Fraud Detection Using Autoencoders in Keras with a TensorFlow Backend

Author: David Ellison, PhD Posted on August 9, 2018

In this tutorial, we will use a [neural network](https://datascience.hubs.vidyard.com/watch/CYfbzzj57RPfCwoMnEHD4M) called an autoencoder to detect fraudulent credit/debit card transactions on a Kaggle dataset. We will introduce the importance of the business case, introduce autoencoders, perform an exploratory data analysis, and create and then evaluate the model. The model will be presented using Keras with a TensorFlow backend using a Jupyter Notebook and generally applicable to a wide range of anomaly detection problems.

Introduction

Card Fraud as a Booming Business

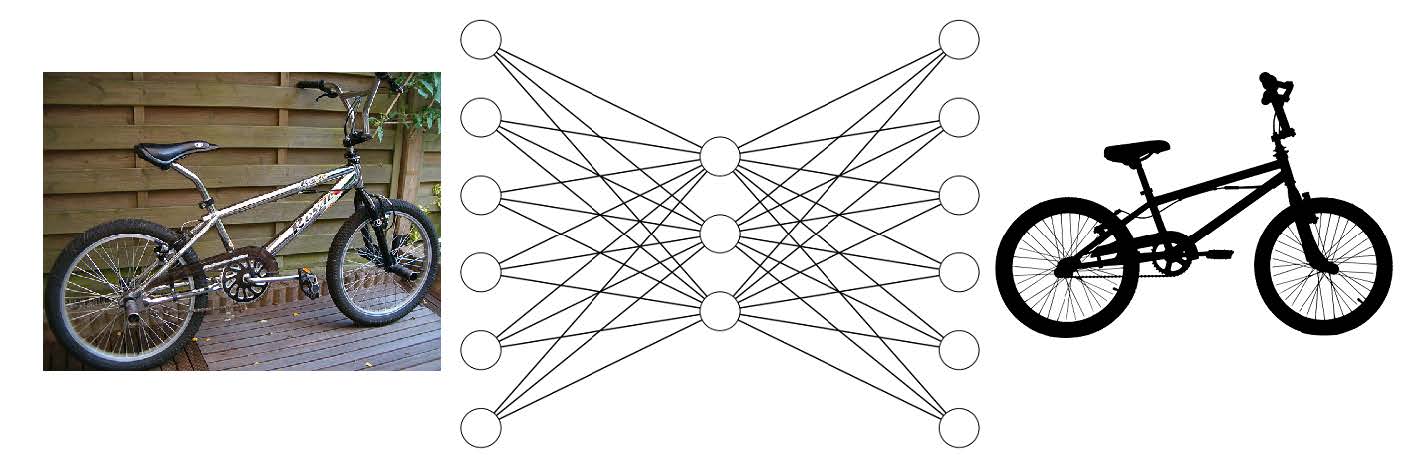
Nilson reports that U.S. card fraud (credit, debt, etc) was reportedly [$9 billion in 2016](https://nilsonreport.com/) and expected to increase to [$12 billion by 2020](https://www.forbes.com/sites/rogeraitken/2016/10/26/us-card-fraud-losses-could-exceed-12bn-by-2020/#59e00dd6d243). For perspective, in 2017 both PayPal's and Mastercard's revenue was only [$10.8 billion each](https://www.forbes.com/global2000/list/26/#header:revenue).

Fraud Detection Algorithms

Traditionally, many major banks have relied on old rules-based expert systems to catch fraud, but these systems have proved all too easy to beat; the financial services industry is relying on increasing complex fraud detection algorithms. Many in the financial services industry have updated their fraud detection to include some basic [machine learning algorithms](https://www.datascience.com/blog/introduction-to-machine-learning-algorithms) including various clustering classifiers, linear approaches, and support vector machines. The most advanced companies in the financial services industry, such as [PayPal](https://www.americanbanker.com/news/how-paypal-is-taking-a-chance-on-ai-to-fight-fraud), have been pioneering more advanced artificial intelligence techniques such as [deep neural networks and autoencoders](https://www.infoworld.com/article/2907877/machine-learning/how-paypal-reduces-fraud-with-machine-learning.html). Long story short, if you want to be where the industry is going and where the jobs are, focus on more advanced fraud detection techniques. This tutorial will focus on one of those more advanced techniques, autoencoders.

Autoencoders and Why You Should Use Them

Autoencoders are a type of neural network that takes an input (e.g. image, dataset), boils that input down to core features, and reverses the process to recreate the input. Although it may sound pointless to feed in input just to get the same thing out, it is in fact very useful for a number of applications. The key here is that the autoencoder boils down (encodes) the input into some key features that it determines in an [unsupervised manner](https://www.datascience.com/blog/supervised-and-unsupervised-machine-learning-algorithms). Hence the name "autoencoder" — it automatically encodes the input.



Let us take this autoencoder of a bicycle as an example. The input is some actual picture of a bicycle that is then reduced to some hidden encoding (perhaps representing components such as handlebars and two wheels) and then is able to reconstruct the original object from that encoding. Of course there will be some loss ("reconstruction error") but hopefully the parts that remain will be the essential pieces of a bicycle.

Now let us assume you fed something into this autoencoder that was a unicycle trying to pose as a bicycle. In the process of breaking down the unicycle into components intended for bicycles, the reconstructed version of the unicycle will be really altered (i.e. suffer a high reconstruction error). It is the assumption in using autoencoders that fraud or anomalies will suffer from a detectably high reconstruction error.

Import Statements

First, let's set up the code and import all the necessary packages.

In [2]:

**# import 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**

**#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**

**#set up graphic style in this case I am using the 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, palette=sns.xkcd\_palette(col\_list))**

Import and Check Data

Download the credit card fraud dataset from [Kaggle](https://www.kaggle.com/mlg-ulb/creditcardfraud/data) and place it in the same directory as your python notebook. The data contains 284,807 European credit card transactions that occurred over two days with 492 fraudulent transactions. Everything except the time and amount has been reduced by a [Principle Component Analysis (PCA) for privacy concerns](https://www.kaggle.com/mlg-ulb/creditcardfraud/home).

In [3]:

**df = pd.read\_csv("creditcard.csv") #unzip and read in data downloaded to the local directory**

**df.head(n=5) #just to check you imported the dataset properly**

Out[3]:

|  | **Time** | **V1** | **V2** | **V3** | **V4** | **V5** | **V6** | **V7** | **V8** | **V9** | **...** | **V21** | **V22** | **V23** | **V24** | **V25** | **V26** | **V27** | **V28** | **Amount** | **Class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.0 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | 0.462388 | 0.239599 | 0.098698 | 0.363787 | ... | -0.018307 | 0.277838 | -0.110474 | 0.066928 | 0.128539 | -0.189115 | 0.133558 | -0.021053 | 149.62 | 0 |
| **1** | 0.0 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | -0.082361 | -0.078803 | 0.085102 | -0.255425 | ... | -0.225775 | -0.638672 | 0.101288 | -0.339846 | 0.167170 | 0.125895 | -0.008983 | 0.014724 | 2.69 | 0 |
| **2** | 1.0 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | 1.800499 | 0.791461 | 0.247676 | -1.514654 | ... | 0.247998 | 0.771679 | 0.909412 | -0.689281 | -0.327642 | -0.139097 | -0.055353 | -0.059752 | 378.66 | 0 |
| **3** | 1.0 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | 1.247203 | 0.237609 | 0.377436 | -1.387024 | ... | -0.108300 | 0.005274 | -0.190321 | -1.175575 | 0.647376 | -0.221929 | 0.062723 | 0.061458 | 123.50 | 0 |
| **4** | 2.0 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | 0.095921 | 0.592941 | -0.270533 | 0.817739 | ... | -0.009431 | 0.798278 | -0.137458 | 0.141267 | -0.206010 | 0.502292 | 0.219422 | 0.215153 | 69.99 | 0 |

The data looks like we would expect on the surface, but let's double check the shape (we are expecting 294,807 rows and 31 columns). It is a well-groomed dataset so we expect no null values.

In [4]:

**df.shape #secondary check on the size of the dataframe**

Out[4]:

**(284807, 31)**

In [5]:

**df.isnull().values.any() #check to see if any values are null, which there are not**

Out[5]:

**False**

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 [6]:

**pd.value\_counts(df['Class'], sort = True) #class comparison 0=Normal 1=Fraud**

Out[6]:

**0 284315**

**1 492**

**Name: Class, 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.

Exploratory Data Analysis

Balance of Data Visualization

Let's get a visual confirmation of the unbalanced data in this fraud dataset.

In [7]:

**#if you don't have an intuitive sense of how imbalanced these two classes are, let's go visual**

**count\_classes = pd.value\_counts(df['Class'], sort = True)**

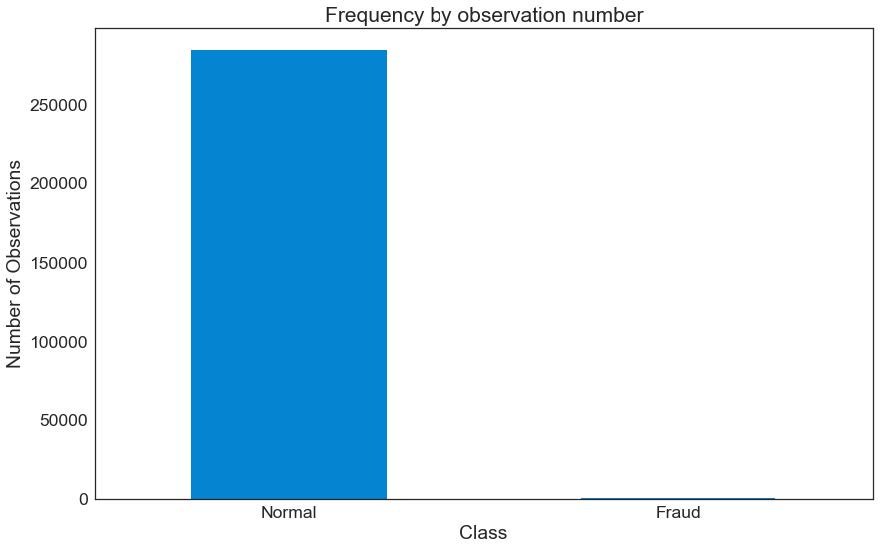
**count\_classes.plot(kind = 'bar', rot=0)**

**plt.xticks(range(2), LABELS)**

**plt.title("Frequency by observation number")**

**plt.xlabel("Class")**

**plt.ylabel("Number of Observations");**

**Error! Filename not specified.**

As you can see, the normal cases strongly outweigh the fraud cases.

Summary Statistics of the Transaction Amount Data

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

In [8]:

**normal\_df = df[df.Class == 0] #save normal\_df observations into a separate df**

**fraud\_df = df[df.Class == 1] #do the same for frauds**

Let's look at some summary statistics and see if there are obvious differences between fraud and normal transactions.

In [9]:

**normal\_df.Amount.describe()**

Out[9]:

**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 [10]:

**fraud\_df.Amount.describe()**

Out[10]:

**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. I could run statistical tests (e.g. t-test) to support the claim that the two samples likely come from populations with similar means and deviations. However, such statistical methods are not the focus of this article on autoencoders.

Visual Exploration of the Transaction Amount Data

We are going to get more familiar with the data and try some basic visuals. In anomaly detection datasets it is common to have the areas of interest "washed out" by abundant data. The most common method is to simply 'slice and dice' the data in a couple different ways until something interesting is found. Although this practice is common, it is **not** a scientifically sound way to explore data. There are always non-meaningful quirks to real data, so just looking until you "find something interesting" is likely going to result in you finding false positives. In other words, you find a random pattern in the current data set that will never be seen again. As a [famous economist](https://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/1991/coase-bio.html) wrote, ["If you torture the data long enough, it will confess."](https://www.goodreads.com/quotes/1249307-if-you-torture-the-data-long-enough-it-will-confess)

In this dataset, I expect 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 I decided to look at the data in a number different $100 and $1,000 intervals. Since it would be tedious to show reader these graphs, I will only show the final graph that only visualizes the transactions above $200.

In [13]:

**#plot of high value transactions**

**bins = np.linspace(200, 2500, 100)**

**plt.hist(normal\_df.Amount, bins, alpha=1, normed=True, label='Normal')**

**plt.hist(fraud\_df.Amount, bins, alpha=0.6, normed=True, label='Fraud')**

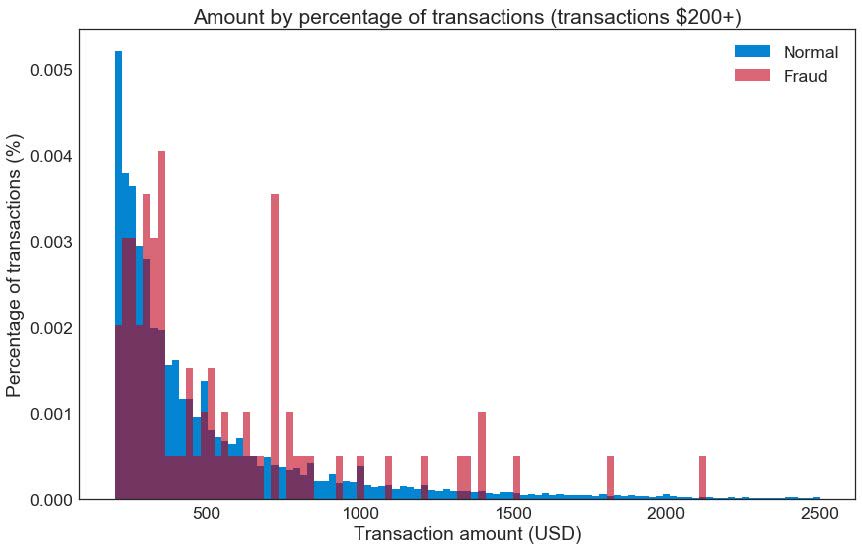
**plt.legend(loc='upper right')**

**plt.title("Amount by percentage of transactions (transactions \$200+)")**

**plt.xlabel("Transaction amount (USD)")**

**plt.ylabel("Percentage of transactions (%)");**

**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.

Visual Exploration of the Data by Hour

With a few exceptions, the transaction amount does not look very informative. Let's look at the time of day next.

In [14]:

**bins = np.linspace(0, 48, 48) #48 hours**

**plt.hist((normal\_df.Time/(60\*60)), bins, alpha=1, normed=True, label='Normal')**

**plt.hist((fraud\_df.Time/(60\*60)), bins, alpha=0.6, normed=True, label='Fraud')**

**plt.legend(loc='upper right')**

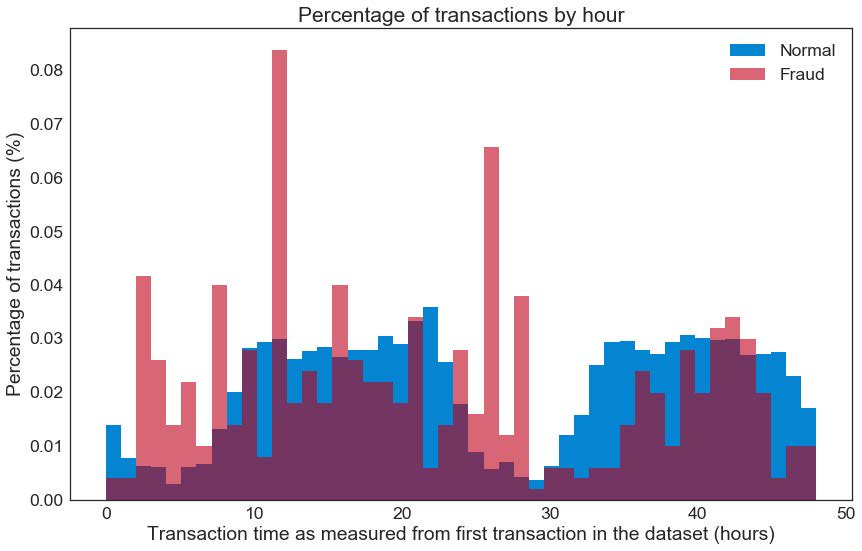
**plt.title("Percentage of transactions by hour")**

**plt.xlabel("Transaction time as measured from first transaction in the dataset (hours)")**

**plt.ylabel("Percentage of transactions (%)");**

**#plt.hist((df.Time/(60\*60)),bins)**

**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, I am assuming 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.

Next, we will explore the potential interaction between transaction amount and hour to see if any patterns emerge.

Visual Exploration of Transaction Amount vs. Hour

In [15]:

**plt.scatter((normal\_df.Time/(60\*60)), normal\_df.Amount, alpha=0.6, label='Normal')**

**plt.scatter((fraud\_df.Time/(60\*60)), fraud\_df.Amount, alpha=0.9, label='Fraud')**

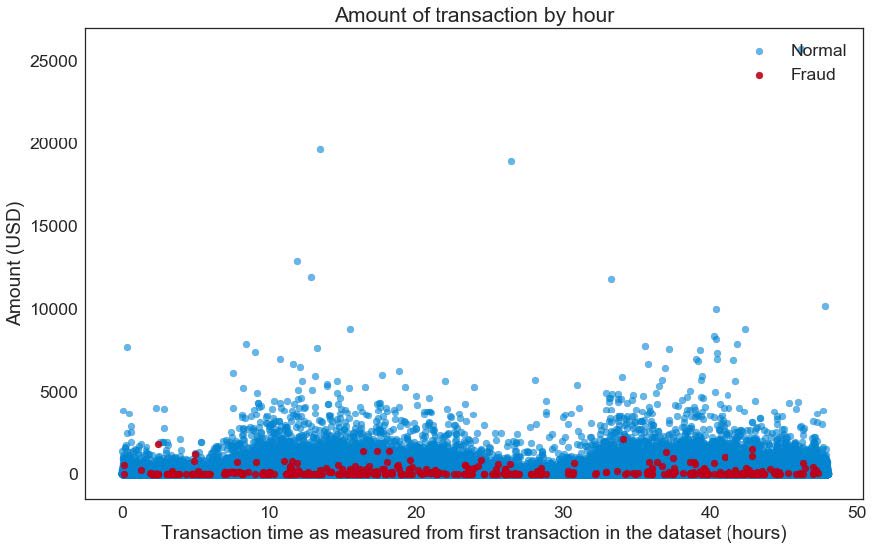
**plt.title("Amount of transaction by hour")**

**plt.xlabel("Transaction time as measured from first transaction in the dataset (hours)")**

**plt.ylabel('Amount (USD)')**

**plt.legend(loc='upper right')**

**plt.show()**

**Error! Filename not specified.**

Again, this is not enough to make a good classifier. For example, it would be hard to draw a line that cleanly separates fraud and normal transactions. For the experienced Data Scientists in the readership, I am excluding more advanced techniques such as the kernel trick.

Model Setup: Basic Autoencoder

Now that more simplistic methods are not proving that useful, we are justified in exploring our autoencoder to see if it does a little better.

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. Although there are many different scaling methods and reasons to choose one method over the other, in this case I will err towards consistency. The reader may remember that 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. I will do the same to the 'time' and 'amount' columns.

In [16]:

**#data = df.drop(['Time'], axis=1) #if you think the var is unimportant**

**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))**

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. This should have been done before the exploratory data analysis, but for ease of explanation I delayed it until right before the model.

In [17]:

**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**

Just confirming the new ndarray is the expected shape.

In [18]:

**train\_x.shape**

Out[18]:

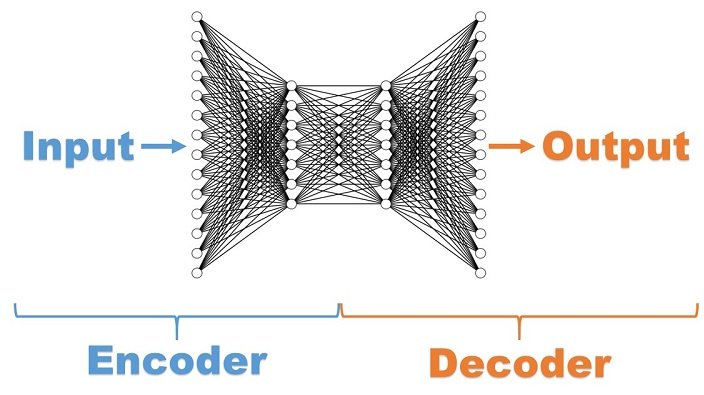
**(227468, 30)**

Creating The Model

Autoencoder Layer Structure and Parameters

Below we set up the structure of the autoencoder. It has symmetric encoding and decoding layers that are "dense" (e.g. full connected). The choice of the size of these layers is relatively arbitrary and generally the coder experiments with a few different layer sizes.

Remember you 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 I expect to represent some feature. In this case, I am encoding 30 columns into 14 dimensions so I am expecting high-level features to be represented by roughly two columns (30/14 = 2.1). Of those high-level features, I am 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 or for reasons that can and have repeatedly filled data science books. Explanation of the balancing of these values is far beyond this tutorial, but I would refer you to excellent texts such as [Hands-On Machine Learning with Scikit-Learn & TensorFlow](http://shop.oreilly.com/product/0636920052289.do) or [Deep Learning](http://shop.oreilly.com/product/0636920035343.do).

In [22]:

**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**

**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)**

Model Training and Logging

Below is where we set up the actual run including checkpoints and the tensorboard.

In [23]:

**autoencoder.compile(metrics=['accuracy'],**

**loss='mean\_squared\_error',**

**optimizer='adam')**

**cp = ModelCheckpoint(filepath="autoencoder\_fraud.h5",**

**save\_best\_only=True,**

**verbose=0)**

**tb = TensorBoard(log\_dir='./logs',**

**histogram\_freq=0,**

**write\_graph=True,**

**write\_images=True)**

**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, tb]).history**

**Train on 227468 samples, validate on 56962 samples**

**Epoch 1/100**

**227468/227468 [==============================] - 7s 29us/step - loss: 0.8688 - acc: 0.4782 - val\_loss: 0.8266 - val\_acc: 0.5893**

**Epoch 2/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.7767 - acc: 0.6053 - val\_loss: 0.7980 - val\_acc: 0.6191**

**Epoch 3/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7575 - acc: 0.6291 - val\_loss: 0.7855 - val\_acc: 0.6376**

**Epoch 4/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7473 - acc: 0.6395 - val\_loss: 0.7781 - val\_acc: 0.6412**

**Epoch 5/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7393 - acc: 0.6487 - val\_loss: 0.7705 - val\_acc: 0.6581**

**Epoch 6/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7319 - acc: 0.6613 - val\_loss: 0.7651 - val\_acc: 0.6663**

**Epoch 7/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7265 - acc: 0.6716 - val\_loss: 0.7602 - val\_acc: 0.6753**

**Epoch 8/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.7223 - acc: 0.6777 - val\_loss: 0.7569 - val\_acc: 0.6803**

**Epoch 9/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7185 - acc: 0.6845 - val\_loss: 0.7526 - val\_acc: 0.6867**

**Epoch 10/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7148 - acc: 0.6900 - val\_loss: 0.7500 - val\_acc: 0.6921**

**Epoch 11/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.7122 - acc: 0.6923 - val\_loss: 0.7456 - val\_acc: 0.6938**

**Epoch 12/100**

**227468/227468 [==============================] - 5s 23us/step - loss: 0.7101 - acc: 0.6939 - val\_loss: 0.7441 - val\_acc: 0.6903**

**Epoch 13/100**

**227468/227468 [==============================] - 6s 24us/step - loss: 0.7084 - acc: 0.6966 - val\_loss: 0.7429 - val\_acc: 0.6963**

**Epoch 14/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.7073 - acc: 0.6969 - val\_loss: 0.7412 - val\_acc: 0.6995**

**Epoch 15/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.7065 - acc: 0.6982 - val\_loss: 0.7408 - val\_acc: 0.7010**

**Epoch 16/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7060 - acc: 0.6987 - val\_loss: 0.7406 - val\_acc: 0.7029**

**Epoch 17/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.7053 - acc: 0.6989 - val\_loss: 0.7405 - val\_acc: 0.6977**

**Epoch 18/100**

**227468/227468 [==============================] - 5s 23us/step - loss: 0.7049 - acc: 0.6996 - val\_loss: 0.7408 - val\_acc: 0.7030**

**Epoch 19/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.7044 - acc: 0.6995 - val\_loss: 0.7388 - val\_acc: 0.6988**

**Epoch 20/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.7042 - acc: 0.6997 - val\_loss: 0.7389 - val\_acc: 0.7003**

**Epoch 21/100**

**227468/227468 [==============================] - 6s 24us/step - loss: 0.7041 - acc: 0.7000 - val\_loss: 0.7395 - val\_acc: 0.7023**

**Epoch 22/100**

**227468/227468 [==============================] - 5s 24us/step - loss: 0.7037 - acc: 0.7005 - val\_loss: 0.7393 - val\_acc: 0.7025**

**Epoch 23/100**

**227468/227468 [==============================] - 5s 22us/step - loss: 0.7037 - acc: 0.6995 - val\_loss: 0.7377 - val\_acc: 0.6994**

**Epoch 24/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.7031 - acc: 0.6998 - val\_loss: 0.7384 - val\_acc: 0.6959**

**Epoch 25/100**

**227468/227468 [==============================] - 6s 25us/step - loss: 0.7029 - acc: 0.6999 - val\_loss: 0.7378 - val\_acc: 0.7061**

**Epoch 26/100**

**227468/227468 [==============================] - 5s 24us/step - loss: 0.7030 - acc: 0.6997 - val\_loss: 0.7373 - val\_acc: 0.7016**

**Epoch 27/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.7027 - acc: 0.6995 - val\_loss: 0.7380 - val\_acc: 0.7038**

**Epoch 28/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7024 - acc: 0.6998 - val\_loss: 0.7368 - val\_acc: 0.7028**

**Epoch 29/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7023 - acc: 0.6995 - val\_loss: 0.7376 - val\_acc: 0.7034**

**Epoch 30/100**

**227468/227468 [==============================] - 4s 20us/step - loss: 0.7021 - acc: 0.7002 - val\_loss: 0.7389 - val\_acc: 0.7040**

**Epoch 31/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.7020 - acc: 0.6998 - val\_loss: 0.7382 - val\_acc: 0.6970**

**Epoch 32/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7018 - acc: 0.7000 - val\_loss: 0.7384 - val\_acc: 0.6972**

**Epoch 33/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7017 - acc: 0.7008 - val\_loss: 0.7371 - val\_acc: 0.7030**

**Epoch 34/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7014 - acc: 0.7008 - val\_loss: 0.7385 - val\_acc: 0.7024**

**Epoch 35/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7015 - acc: 0.7013 - val\_loss: 0.7389 - val\_acc: 0.6949**

**Epoch 36/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7011 - acc: 0.7011 - val\_loss: 0.7354 - val\_acc: 0.7014**

**Epoch 37/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7014 - acc: 0.7010 - val\_loss: 0.7359 - val\_acc: 0.7027**

**Epoch 38/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7009 - acc: 0.7018 - val\_loss: 0.7369 - val\_acc: 0.7021**

**Epoch 39/100**

**227468/227468 [==============================] - 6s 26us/step - loss: 0.7008 - acc: 0.7032 - val\_loss: 0.7381 - val\_acc: 0.7010**

**Epoch 40/100**

**227468/227468 [==============================] - 5s 22us/step - loss: 0.7011 - acc: 0.7024 - val\_loss: 0.7383 - val\_acc: 0.7015**

**Epoch 41/100**

**227468/227468 [==============================] - 6s 24us/step - loss: 0.7009 - acc: 0.7024 - val\_loss: 0.7373 - val\_acc: 0.6942**

**Epoch 42/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7005 - acc: 0.7034 - val\_loss: 0.7355 - val\_acc: 0.7117**

**Epoch 43/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.7004 - acc: 0.7029 - val\_loss: 0.7350 - val\_acc: 0.7068**

**Epoch 44/100**

**227468/227468 [==============================] - 4s 18us/step - loss: 0.7005 - acc: 0.7030 - val\_loss: 0.7353 - val\_acc: 0.7078**

**Epoch 45/100**

**227468/227468 [==============================] - 4s 18us/step - loss: 0.7004 - acc: 0.7040 - val\_loss: 0.7354 - val\_acc: 0.7082**

**Epoch 46/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.7002 - acc: 0.7044 - val\_loss: 0.7348 - val\_acc: 0.7048**

**Epoch 47/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.7003 - acc: 0.7039 - val\_loss: 0.7351 - val\_acc: 0.7065**

**Epoch 48/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.7000 - acc: 0.7045 - val\_loss: 0.7346 - val\_acc: 0.7062**

**Epoch 49/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.6997 - acc: 0.7047 - val\_loss: 0.7353 - val\_acc: 0.7121**

**Epoch 50/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.7000 - acc: 0.7049 - val\_loss: 0.7346 - val\_acc: 0.7088**

**Epoch 51/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.6997 - acc: 0.7045 - val\_loss: 0.7355 - val\_acc: 0.7058**

**Epoch 52/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.6997 - acc: 0.7043 - val\_loss: 0.7353 - val\_acc: 0.7114**

**Epoch 53/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.6995 - acc: 0.7048 - val\_loss: 0.7378 - val\_acc: 0.7015**

**Epoch 54/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.6995 - acc: 0.7052 - val\_loss: 0.7340 - val\_acc: 0.7035**

**Epoch 55/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.6993 - acc: 0.7054 - val\_loss: 0.7342 - val\_acc: 0.7043**

**Epoch 56/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.6992 - acc: 0.7056 - val\_loss: 0.7346 - val\_acc: 0.7092**

**Epoch 57/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.6990 - acc: 0.7056 - val\_loss: 0.7340 - val\_acc: 0.7056**

**Epoch 58/100**

**227468/227468 [==============================] - 4s 16us/step - loss: 0.6991 - acc: 0.7059 - val\_loss: 0.7350 - val\_acc: 0.7033**

**Epoch 59/100**

**227468/227468 [==============================] - 4s 17us/step - loss: 0.6990 - acc: 0.7058 - val\_loss: 0.7335 - val\_acc: 0.7134**

**Epoch 60/100**

**227468/227468 [==============================] - 4s 18us/step - loss: 0.6987 - acc: 0.7056 - val\_loss: 0.7328 - val\_acc: 0.7079**

**Epoch 61/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6986 - acc: 0.7056 - val\_loss: 0.7324 - val\_acc: 0.7090**

**Epoch 62/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6985 - acc: 0.7054 - val\_loss: 0.7338 - val\_acc: 0.7097**

**Epoch 63/100**

**227468/227468 [==============================] - 4s 18us/step - loss: 0.6987 - acc: 0.7052 - val\_loss: 0.7337 - val\_acc: 0.7049**

**Epoch 64/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6981 - acc: 0.7051 - val\_loss: 0.7357 - val\_acc: 0.7065**

**Epoch 65/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6984 - acc: 0.7044 - val\_loss: 0.7331 - val\_acc: 0.6995**

**Epoch 66/100**

**227468/227468 [==============================] - 5s 22us/step - loss: 0.6982 - acc: 0.7038 - val\_loss: 0.7321 - val\_acc: 0.7048**

**Epoch 67/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6978 - acc: 0.7033 - val\_loss: 0.7323 - val\_acc: 0.7061**

**Epoch 68/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6976 - acc: 0.7035 - val\_loss: 0.7324 - val\_acc: 0.7038**

**Epoch 69/100**

**227468/227468 [==============================] - 4s 20us/step - loss: 0.6979 - acc: 0.7032 - val\_loss: 0.7337 - val\_acc: 0.7021**

**Epoch 70/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6976 - acc: 0.7038 - val\_loss: 0.7335 - val\_acc: 0.7061**

**Epoch 71/100**

**227468/227468 [==============================] - 5s 22us/step - loss: 0.6978 - acc: 0.7036 - val\_loss: 0.7326 - val\_acc: 0.7086**

**Epoch 72/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6974 - acc: 0.7044 - val\_loss: 0.7332 - val\_acc: 0.7080**

**Epoch 73/100**

**227468/227468 [==============================] - 5s 24us/step - loss: 0.6973 - acc: 0.7036 - val\_loss: 0.7328 - val\_acc: 0.7054**

**Epoch 74/100**

**227468/227468 [==============================] - 6s 25us/step - loss: 0.6974 - acc: 0.7046 - val\_loss: 0.7325 - val\_acc: 0.7077**

**Epoch 75/100**

**227468/227468 [==============================] - 6s 26us/step - loss: 0.6974 - acc: 0.7041 - val\_loss: 0.7317 - val\_acc: 0.7025**

**Epoch 76/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6973 - acc: 0.7040 - val\_loss: 0.7336 - val\_acc: 0.7054**

**Epoch 77/100**

**227468/227468 [==============================] - 4s 20us/step - loss: 0.6971 - acc: 0.7046 - val\_loss: 0.7320 - val\_acc: 0.7049**

**Epoch 78/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6972 - acc: 0.7034 - val\_loss: 0.7356 - val\_acc: 0.6989**

**Epoch 79/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6972 - acc: 0.7053 - val\_loss: 0.7326 - val\_acc: 0.6985**

**Epoch 80/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6973 - acc: 0.7046 - val\_loss: 0.7328 - val\_acc: 0.6986**

**Epoch 81/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6969 - acc: 0.7049 - val\_loss: 0.7314 - val\_acc: 0.7094**

**Epoch 82/100**

**227468/227468 [==============================] - 4s 18us/step - loss: 0.6971 - acc: 0.7049 - val\_loss: 0.7324 - val\_acc: 0.7027**

**Epoch 83/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6970 - acc: 0.7053 - val\_loss: 0.7349 - val\_acc: 0.7080**

**Epoch 84/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6970 - acc: 0.7057 - val\_loss: 0.7328 - val\_acc: 0.7017**

**Epoch 85/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6968 - acc: 0.7047 - val\_loss: 0.7319 - val\_acc: 0.7055**

**Epoch 86/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6969 - acc: 0.7050 - val\_loss: 0.7343 - val\_acc: 0.7056**

**Epoch 87/100**

**227468/227468 [==============================] - 4s 18us/step - loss: 0.6968 - acc: 0.7061 - val\_loss: 0.7315 - val\_acc: 0.7090**

**Epoch 88/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6968 - acc: 0.7059 - val\_loss: 0.7330 - val\_acc: 0.7010**

**Epoch 89/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6967 - acc: 0.7052 - val\_loss: 0.7318 - val\_acc: 0.7028**

**Epoch 90/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6969 - acc: 0.7055 - val\_loss: 0.7331 - val\_acc: 0.7003**

**Epoch 91/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6969 - acc: 0.7058 - val\_loss: 0.7326 - val\_acc: 0.7050**

**Epoch 92/100**

**227468/227468 [==============================] - 4s 19us/step - loss: 0.6965 - acc: 0.7059 - val\_loss: 0.7325 - val\_acc: 0.7058**

**Epoch 93/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6968 - acc: 0.7059 - val\_loss: 0.7314 - val\_acc: 0.7091**

**Epoch 94/100**

**227468/227468 [==============================] - 5s 24us/step - loss: 0.6969 - acc: 0.7054 - val\_loss: 0.7315 - val\_acc: 0.7110**

**Epoch 95/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6968 - acc: 0.7060 - val\_loss: 0.7315 - val\_acc: 0.7060**

**Epoch 96/100**

**227468/227468 [==============================] - 5s 21us/step - loss: 0.6966 - acc: 0.7053 - val\_loss: 0.7318 - val\_acc: 0.7104**

**Epoch 97/100**

**227468/227468 [==============================] - 5s 24us/step - loss: 0.6964 - acc: 0.7059 - val\_loss: 0.7356 - val\_acc: 0.7011**

**Epoch 98/100**

**227468/227468 [==============================] - 6s 26us/step - loss: 0.6965 - acc: 0.7058 - val\_loss: 0.7330 - val\_acc: 0.6992**

**Epoch 99/100**

**227468/227468 [==============================] - 5s 20us/step - loss: 0.6968 - acc: 0.7063 - val\_loss: 0.7318 - val\_acc: 0.7094**

**Epoch 100/100**

**227468/227468 [==============================] - 5s 23us/step - loss: 0.6965 - acc: 0.7061 - val\_loss: 0.7312 - val\_acc: 0.7021**

In [32]:

**autoencoder = load\_model('autoencoder\_fraud.h5')**

Model Evaluation

Model Loss

The model seems to be performing well enough, although there is significant room for improvement. This simple autoencoder architecture was chosen for ease of explanation within this tutorial. However, it is my intuition that it is too simple relative to complex financial data and that overall performance could be improved by adding more hidden layers. More hidden layers would allow this network to encode more complex relationships between the input features. Please feel free to experiment with the code and let me know what you find out.

The loss of our current model seems to be converging and so more training epochs are not likely going to help. Let's explore this visually to confirm.

In [33]:

**plt.plot(history['loss'], linewidth=2, label='Train')**

**plt.plot(history['val\_loss'], linewidth=2, label='Test')**

**plt.legend(loc='upper right')**

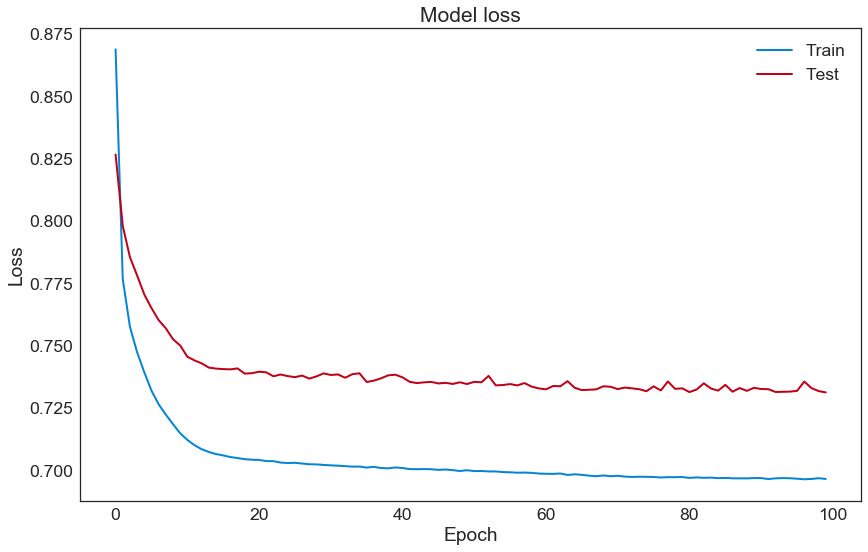
**plt.title('Model loss')**

**plt.ylabel('Loss')**

**plt.xlabel('Epoch')**

**#plt.ylim(ymin=0.70,ymax=1)**

**plt.show()**



Reconstruction Error Check

Autoencoders are trained to reduce reconstruction error which we show below:

In [34]:

**test\_x\_predictions = autoencoder.predict(test\_x)**

**mse = np.mean(np.power(test\_x - test\_x\_predictions, 2), axis=1)**

**error\_df = pd.DataFrame({'Reconstruction\_error': mse,**

**'True\_class': test\_y})**

**error\_df.describe()**

Out[34]:

|  | **Reconstruction\_error** | **True\_class** |
| --- | --- | --- |
| **count** | 56962.000000 | 56962.000000 |
| **mean** | 0.731118 | 0.002019 |
| **std** | 3.144372 | 0.044887 |
| **min** | 0.050197 | 0.000000 |
| **25%** | 0.248630 | 0.000000 |
| **50%** | 0.386735 | 0.000000 |
| **75%** | 0.616129 | 0.000000 |
| **max** | 204.803125 | 1.000000 |

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. Without getting into detail, this is something called the [Accuracy Paradox](https://en.wikipedia.org/wiki/Accuracy_paradox).

In [35]:

**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.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])**

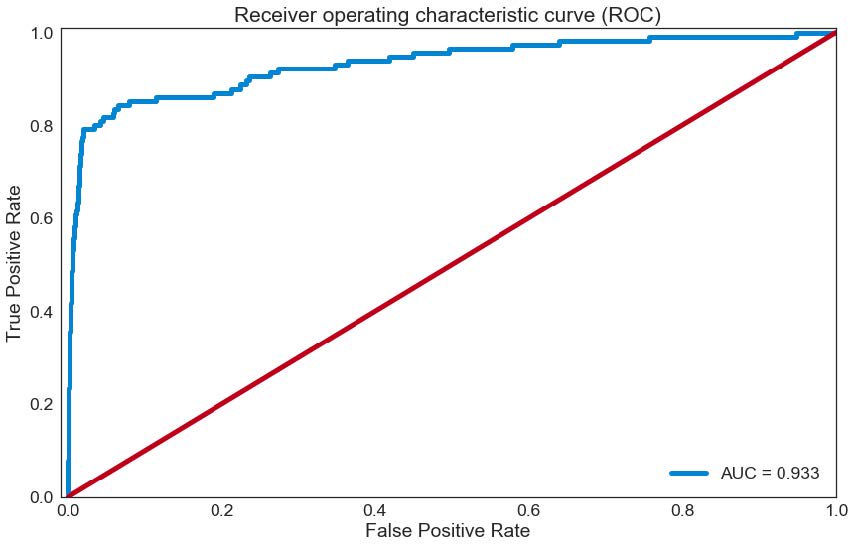
**plt.legend(loc='lower right')**

**plt.title('Receiver operating characteristic curve (ROC)')**

**plt.ylabel('True Positive Rate')**

**plt.xlabel('False Positive Rate')**

**plt.show()**



Recall vs. Precision Thresholding

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

In [36]:

**precision\_rt, recall\_rt, threshold\_rt = precision\_recall\_curve(error\_df.True\_class, error\_df.Reconstruction\_error)**

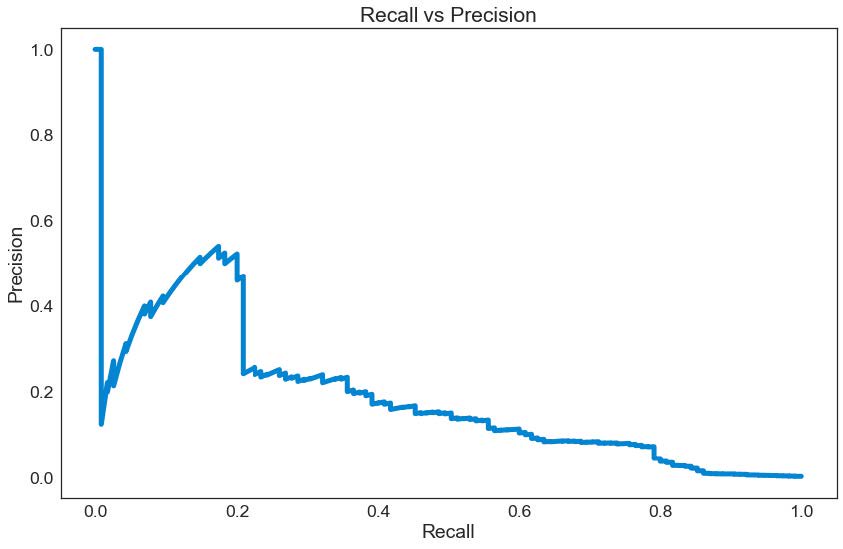
**plt.plot(recall\_rt, precision\_rt, linewidth=5, label='Precision-Recall curve')**

**plt.title('Recall vs Precision')**

**plt.xlabel('Recall')**

**plt.ylabel('Precision')**

**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. Add those two weights to the calculation and you can come up with some theoretical optimal solution. This is rarely the way it is done in practice, however, as it is hard to quantify a lot of those costs (such as customer annoyance at getting fraud alerts too frequently), or because of various structural, technical, or business rules preventing the optimized solution from being chosen.

In [37]:

**plt.plot(threshold\_rt, precision\_rt[1:], label="Precision",linewidth=5)**

**plt.plot(threshold\_rt, recall\_rt[1:], label="Recall",linewidth=5)**

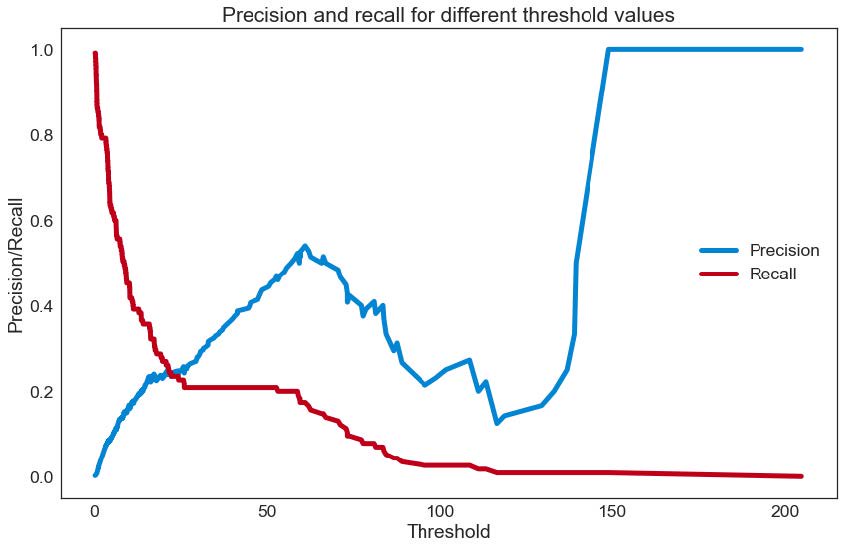
**plt.title('Precision and recall for different threshold values')**

**plt.xlabel('Threshold')**

**plt.ylabel('Precision/Recall')**

**plt.legend()**

**plt.show()**



Now that we have talked with the business client and established a threshold, let's see how that compares to reconstruction error. Where the threshold is set seems to miss the main cluster of the normal transactions, but still get a lot of the fraud transactions.

Reconstruction Error vs Threshold Check

In [38]:

**threshold\_fixed = 5**

**groups = error\_df.groupby('True\_class')**

**fig, ax = plt.subplots()**

**for name, group in groups:**

**ax.plot(group.index, group.Reconstruction\_error, marker='o', ms=3.5, linestyle='',**

**label= "Fraud" if name == 1 else "Normal")**

**ax.hlines(threshold\_fixed, ax.get\_xlim()[0], ax.get\_xlim()[1], colors="r", zorder=100, label='Threshold')**

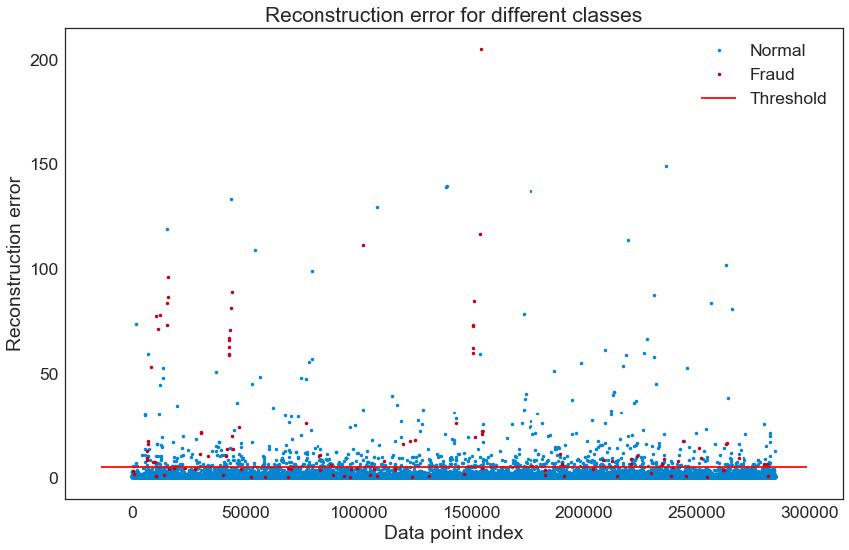
**ax.legend()**

**plt.title("Reconstruction error for different classes")**

**plt.ylabel("Reconstruction error")**

**plt.xlabel("Data point index")**

**plt.show();**



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. A 1:10 ratio is a fairly standard benchmark if there are no business rules or cost tradeoffs that dominate that decision. However, I can assure any data scientist that there will indeed be at least those outside influences, if not vastly more outside influences ranging from regulatory and privacy concerns to executive confidence in data and technology in general.

In [39]:

**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=(12, 12))**

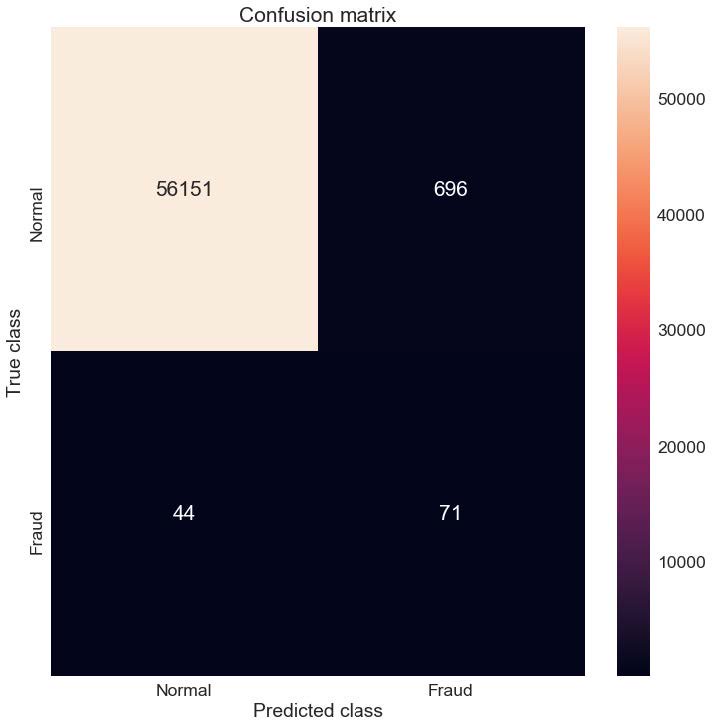
**sns.heatmap(conf\_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d");**

**plt.title("Confusion matrix")**

**plt.ylabel('True class')**

**plt.xlabel('Predicted class')**

**plt.show()**

**Error! Filename not specified.**

You will also notice we caught about 60% of the fraud cases, which might seem low at face value but remember there are no magic bullets here. Remember two things: 1) you will never catch even close to 100% of the fraud cases in any way that is even remotely real-world useful and 2) your fraud detection algorithm will be running as a part of the overall ensemble of fraud detectors that will hopefully complement your model.

Data science, as with so much else in life, is a team effort. With this tutorial and some real-world experience, it is my hope that the reader will be able to contribute more value to the organization or community in which they choose to operate.

Conclusion

In this tutorial, I presented the business case for card payment fraud detection and provided a brief overview of the algorithms in use. I then used some basic exploratory data analysis techniques to show that simple linear methods would not be a good choice as a fraud detection algorithm and so chose to explore autoencoders. I then explained and ran a simple autoencoder written in Keras and analyzed the utility of that model. Finally, I discussed some of the business and real-world implications to choices made with the model.

# Credit Card Fraud Detection using Autoencoders in H2O

[](https://towardsdatascience.com/@manisharajarathna?source=post_header_lockup)

[Maneesha Rajaratne](https://towardsdatascience.com/@manisharajarathna)Follow

Nov 17, 2018



Frauds in the finance field are very rare to be identified. Because of that, it can do a severe damage to the financial field. It is estimated that fraud costs at least $80 billion a year across all lines of insurance. If there is a small possibility of detecting fraudulent activities, that can do a major impact on annual losses. That is why financial companies invest in machine learning as a preemptive approach to tackling fraud.

The benefits of using a machine learning approach are that,

* It helps to find hidden and implicit correlations in data.
* Faster data processing and less manual work
* Automatic detection of possible fraud scenarios.

The best way to detect frauds is anomaly detection.

### Anomaly Detection

Anomaly detection is a technique to identify unusual patterns that do not conform to the expected behaviors, called outliers. It has many applications in business from fraud detection in credit card transactions to fault detection in operating environments. Machine learning approaches for Anomaly detection;

* K-Nearest Neighbor
* Autoencoders — Deep neural network
* K-means
* Support Vector Machine
* Naive Bayes

Today we will be using Autoencoders to train the model.

### ****Autoencoders****

Most of us are not familiar with this model. Autoencoders is an unsupervised Neural Network. It is a data compression algorithm which takes the input and going through a compressed representation and gives the reconstructed output.

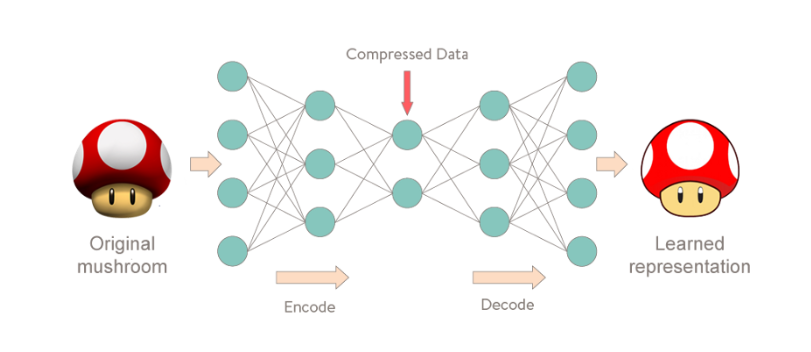


Figure 1: Neural network representation of Autoencoders

### Dataset

As for the dataset we will be using Credit Card Transaction dataset provided by Kaggle: <https://www.kaggle.com/mlg-ulb/creditcardfraud>

The dataset includes 284,807 transactions. among them, 492 transactions are labeled as frauds. Because of this, the dataset is highly imbalanced. It contains only numerical variables. Feature **‘Time’**contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature **‘Amount’**is the transaction Amount, this feature can be used for example-dependent cost-sensitive learning. Feature **‘Class’**is the response variable and it takes value 1 in case of fraud and 0 otherwise.

You can find my Kaggle Kernel here: <https://www.kaggle.com/maneesha96/credit-card-fraud-detection-using-autoencoders>

Full code: <https://github.com/Mash96/Credit-Card-Fraud-Detection>

Then Let's get started!!!

#### Setup

We will be using H2O as the ML platform today. You can find more info here: [https://www.h2o.ai](https://www.h2o.ai/)

import h2o  
import matplotlib.pyplot as plt  
from pylab import rcParams  
import numpy as np # linear algebra  
import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)  
import os  
from h2o.estimators.deeplearning import H2OAutoEncoderEstimator, H2ODeepLearningEstimator

Initialize H2O server

h2o.init(max\_mem\_size = 2) # initializing h2o server  
h2o.remove\_all()

Loading dataset using pandas data frame

creditData = pd.read\_csv(r"File\_Path\creditcard.csv")   
creditData.describe()  
# H2O method

# creditData\_df = h2o.import\_file(r"File\_Path\creditcard.csv")

#### Exploration

creditData.shape

> (284807, 31)

Checking for null values in the dataset

creditData.isnull().values.any() # pandas method  
# creditData\_h2o.na\_omit() # h2o method  
# creditData\_h2o.nacnt() # no missing values found

> False

In order to proceed we need to convert the pandas data frame to H2O data frame

# Turns python pandas frame into an H2OFrame  
creditData\_h2o = h2o.H2OFrame(creditData)

# Let’s plot the Transaction class against the Frequency  
labels = [‘normal’,’fraud’]  
classes = pd.value\_counts(creditData[‘Class’], sort = True)  
classes.plot(kind = ‘bar’, rot=0)  
plt.title(“Transaction class distribution”)  
plt.xticks(range(2), labels)  
plt.xlabel(“Class”)  
plt.ylabel(“Frequency”)



Figure 2

fraud = creditData[creditData.Class == 1]  
normal = creditData[creditData.Class == 0]  
# Amount vs Class  
f, (ax1, ax2) = plt.subplots(2,1,sharex=True)  
f.suptitle('Amount per transaction by class')

ax1.hist(fraud.Amount, bins = 50)  
ax1.set\_title('Fraud List')

ax2.hist(normal.Amount, bins = 50)  
ax2.set\_title('Normal')

plt.xlabel('Amount')  
plt.ylabel('Number of Transactions')  
plt.xlim((0, 10000))  
plt.yscale('log')  
plt.show()

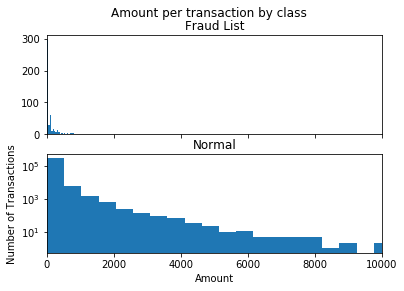


Figure 3

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

ax1.scatter(fraud.Time, fraud.Amount)  
ax1.set\_title('Fraud List')

ax2.scatter(normal.Time, normal.Amount)  
ax2.set\_title('Normal')

plt.xlabel('Time (in seconds)')  
plt.ylabel('Amount')  
plt.show()

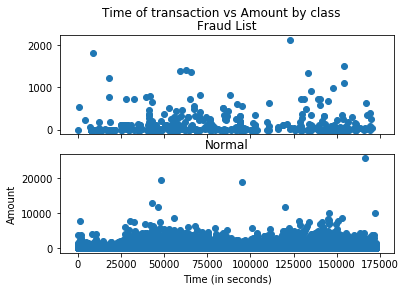


Figure 4

#plotting the dataset considering the class  
color = {1:'red', 0:'yellow'}  
fraudlist = creditData[creditData.Class == 1]  
normal = creditData[creditData.Class == 0]  
fig,axes = plt.subplots(1,2)

axes[0].scatter(list(range(1,fraudlist.shape[0] + 1)), fraudlist.Amount,color='red')  
axes[1].scatter(list(range(1, normal.shape[0] + 1)), normal.Amount,color='yellow')  
plt.show()

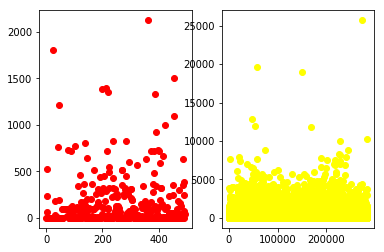


Figure 5: Frauds vs Normals

#### Preparing Data

The Time variable is not giving an impact on the model prediction. This can figure out from data visualization. Before moving on to the training part, we need to figure out which variables are important and which are not. So we can drop the unwanted variables.

features= creditData\_h2o.drop(['Time'], axis=1)

Split the data frame as training set and testing set keeping 80% for the training set and rest to the testing set.

train, test = features.split\_frame([0.8])  
print(train.shape)  
print(test.shape)

> (227722, 30)  
> (57085, 30)

Our dataset has a lot of non-fraud transactions. Because of this for the model training, we only send non-fraud transactions. So that the model will learn the pattern of normal transactions.

# converting to pandas dataframe  
train\_df = train.as\_data\_frame()  
test\_df = test.as\_data\_frame()

train\_df = train\_df[train\_df['Class'] == 0]  
# drop the Class variable  
train\_df = train\_df.drop(['Class'], axis=1)

Y\_test\_df = test\_df['Class'] # true labels of the testing set

test\_df = test\_df.drop(['Class'], axis=1)

train\_df.shape

> (227335, 29)

#### Model Building

train\_h2o = h2o.H2OFrame(train\_df) # converting to h2o frame  
test\_h2o = h2o.H2OFrame(test\_df)  
x = train\_h2o.columns

When building the model, 4 fully connected hidden layers were chosen with, [14,7,7,14] number of nodes for each layer. First two for the **encoder**and last two for the **decoder**.

anomaly\_model = H2ODeepLearningEstimator(activation = "Tanh",  
 hidden = [14,7,7,14],  
 epochs = 100,  
 standardize = True,  
 stopping\_metric = 'MSE',   
 loss = 'automatic',  
 train\_samples\_per\_iteration = 32,  
 shuffle\_training\_data = True,   
 autoencoder = True,  
 l1 = 10e-5)  
anomaly\_model.train(x=x, training\_frame = train\_h2o)

#### Model Evaluation

Variable Importance : In H2O there is a special way of analyzing which variables are giving higher impact on the model.

anomaly\_model.\_model\_json['output']['variable\_importances'].as\_data\_frame()

**Visualization**

# plotting the variable importance  
rcParams['figure.figsize'] = 14, 8  
#plt.rcdefaults()  
fig, ax = plt.subplots()  
  
variables = anomaly\_model.\_model\_json['output']['variable\_importances']['variable']  
var = variables[0:15]  
y\_pos = np.arange(len(var))  
  
scaled\_importance = anomaly\_model.\_model\_json['output']['variable\_importances']['scaled\_importance']  
sc = scaled\_importance[0:15]  
  
ax.barh(y\_pos, sc, align='center', color='green', ecolor='black')  
ax.set\_yticks(y\_pos)  
ax.set\_yticklabels(variables)  
ax.invert\_yaxis()  
ax.set\_xlabel('Scaled Importance')  
ax.set\_title('Variable Importance')  
plt.show()



Figure 6

# plotting the loss  
scoring\_history = anomaly\_model.score\_history()  
%matplotlib inline  
rcParams['figure.figsize'] = 14, 8  
plt.plot(scoring\_history['training\_mse'])  
plt.title('model loss')  
plt.ylabel('loss')  
plt.xlabel('epoch')

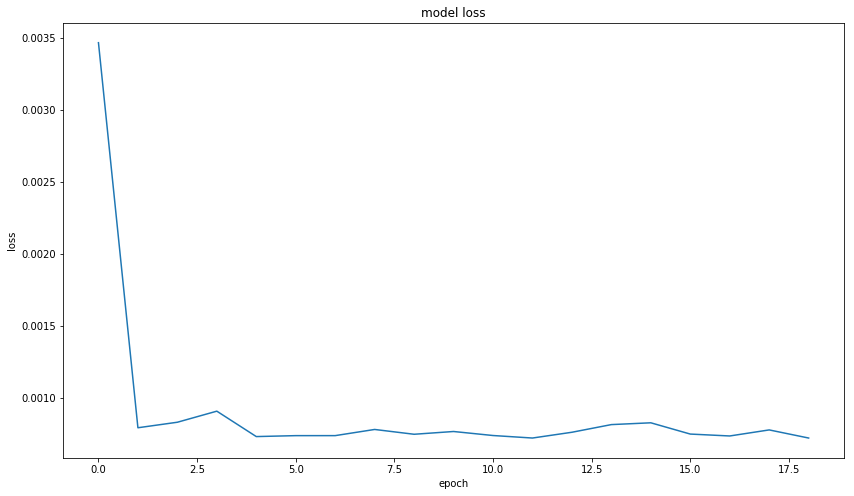


Figure 7

**The testing set** has both normal and fraud transactions in it. The Autoencoder will learn to identify the pattern of the input data. If an anomalous test point does not match the learned pattern, the autoencoder will likely have a high error rate in reconstructing this data, indicating anomalous data. So that we can identify the anomalies of the data. To calculate the error, it uses **Mean Squared Error**(MSE)

test\_rec\_error = anomaly\_model.anomaly(test\_h2o)   
# anomaly is a H2O function which calculates the error for the dataset

# converting to pandas dataframe  
test\_rec\_error\_df = test\_rec\_error.as\_data\_frame()

# plotting the testing dataset against the error  
test\_rec\_error\_df['id']=test\_rec\_error\_df.index  
rcParams['figure.figsize'] = 14, 8  
test\_rec\_error\_df.plot(kind="scatter", x='id', y="Reconstruction.MSE")  
plt.show()

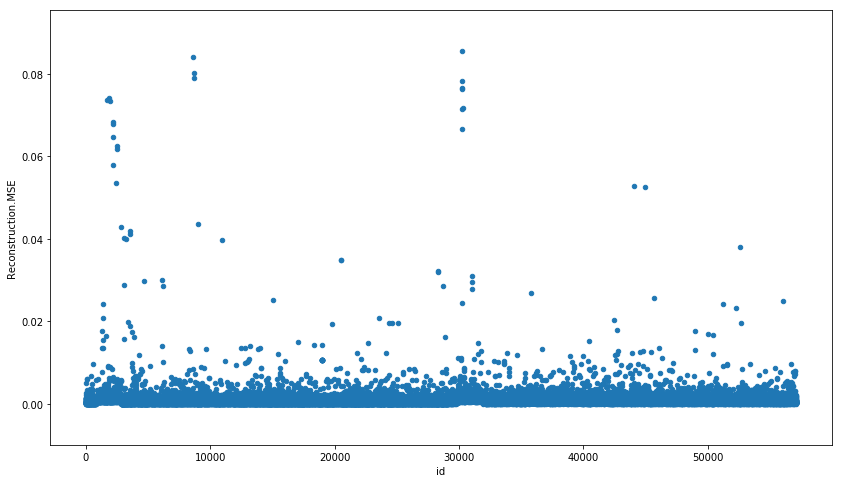


Figure 8: Anomalies in testing set

# predicting the class for the testing dataset  
predictions = anomaly\_model.predict(test\_h2o)

error\_df = pd.DataFrame({'reconstruction\_error': test\_rec\_error\_df['Reconstruction.MSE'],  
 'true\_class': Y\_test\_df})  
error\_df.describe()

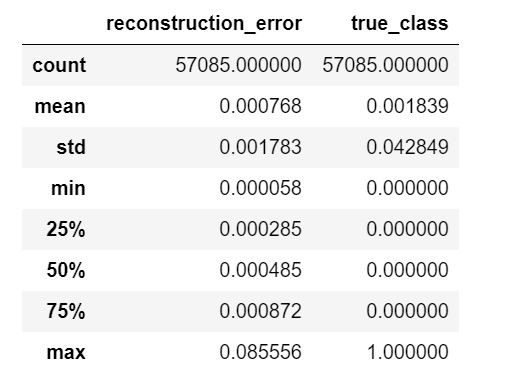


Figure 9

# reconstruction error for the normal transactions in the testing dataset  
fig = plt.figure()  
ax = fig.add\_subplot(111)  
rcParams['figure.figsize'] = 14, 8  
normal\_error\_df = error\_df[(error\_df['true\_class']== 0) & (error\_df['reconstruction\_error'] < 10)]  
\_ = ax.hist(normal\_error\_df.reconstruction\_error.values, bins=10)



Figure 10

# reconstruction error for the fraud transactions in the testing dataset  
fig = plt.figure()  
ax = fig.add\_subplot(111)  
rcParams['figure.figsize'] = 14, 8  
fraud\_error\_df = error\_df[error\_df['true\_class'] == 1]  
\_ = ax.hist(fraud\_error\_df.reconstruction\_error.values, bins=10)

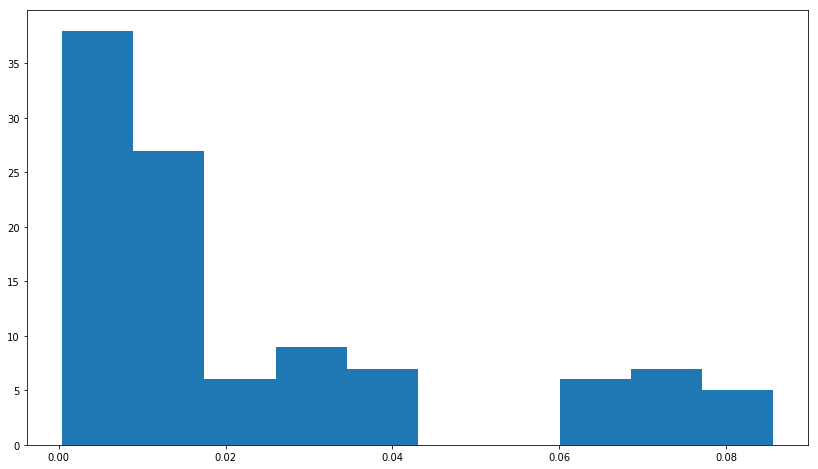


Figure 11

#### ROC Curve

from sklearn.metrics import (confusion\_matrix, precision\_recall\_curve, auc,  
 roc\_curve, recall\_score, classification\_report, f1\_score,  
 precision\_recall\_fscore\_support)  
fpr, tpr, thresholds = roc\_curve(error\_df.true\_class, error\_df.reconstruction\_error)  
roc\_auc = auc(fpr, tpr)

plt.title('Receiver Operating Characteristic')  
plt.plot(fpr, tpr, label='AUC = %0.4f'% roc\_auc)  
plt.legend(loc='lower right')  
plt.plot([0,1],[0,1],'r--')  
plt.xlim([-0.001, 1])  
plt.ylim([0, 1.001])  
plt.ylabel('True Positive Rate')  
plt.xlabel('False Positive Rate')  
plt.show();

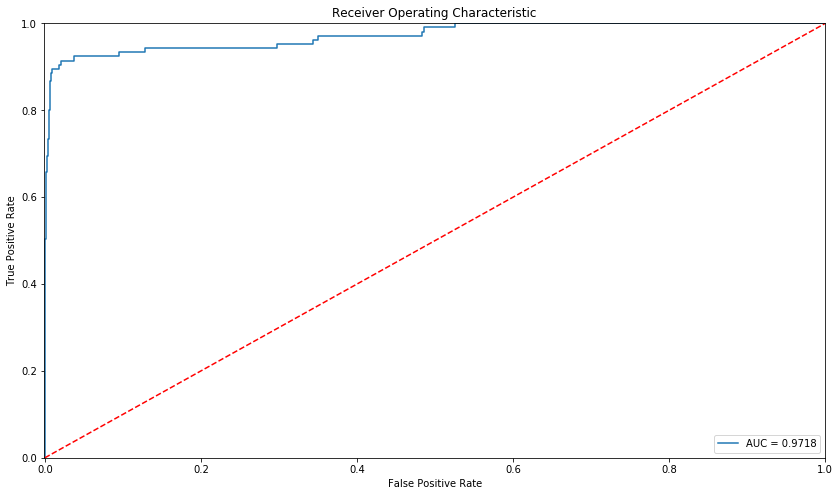


Figure 12

The **accuracy** is **0.9718**

#### Precision & Recall

Since the data is highly imbalanced, it cannot be measured only by using accuracy. Precision vs Recall was chosen as the matrix for the classification task.

**Precision**: Measuring the relevancy of obtained results.

[ True positives / (True positives + False positives)]

**Recall**: Measuring how many relevant results are returned.

[ True positives / (True positives + False negatives)]

**True Positives** — Number of actual frauds predicted as frauds

**False Positives** — Number of non-frauds predicted as frauds

**False Negatives** — Number of frauds predicted as non-frauds.

precision, recall, th = precision\_recall\_curve(error\_df.true\_class, error\_df.reconstruction\_error)  
plt.plot(recall, precision, 'b', label='Precision-Recall curve')  
plt.title('Recall vs Precision')  
plt.xlabel('Recall')  
plt.ylabel('Precision')  
plt.show()

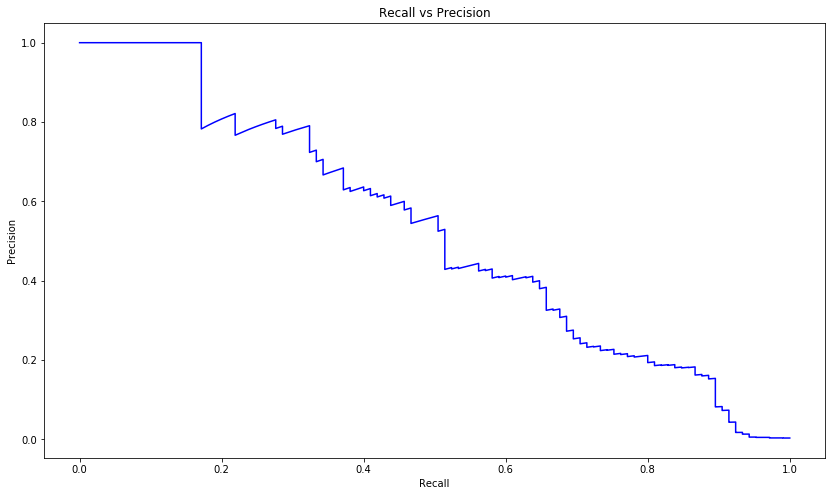


Figure 13

We need to find a better threshold that can separate the anomalies from normal. This can be done by calculating the intersection of the **Precision/Recall vs Threshold** graph.

plt.plot(th, precision[1:], label="Precision",linewidth=5)  
plt.plot(th, recall[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()

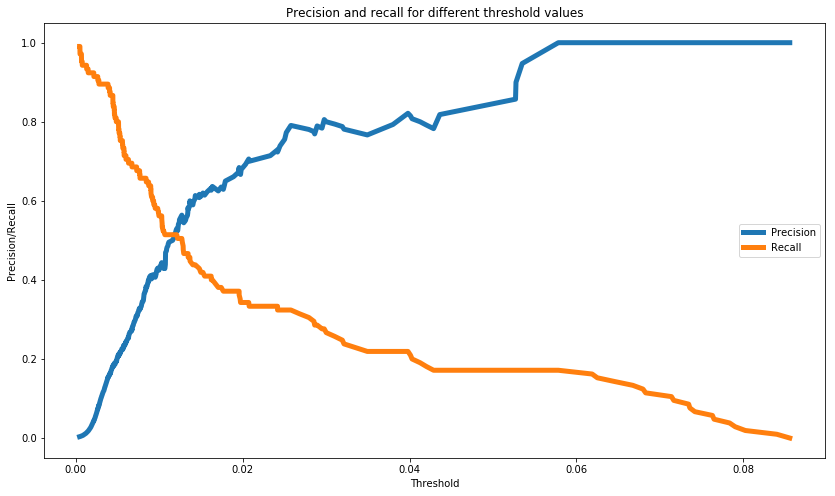


Figure 14

# plot the testing set with the threshold  
threshold = 0.01  
groups = error\_df.groupby('true\_class')  
fig, ax = plt.subplots()

for name, group in groups:  
 ax.plot(group.index, group.reconstruction\_error, marker='o', ms=3.5, linestyle='',  
 label= "Fraud" if name == 1 else "Normal")  
ax.hlines(threshold, ax.get\_xlim()[0], ax.get\_xlim()[1], colors="r", zorder=100, label='Threshold')  
ax.legend()  
plt.title("Reconstruction error for different classes")  
plt.ylabel("Reconstruction error")  
plt.xlabel("Data point index")  
plt.show();

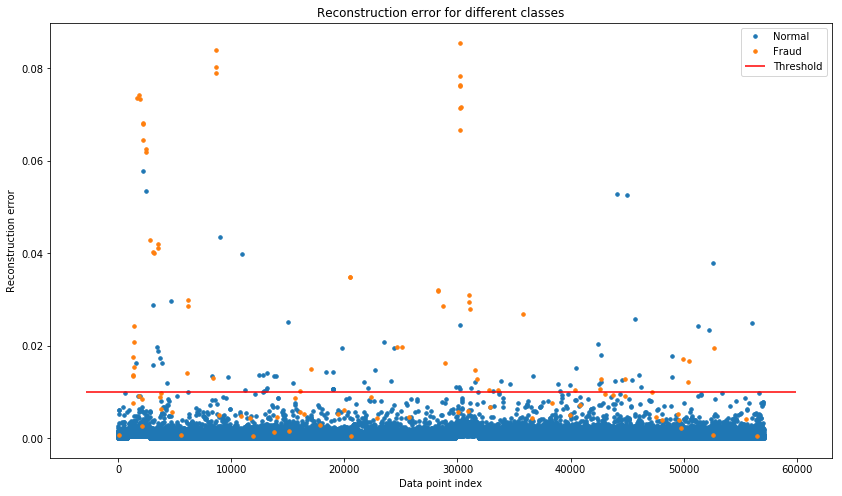


Figure 15

#### Confusion Matrix

import seaborn as sns  
LABELS = ['Normal', 'Fraud']  
y\_pred = [1 if e > threshold else 0 for e in error\_df.reconstruction\_error.values]  
conf\_matrix = confusion\_matrix(error\_df.true\_class, y\_pred)  
plt.figure(figsize=(12, 12))  
sns.heatmap(conf\_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d");  
plt.title("Confusion matrix")  
plt.ylabel('True class')  
plt.xlabel('Predicted class')  
plt.show()

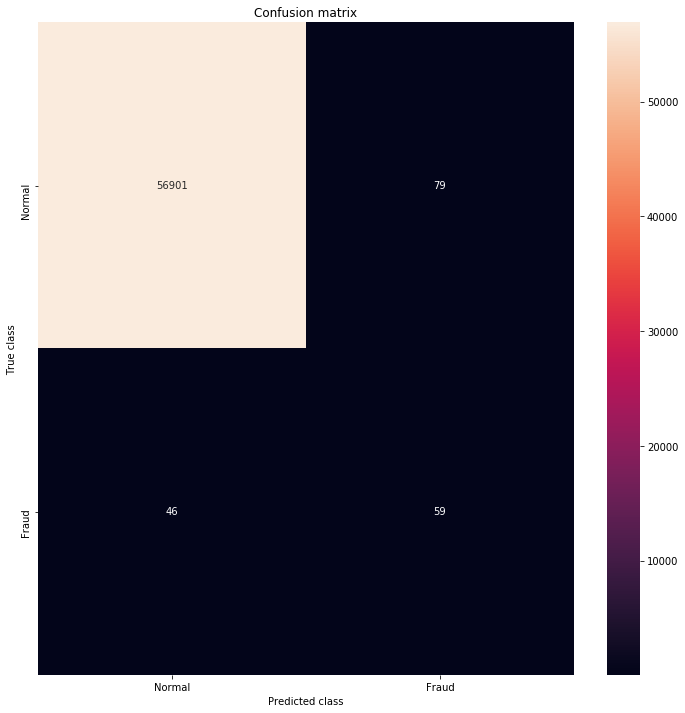


Figure 16

#### Classification Report

csr = classification\_report(error\_df.true\_class, y\_pred)  
print(csr)

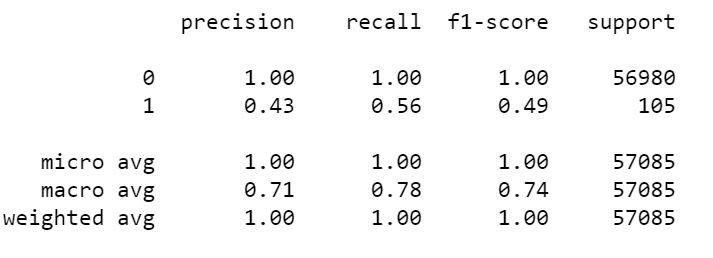


Figure 17

### Conclusion

Our model is catching most of the fraudulent data. In Autoencoders, it gives a good accuracy. But if we look into Precision and Recall of the dataset, it is not performing enough. As I mentioned earlier, there are other anomaly detection methods that perform well in highly imbalanced datasets.

I have tried more methods on this dataset. So I will see you soon with those. :)

#### References

<https://medium.com/@curiousily/credit-card-fraud-detection-using-autoencoders-in-keras-tensorflow-for-hackers-part-vii-20e0c85301bd>