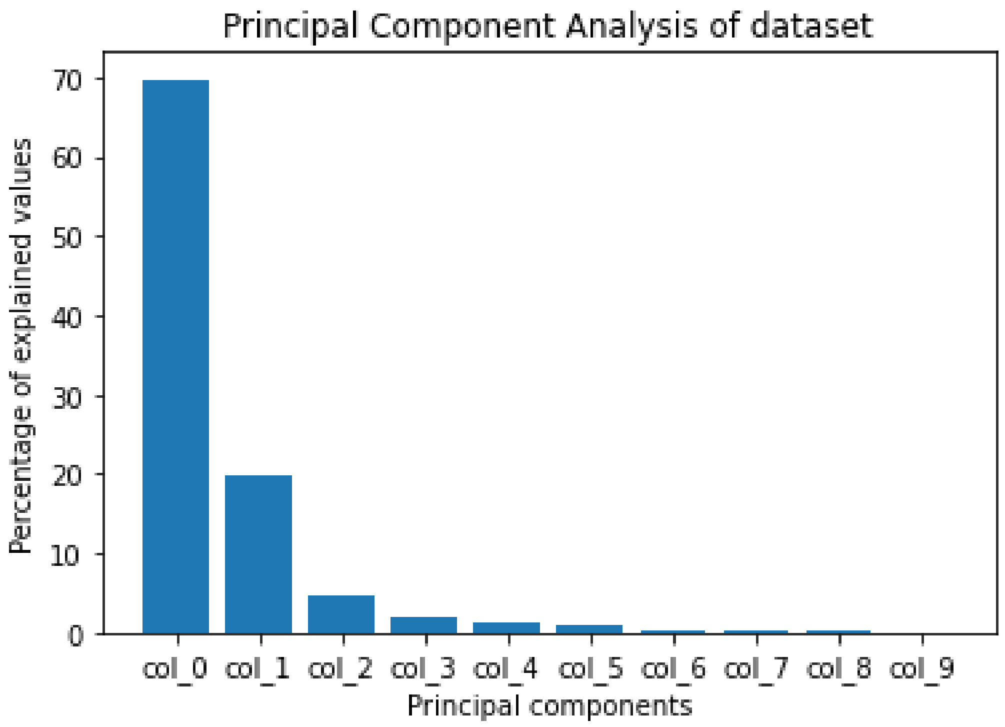
$$y(x) = \begin{cases} 1, & 0 < x < 5438 \\ 0, & 5438 \leq x \leq \text{len(targets)} \end{cases}$$

This is obviously just silly, and is purely an artifact of how the data is structured. This is why we delete the column "Unnamed: 0"

We now have a feature matrix X with 112 columns. It might be a good idea to check whether we need all of these columns, or if we can get away with using fewer. We do this with principal component analysis (PCA)

[4]:
```python
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA


scaler = StandardScaler()
X_scaled = scaler.fit(X.T).transform(X.T)
#Doing PCA
pca = PCA()
pca.fit(X_scaled.T)
pca_data = pca.transform(X_scaled.T)
#Rounding off the percentages
per_val = np.round(pca.explained_variance_ratio_*100, decimals = 1)
#Labels of PC for plotting
```

```
labels = df.keys()
labels2 = np.arange(1,len(labels)+1)



#We plot the 10 highest contributors
plt.bar(x=range(1,11), height = per_val[0:10], tick_label = labels[0:10])
plt.title("Principal Component Analysis of dataset")
plt.ylabel("Percentage of explained values")
plt.xlabel("Principal components")
plt.show()
```



We can see that most of the variance comes from the 8 first columns. We can try to reduce the dataset to the 8 first columns, and see if there is a training time/ accuracy trade-off here

[5]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from time import perf_counter

#Create a dataset with only the 9 highest contributors
X_reduced = X[:,0:8]
```

```
X_train, X_test, X_train_reduced, X_test_reduced, y_train, y_test =␣
 ↪train_test_split(X, X_reduced, Y, test_size = 0.2)



t_train_start = perf_counter()

model_full = AdaBoostClassifier()
model_full.fit(X_train, y_train)
predicted_full = model_full.predict(X_test)

t_train_stop = perf_counter()
t_total_full = t_train_stop-t_train_start

t_train_start = perf_counter()

model_reduced = AdaBoostClassifier()
model_reduced.fit(X_train_reduced, y_train)
predicted_reduced = model_reduced.predict(X_test_reduced)

t_train_stop = perf_counter()
t_total_reduced = t_train_stop-t_train_start

S_f = accuracy_score(predicted_full, y_test)

S_r = accuracy_score(predicted_reduced, y_test)

print("Time to train on full dataset: %1.2f s" %(t_total_full))
print("Time to train on reduced dataset: %1.2f s" %(t_total_reduced))
print("Training accuracy on full dataset: %1.2f" %(S_f))
print("Training accuracy on reduced dataset: %1.2f" %(S_r))
```

```
Time to train on full dataset: 1.79 s
Time to train on reduced dataset: 0.46 s
Training accuracy on full dataset: 0.91
Training accuracy on reduced dataset: 0.85
```

We can see that we save a significant amount of time, although we do sacrifice a lot of accuracy. We move along with the reduced dataset, and see if we can make some decent predictions. The reason for this is that we can save a whole lot of time in training, if we can find a good enough model. We start with cross validation

[6]:
```
from sklearn.model_selection import cross_val_score
from sklearn.utils import shuffle
clf = AdaBoostClassifier()
#shuffle(X_train_reduced, y_train)
```

```
scores = cross_val_score(clf, X_train_reduced, y_train, cv = 5, scoring =␣
  ↪"accuracy")
scores
```

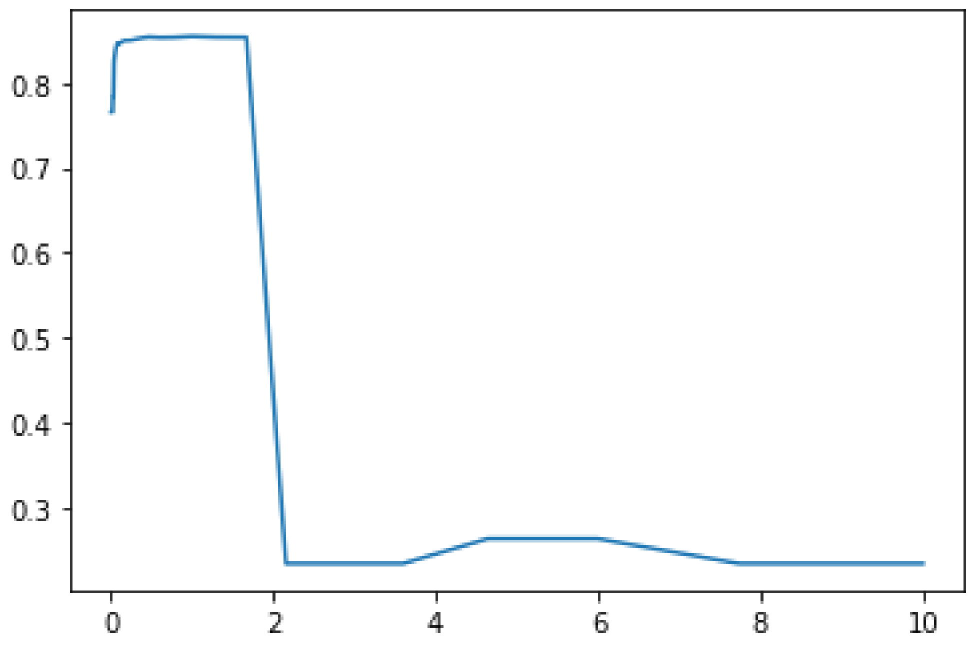
```
[6]: array([0.86381679, 0.85312977, 0.83847328, 0.85129771, 0.85247404])
```

That did not really improve anything. We can also do an (not really) exhaustive grid search, but before that we can check if the learning rate is of the correct order of magnitude. This is a largely useless excersise (in this case), but let's do it anyway

```
[7]: #Use logspace when checking order of magnitude
a = np.logspace(-10, 1, 100)
accs2 = np.zeros(len(a))

i = 0
for lr in a:
    model_reduced = AdaBoostClassifier(n_estimators = 40, learning_rate = lr)
    model_reduced.fit(X_train_reduced, y_train)
    pred_ada = model_reduced.predict(X_test_reduced)
    accs2[i] = accuracy_score(pred_ada, y_test)
    i += 1
```

```
[8]: plt.plot(a, accs2)
print(a[np.argmax(accs2)])
```

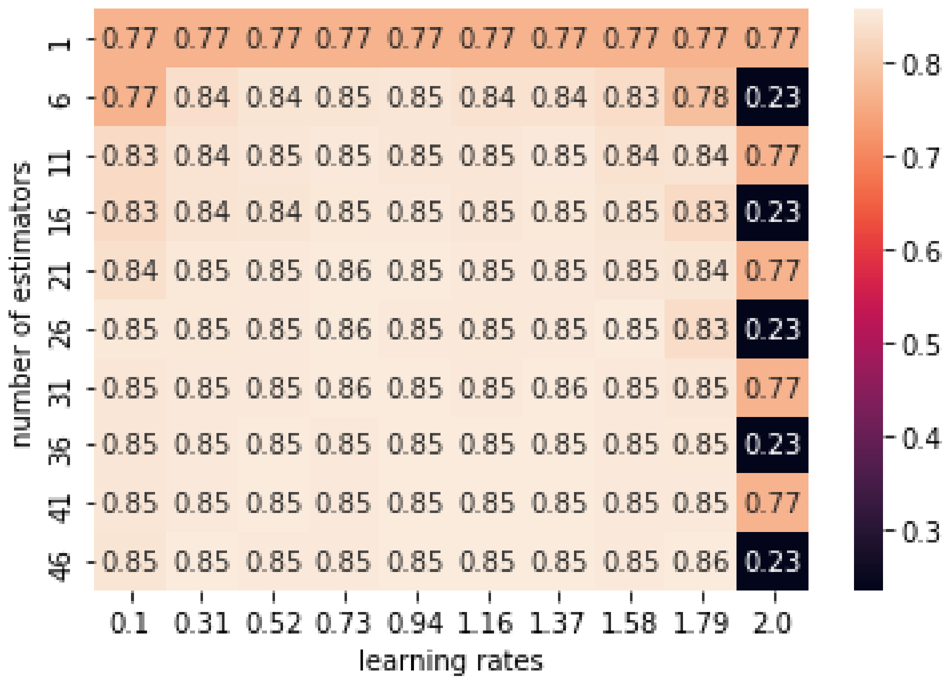
```
1.0
```



We see that a sensible learning rate γ will roughly be on the interval $\gamma \in [0.1, 1.9]$, so we use that. The number of estimators we let be between 1 and 50

```
[9]: learning_rates = np.linspace(0.1, 2, 10)
     estimators = np.arange(1,50, 5)

     accs = np.zeros((len(estimators), len(learning_rates)))

     i = 0
     for est in estimators:
         j = 0
         for lr in learning_rates:
             model_reduced = AdaBoostClassifier(n_estimators = est, learning_rate =␣
       ↪lr)
             model_reduced.fit(X_train_reduced, y_train)
             pred_ada_2 = model_reduced.predict(X_test_reduced)
             accs[i, j] = accuracy_score(pred_ada_2, y_test)
             j += 1
         i += 1
```

```
[10]: import seaborn as sb

      s = sb.heatmap(accs, annot = True, xticklabels = np.round(learning_rates,␣
       ↪decimals = 2), yticklabels = estimators)
      s.set(xlabel = "learning rates", ylabel = "number of estimators")

      pred2 = model_reduced.predict(X_train_reduced)
      acc2 = accuracy_score(pred2, y_train)
```



	0.1	0.31	0.52	0.73	0.94	1.16	1.37	1.58	1.79	2.0
1	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77
6	0.77	0.84	0.84	0.85	0.85	0.84	0.84	0.83	0.78	0.23
11	0.83	0.84	0.85	0.85	0.85	0.85	0.85	0.84	0.84	0.77
16	0.83	0.84	0.84	0.85	0.85	0.85	0.85	0.85	0.83	0.23
21	0.84	0.85	0.85	0.86	0.85	0.85	0.85	0.85	0.84	0.77
26	0.85	0.85	0.85	0.86	0.85	0.85	0.85	0.85	0.83	0.23
31	0.85	0.85	0.85	0.86	0.85	0.85	0.86	0.85	0.85	0.77
36	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.23
41	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.77
46	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.86	0.23

A more exhaustive search proved to be unfruitful. We might have to try other methods if we want to achieve a higher accuracy.

[11]:
```
boost = GradientBoostingClassifier()
boost.fit(X_train_reduced, y_train)
pred_boost = boost.predict(X_test_reduced)
acc_boost = accuracy_score(pred_boost, y_test)
```

[12]:
```
print(acc_boost)
```

```
0.8600390815828041
```

Without tweaking any parameters, we get a model equal to that of the AdaBoost. We might be able to improve the gradient boosted model with some parameter tuning, but I'm not going to do that.

We have seen that the Adaboost method is not (in this case) particularly sensitive to learning rate and number of estimators, so it probably won't help us greatly trying to adjust these hyper-parameters. We could take a look at the base_estimator parameter (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html), and create a custom decision tree, but it probably won't increase accuracy significantly. Let me demonstrate

[13]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.experimental import enable_halving_search_cv

base = DecisionTreeClassifier(max_depth = 6)
```

```
model_deep = AdaBoostClassifier(base_estimator = base, n_estimators = 50,␣
  ↪learning_rate = 1)
model_deep.fit(X_train_reduced, y_train)
pred_ada_base = model_deep.predict(X_test_reduced)

accuracy_score(pred_ada_base,y_test)
```

[13]:
```
0.8326819736199316
```

Still no improvement. We could try using cross-validation, grid-search, or another method of parameter tuning, bit I think it's time we changed pace. Let's see if a basic neural network can help us crank that accuracy (and Soulja Boy)

[14]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt

#For one-hot encoding the targets
def to_categorical_numpy(integer_vector):
    n_inputs = len(integer_vector)
    n_categories = np.max(integer_vector) + 1
    onehot_vector = np.zeros((n_inputs, n_categories))
    onehot_vector[range(n_inputs), integer_vector] = 1

    return onehot_vector
```

After some testing and tuning, I found the following architecture, optimizer, loss function, etc, to be working good (not optimal, but better than everything else i tried)

[22]:
```
y_onehot = to_categorical_numpy(Y)
X_train, X_test, y_train_onehot, y_test_onehot, y_train, y_test =␣
  ↪train_test_split(X_reduced, y_onehot, Y, test_size = 0.2 )

in_size = len(X_train[0])
out_size = len(y_train_onehot[0])

model = Sequential()
model.add(Dense(units = 512, activation = "sigmoid", input_dim = in_size))
model.add(Dense(units = out_size, activation = "softmax"))
model.compile(loss = "BinaryCrossentropy", optimizer = "Adamax", metrics =␣
  ↪"accuracy")

#KLDivergence
#LogCosh

model.fit(X_train, y_train_onehot, epochs = 100, batch_size = 128)
```

```
Epoch 1/100
128/128 [==============================] - 1s 2ms/step - loss: 0.5302 -
accuracy: 0.7352
Epoch 2/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4806 -
accuracy: 0.7784
Epoch 3/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4429 -
accuracy: 0.8117
Epoch 4/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4223 -
accuracy: 0.8234
Epoch 5/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4123 -
accuracy: 0.8286
Epoch 6/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4060 -
accuracy: 0.8309
Epoch 7/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4030 -
accuracy: 0.8320
Epoch 8/100
128/128 [==============================] - 0s 2ms/step - loss: 0.4011 -
accuracy: 0.8325
Epoch 9/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3983 -
accuracy: 0.8347
Epoch 10/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3969 -
accuracy: 0.8352
Epoch 11/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3955 -
accuracy: 0.8336
Epoch 12/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3939 -
accuracy: 0.8351
Epoch 13/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3921 -
accuracy: 0.8372
Epoch 14/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3910 -
accuracy: 0.8364
Epoch 15/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3892 -
accuracy: 0.8361
Epoch 16/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3866 -
accuracy: 0.8360
```

```
Epoch 17/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3857 -
accuracy: 0.8386
Epoch 18/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3852 -
accuracy: 0.8372
Epoch 19/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3832 -
accuracy: 0.8397
Epoch 20/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3824 -
accuracy: 0.8389
Epoch 21/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3812 -
accuracy: 0.8391
Epoch 22/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3799 -
accuracy: 0.8390
Epoch 23/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3789 -
accuracy: 0.8394
Epoch 24/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3776 -
accuracy: 0.8412
Epoch 25/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3771 -
accuracy: 0.8391
Epoch 26/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3747 -
accuracy: 0.8405
Epoch 27/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3737 -
accuracy: 0.8410
Epoch 28/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3726 -
accuracy: 0.8416
Epoch 29/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3713 -
accuracy: 0.8440
Epoch 30/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3698 -
accuracy: 0.8428
Epoch 31/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3695 -
accuracy: 0.8419
Epoch 32/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3685 -
accuracy: 0.8434
```

```
Epoch 33/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3676 -
accuracy: 0.8426
Epoch 34/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3659 -
accuracy: 0.8438
Epoch 35/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3658 -
accuracy: 0.8431
Epoch 36/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3643 -
accuracy: 0.8450
Epoch 37/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3638 -
accuracy: 0.8455
Epoch 38/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3631 -
accuracy: 0.8457
Epoch 39/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3628 -
accuracy: 0.8470
Epoch 40/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3619 -
accuracy: 0.8463
Epoch 41/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3618 -
accuracy: 0.8459
Epoch 42/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3608 -
accuracy: 0.8454
Epoch 43/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3610 -
accuracy: 0.8455
Epoch 44/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3592 -
accuracy: 0.8470
Epoch 45/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3599 -
accuracy: 0.8482
Epoch 46/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3588 -
accuracy: 0.8463
Epoch 47/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3586 -
accuracy: 0.8470
Epoch 48/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3581 -
accuracy: 0.8470
```

```
Epoch 49/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3577 -
accuracy: 0.8480
Epoch 50/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3571 -
accuracy: 0.8477
Epoch 51/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3561 -
accuracy: 0.8482
Epoch 52/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3563 -
accuracy: 0.8480
Epoch 53/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3558 -
accuracy: 0.8474
Epoch 54/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3552 -
accuracy: 0.8493
Epoch 55/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3549 -
accuracy: 0.8484
Epoch 56/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3560 -
accuracy: 0.8477
Epoch 57/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3544 -
accuracy: 0.8498
Epoch 58/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3547 -
accuracy: 0.8486
Epoch 59/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3539 -
accuracy: 0.8498
Epoch 60/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3531 -
accuracy: 0.8488
Epoch 61/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3535 -
accuracy: 0.8492
Epoch 62/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3527 -
accuracy: 0.8504
Epoch 63/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3524 -
accuracy: 0.8488
Epoch 64/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3525 -
accuracy: 0.8495
```

```
Epoch 65/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3519 -
accuracy: 0.8495
Epoch 66/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3519 -
accuracy: 0.8503
Epoch 67/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3530 -
accuracy: 0.8488
Epoch 68/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3519 -
accuracy: 0.8495
Epoch 69/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3510 -
accuracy: 0.8503
Epoch 70/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3503 -
accuracy: 0.8504
Epoch 71/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3507 -
accuracy: 0.8506
Epoch 72/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3493 -
accuracy: 0.8513
Epoch 73/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3498 -
accuracy: 0.8504
Epoch 74/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3500 -
accuracy: 0.8485
Epoch 75/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3500 -
accuracy: 0.8497
Epoch 76/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3495 -
accuracy: 0.8500
Epoch 77/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3487 -
accuracy: 0.8495
Epoch 78/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3482 -
accuracy: 0.8512
Epoch 79/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3491 -
accuracy: 0.8507
Epoch 80/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3482 -
accuracy: 0.8494
```

```
Epoch 81/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3477 -
accuracy: 0.8512
Epoch 82/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3476 -
accuracy: 0.8492
Epoch 83/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3477 -
accuracy: 0.8505
Epoch 84/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3475 -
accuracy: 0.8507
Epoch 85/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3470 -
accuracy: 0.8499
Epoch 86/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3470 -
accuracy: 0.8507
Epoch 87/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3469 -
accuracy: 0.8512
Epoch 88/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3468 -
accuracy: 0.8506
Epoch 89/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3461 -
accuracy: 0.8513
Epoch 90/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3459 -
accuracy: 0.8516
Epoch 91/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3459 -
accuracy: 0.8520
Epoch 92/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3467 -
accuracy: 0.8512
Epoch 93/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3460 -
accuracy: 0.8510
Epoch 94/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3463 -
accuracy: 0.8508
Epoch 95/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3448 -
accuracy: 0.8507
Epoch 96/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3452 -
accuracy: 0.8515
```

```
Epoch 97/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3463 -
accuracy: 0.8506
Epoch 98/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3448 -
accuracy: 0.8513
Epoch 99/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3448 -
accuracy: 0.8512
Epoch 100/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3450 -
accuracy: 0.8510
```

```
[22]: <keras.callbacks.History at 0x7f03e81b8520>
```

```
[17]: pred_nn = model.predict(X_test_reduced)
s = 0
for i in range(len(X_test)):
    true = np.argmax(y_test_onehot[i])
    guess = np.argmax(pred_nn[i])
    if true == guess:
        s += 1
print("test accuracy is " + str(s/len(y_test)))
```

```
test accuracy is 0.853199804592086
```

Still no improvement. Now let's try the neural net again with the full dataset, just because thus far the models we have made have been a bit sad

```
[23]: X_train, X_test, y_train_onehot, y_test_onehot, y_train, y_test =␣
 ↪train_test_split(X, y_onehot, Y, test_size = 0.2 )

in_size = len(X_train[0])
out_size = len(y_train_onehot[0])

model = Sequential()
model.add(Dense(units = 512, activation = "sigmoid", input_dim = in_size))
model.add(Dense(units = out_size, activation = "softmax"))
model.compile(loss = "BinaryCrossentropy", optimizer = "Adamax", metrics =␣
 ↪"accuracy")

#KLDivergence
#LogCosh

model.fit(X_train, y_train_onehot, epochs = 100, batch_size = 128)
```

```
Epoch 1/100
128/128 [==============================] - 1s 2ms/step - loss: 0.4220 -
accuracy: 0.8211
```

```
Epoch 2/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3686 -
accuracy: 0.8483
Epoch 3/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3364 -
accuracy: 0.8757
Epoch 4/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3148 -
accuracy: 0.8868
Epoch 5/100
128/128 [==============================] - 0s 2ms/step - loss: 0.3002 -
accuracy: 0.8902
Epoch 6/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2888 -
accuracy: 0.8948
Epoch 7/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2807 -
accuracy: 0.8969
Epoch 8/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2738 -
accuracy: 0.8999
Epoch 9/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2689 -
accuracy: 0.9017
Epoch 10/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2649 -
accuracy: 0.9024
Epoch 11/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2613 -
accuracy: 0.9030
Epoch 12/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2584 -
accuracy: 0.9049
Epoch 13/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2555 -
accuracy: 0.9070
Epoch 14/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2526 -
accuracy: 0.9075
Epoch 15/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2510 -
accuracy: 0.9076
Epoch 16/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2491 -
accuracy: 0.9086
Epoch 17/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2460 -
accuracy: 0.9102
```

```
Epoch 18/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2455 -
accuracy: 0.9092
Epoch 19/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2431 -
accuracy: 0.9116
Epoch 20/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2411 -
accuracy: 0.9113
Epoch 21/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2402 -
accuracy: 0.9120
Epoch 22/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2388 -
accuracy: 0.9131
Epoch 23/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2382 -
accuracy: 0.9138
Epoch 24/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2366 -
accuracy: 0.9139
Epoch 25/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2368 -
accuracy: 0.9142
Epoch 26/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2348 -
accuracy: 0.9139
Epoch 27/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2346 -
accuracy: 0.9158
Epoch 28/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2327 -
accuracy: 0.9165
Epoch 29/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2309 -
accuracy: 0.9171
Epoch 30/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2306 -
accuracy: 0.9175
Epoch 31/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2300 -
accuracy: 0.9182
Epoch 32/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2293 -
accuracy: 0.9184
Epoch 33/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2280 -
accuracy: 0.9180
```

```
Epoch 34/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2272 -
accuracy: 0.9189
Epoch 35/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2264 -
accuracy: 0.9190
Epoch 36/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2258 -
accuracy: 0.9191
Epoch 37/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2244 -
accuracy: 0.9210
Epoch 38/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2229 -
accuracy: 0.9204
Epoch 39/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2212 -
accuracy: 0.9204
Epoch 40/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2215 -
accuracy: 0.9205
Epoch 41/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2206 -
accuracy: 0.9216
Epoch 42/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2204 -
accuracy: 0.9203
Epoch 43/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2201 -
accuracy: 0.9215
Epoch 44/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2200 -
accuracy: 0.9218
Epoch 45/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2206 -
accuracy: 0.9202
Epoch 46/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2193 -
accuracy: 0.9223
Epoch 47/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2180 -
accuracy: 0.9226
Epoch 48/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2168 -
accuracy: 0.9229
Epoch 49/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2159 -
accuracy: 0.9226
```

```
Epoch 50/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2172 -
accuracy: 0.9220
Epoch 51/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2158 -
accuracy: 0.9225
Epoch 52/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2156 -
accuracy: 0.9216
Epoch 53/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2157 -
accuracy: 0.9238
Epoch 54/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2140 -
accuracy: 0.9230
Epoch 55/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2138 -
accuracy: 0.9239
Epoch 56/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2131 -
accuracy: 0.9239
Epoch 57/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2140 -
accuracy: 0.9241
Epoch 58/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2139 -
accuracy: 0.9249
Epoch 59/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2114 -
accuracy: 0.9253
Epoch 60/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2107 -
accuracy: 0.9247
Epoch 61/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2128 -
accuracy: 0.9248
Epoch 62/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2103 -
accuracy: 0.9248
Epoch 63/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2106 -
accuracy: 0.9252
Epoch 64/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2093 -
accuracy: 0.9251
Epoch 65/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2093 -
accuracy: 0.9247
```

```
Epoch 66/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2102 -
accuracy: 0.9250
Epoch 67/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2096 -
accuracy: 0.9261
Epoch 68/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2090 -
accuracy: 0.9261
Epoch 69/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2095 -
accuracy: 0.9250
Epoch 70/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2092 -
accuracy: 0.9252
Epoch 71/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2101 -
accuracy: 0.9260
Epoch 72/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2079 -
accuracy: 0.9263
Epoch 73/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2066 -
accuracy: 0.9269
Epoch 74/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2066 -
accuracy: 0.9261
Epoch 75/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2065 -
accuracy: 0.9262
Epoch 76/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2058 -
accuracy: 0.9263
Epoch 77/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2057 -
accuracy: 0.9270
Epoch 78/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2042 -
accuracy: 0.9279
Epoch 79/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2032 -
accuracy: 0.9282
Epoch 80/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2035 -
accuracy: 0.9277
Epoch 81/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2030 -
accuracy: 0.9288
```

```
Epoch 82/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2030 -
accuracy: 0.9284
Epoch 83/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2035 -
accuracy: 0.9288
Epoch 84/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2036 -
accuracy: 0.9282
Epoch 85/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2025 -
accuracy: 0.9284
Epoch 86/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2022 -
accuracy: 0.9287
Epoch 87/100
128/128 [==============================] - 0s 2ms/step - loss: 0.2011 -
accuracy: 0.9293
Epoch 88/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1993 -
accuracy: 0.9309
Epoch 89/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1988 -
accuracy: 0.9308
Epoch 90/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1988 -
accuracy: 0.9308
Epoch 91/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1994 -
accuracy: 0.9320
Epoch 92/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1975 -
accuracy: 0.9307
Epoch 93/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1981 -
accuracy: 0.9308
Epoch 94/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1972 -
accuracy: 0.9317
Epoch 95/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1979 -
accuracy: 0.9300
Epoch 96/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1970 -
accuracy: 0.9315
Epoch 97/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1965 -
accuracy: 0.9323
```

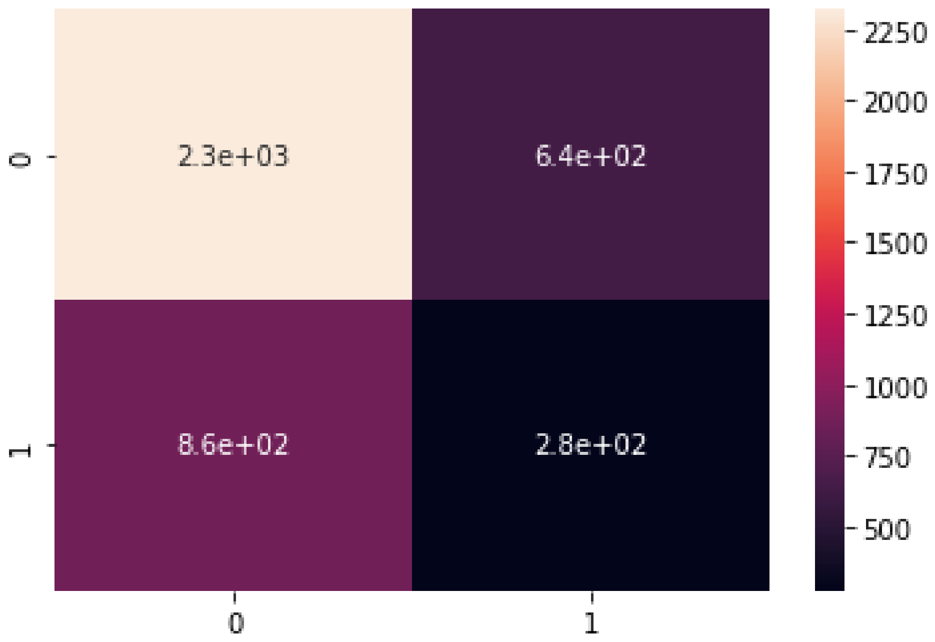
```
Epoch 98/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1961 -
accuracy: 0.9315
Epoch 99/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1960 -
accuracy: 0.9316
Epoch 100/100
128/128 [==============================] - 0s 2ms/step - loss: 0.1963 -
accuracy: 0.9331
```

[23]:
```
<keras.callbacks.History at 0x7f03e0f8c3a0>
```

We're now back at an accuracy comparable to that of the first adaboost model. Let's take a look at the confusion matrix:

[25]:
```
pred_new = np.zeros(len(pred_nn))
i = 0
for val in pred_nn:
    pred_new[i] = np.argmax(pred_nn[i])
    i += 1

cm = confusion_matrix(y_test, pred_new)
sb.heatmap(cm, annot = True)
print(np.sum(cm))
```

```
4094
```



We see that our model would have to be significantly improved if we want to actually use it. If it were to be used as an assist tool it could be ok, but as an automated fraud detector/ preventor it will cause many people a lot of grief. As I've mentioned before, we don't know what each column in the dataset represent. But let's for a moment assume that they represent some information about bank transactions. If this is the case, a relatively large number ($\sim 3.3\%$) of false positives won't be a problem, because if a costumer gets flagged with having multiple predicted fraudulent transactions the probability of these being false is greatly reduced. Unless the machine learning model has falsly correlated some attributes as being predictors, and this costumer over-represent these attributes. A costumer with one positive out of many negatives may warrant investigation but one could perhaps assume the positive to be false. The false negatives are trickier as they won't be flagged at all, and will go on undetected. Unless someone with a high number of true positives also get a few negative predictions (false in this case), it's hard to do anything them at all.
To make a good model we will have to up the accuracy to at least $98-99\%$. To do this, more data would be great. The more rows of data we have, the more probable it becomes that our model will achieve the desired accuracy. There is also a lot more that could be done in the model selection department. Building and tuning a neural network is a dark art and this notebook is just a demo of a few ML techniques and such.

```
[ ]:
```