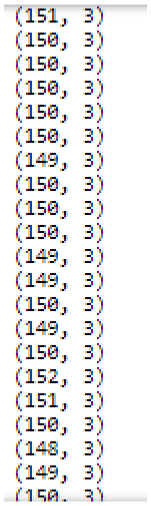
NN Keras

For accelerometer classification

File approaching

The first problem to deal with for our Neural Network is to understand properly the data, which is noisy and hard to understand due to the its form and its size which varies in size from (136,3) – (159,3) .

There are many ways to deal with this problem like oversampling with more data to not lose any additional and maybe important data for our classification. I have chosen to do under sampling, taking the minimum of each file to have more accurate data.



```
(151, 3)
(150, 3)
(150, 3)
(150, 3)
(150, 3)
(150, 3)
(149, 3)
(150, 3)
(150, 3)
(150, 3)
(149, 3)
(149, 3)
(150, 3)
(149, 3)
(150, 3)
(152, 3)
(151, 3)
(150, 3)
(148, 3)
(149, 3)
(150, 3)
```

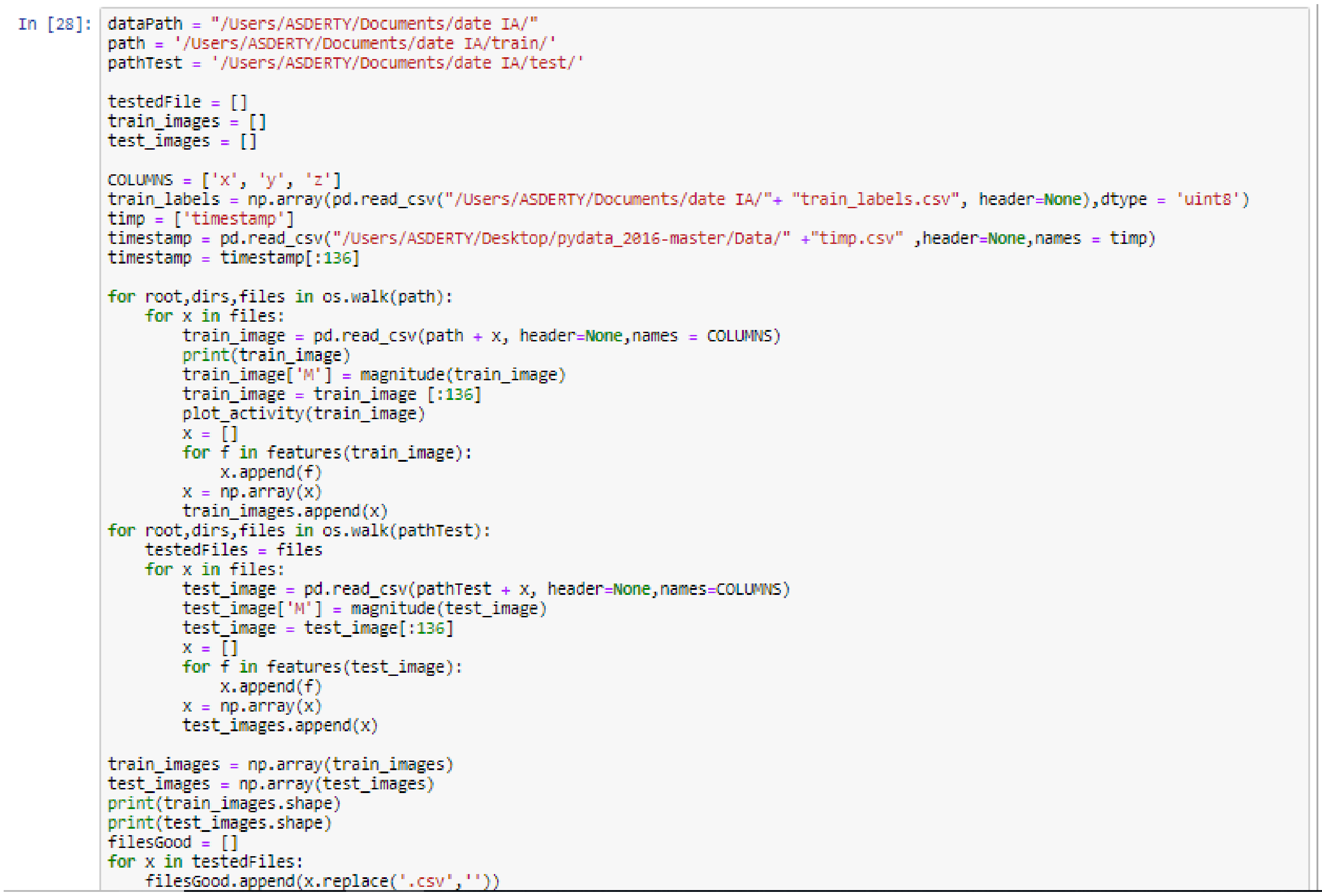
This is a short example for the csv files sizes which varies.

Now that we have chosen a way to tackle the inconsistency of the data sizes we can now deal with the files.

First we have to read all the files of train 9000 and 5000 test.

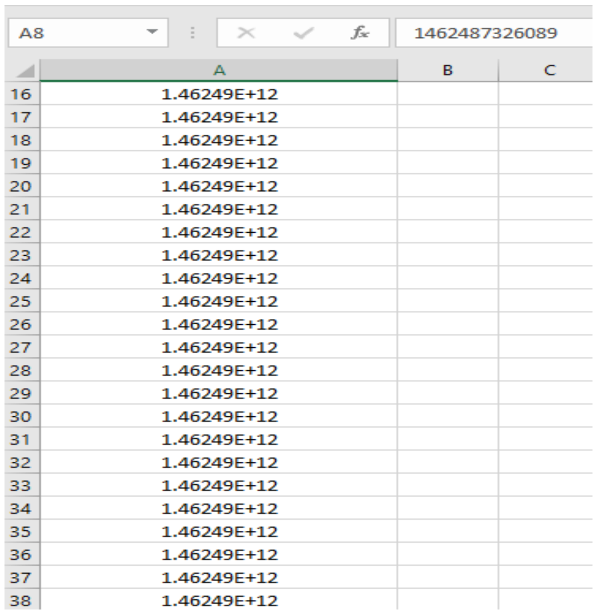
First for this matter, the accelerometer data it was read in 1.5 seconds, I have created a file that imitates approximately the time interval for more accurate data recognition and for processing it. Which will see it later

Here we have it, I have used panda and numpy for processing and opening the data for a more conveniant approach and a easier use and handling.



```
In [28]: dataPath = "/Users/ASDERTY/Documents/date IA/"
         path = '/Users/ASDERTY/Documents/date IA/train/'
         pathTest = '/Users/ASDERTY/Documents/date IA/test/'

         testedFile = []
         train_images = []
         test_images = []

         COLUMNS = ['x', 'y', 'z']
         train_labels = np.array(pd.read_csv("/Users/ASDERTY/Documents/date IA/"+ "train_labels.csv", header=None),dtype = 'uint8')
         timp = ['timestamp']
         timestamp = pd.read_csv("/Users/ASDERTY/Desktop/pydata_2016-master/Data/" +"timp.csv" ,header=None,names = timp)
         timestamp = timestamp[:136]

         for root,dirs,files in os.walk(path):
             for x in files:
                 train_image = pd.read_csv(path + x, header=None,names = COLUMNS)
                 print(train_image)
                 train_image['M'] = magnitude(train_image)
                 train_image = train_image [:136]
                 plot_activity(train_image)
                 x = []
                 for f in features(train_image):
                     x.append(f)
                 x = np.array(x)
                 train_images.append(x)
         for root,dirs,files in os.walk(pathTest):
             testedFiles = files
             for x in files:
                 test_image = pd.read_csv(pathTest + x, header=None,names=COLUMNS)
                 test_image['M'] = magnitude(test_image)
                 test_image = test_image[:136]
                 x = []
                 for f in features(test_image):
                     x.append(f)
                 x = np.array(x)
                 test_images.append(x)

         train_images = np.array(train_images)
         test_images = np.array(test_images)
         print(train_images.shape)
         print(test_images.shape)
         filesGood = []
         for x in testedFiles:
             filesGood.append(x.replace('.csv',''))
```

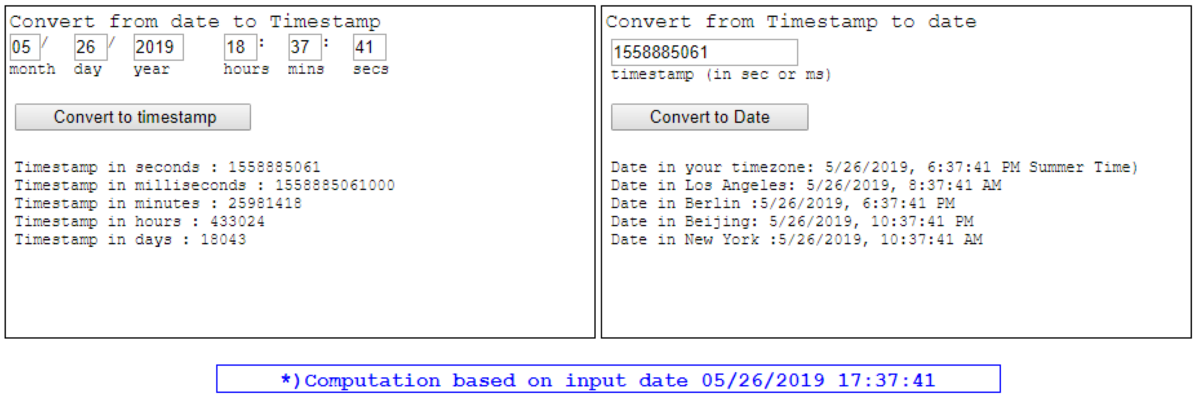
Now we see that theres something different, about the timestamp file which is a pure copy of maybe 2 seconds intervals for accelerometer data which looks like this.



A8 | 1462487326089

	A	B	C
16	1.46249E+12		
17	1.46249E+12		
18	1.46249E+12		
19	1.46249E+12		
20	1.46249E+12		
21	1.46249E+12		
22	1.46249E+12		
23	1.46249E+12		
24	1.46249E+12		
25	1.46249E+12		
26	1.46249E+12		
27	1.46249E+12		
28	1.46249E+12		
29	1.46249E+12		
30	1.46249E+12		
31	1.46249E+12		
32	1.46249E+12		
33	1.46249E+12		
34	1.46249E+12		
35	1.46249E+12		
36	1.46249E+12		
37	1.46249E+12		
38	1.46249E+12		

The format is the android timestamp format for data which can be converted right here to understand its format https://www.timestampconvert.com/



```
Convert from date to Timestamp
05 / 26 / 2019    18 : 37 : 41
month day year    hours mins secs

Timestamp in seconds : 1558885061
Timestamp in milliseconds : 1558885061000
Timestamp in minutes : 25981418
Timestamp in hours : 433024
Timestamp in days : 18043
```

```
Convert from Timestamp to date
1558885061
timestamp (in sec or ms)

Date in your timezone: 5/26/2019, 6:37:41 PM Summer Time)
Date in Los Angeles: 5/26/2019, 8:37:41 AM
Date in Berlin :5/26/2019, 6:37:41 PM
Date in Beijing: 5/26/2019, 10:37:41 PM
Date in New York :5/26/2019, 10:37:41 AM
```

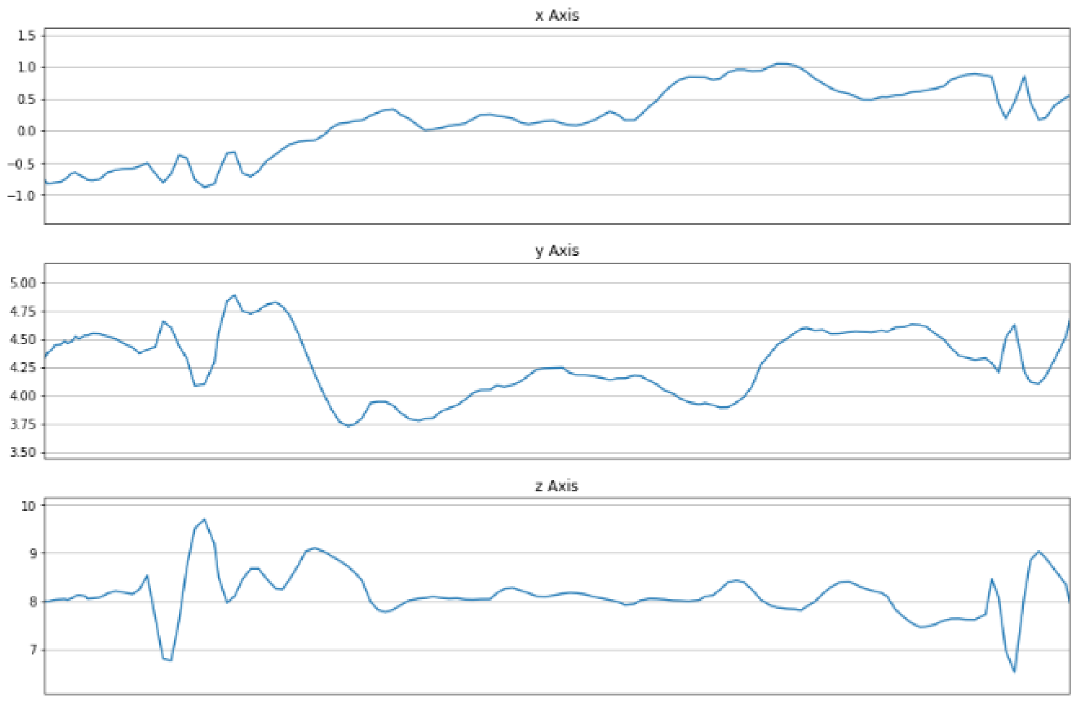
```
*)Computation based on input date 05/26/2019 17:37:41
```

And here we have a timestamp for the 26/5/2019 date which this documentation is written .Wow.

Anyway parsing through the data we came to a end and our x,y and z data looks like this independantly

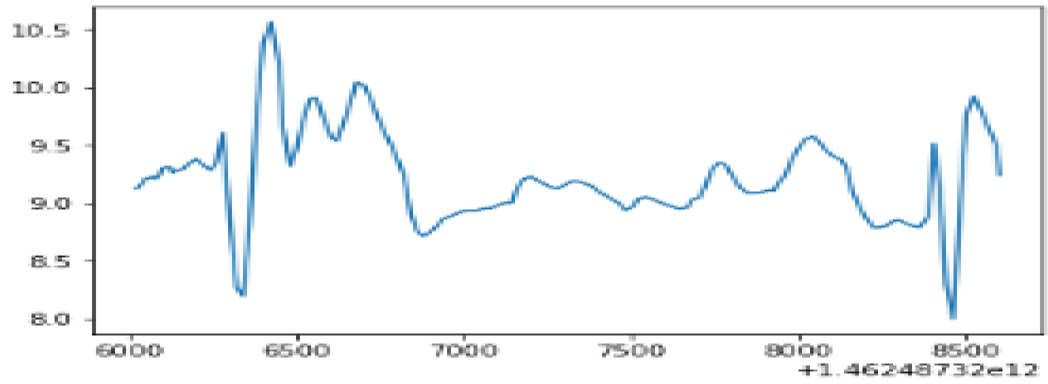
Tackling data

This is the first file displaying the axis independently



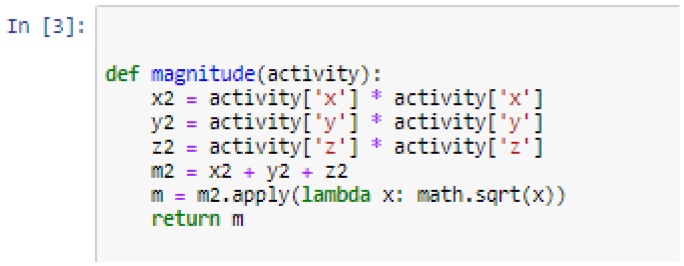
We can observe that the values for each axis are pretty diferent from each other.

We can use mathematical formula to combine them, magnitude it is like combining the three lines into one with the formula of the distance of a point to another which is magnitude = $\sqrt{x^2 + y^2 + z^2}$



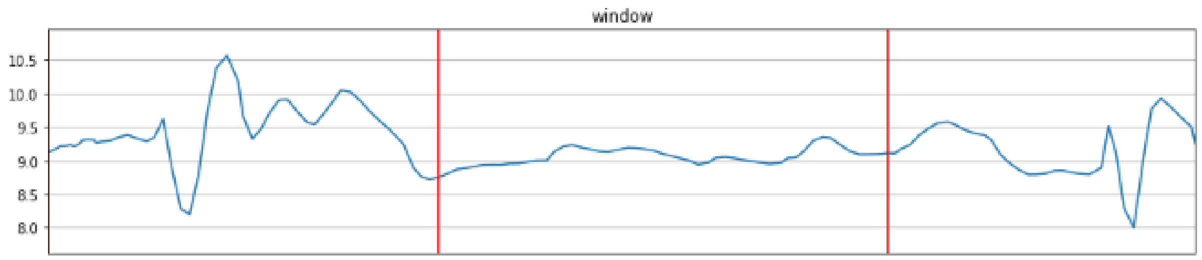
Pretty life saver, we combined three lines into one, which is more easier for our data, but it is enough ?

Here is the formula for magnitude

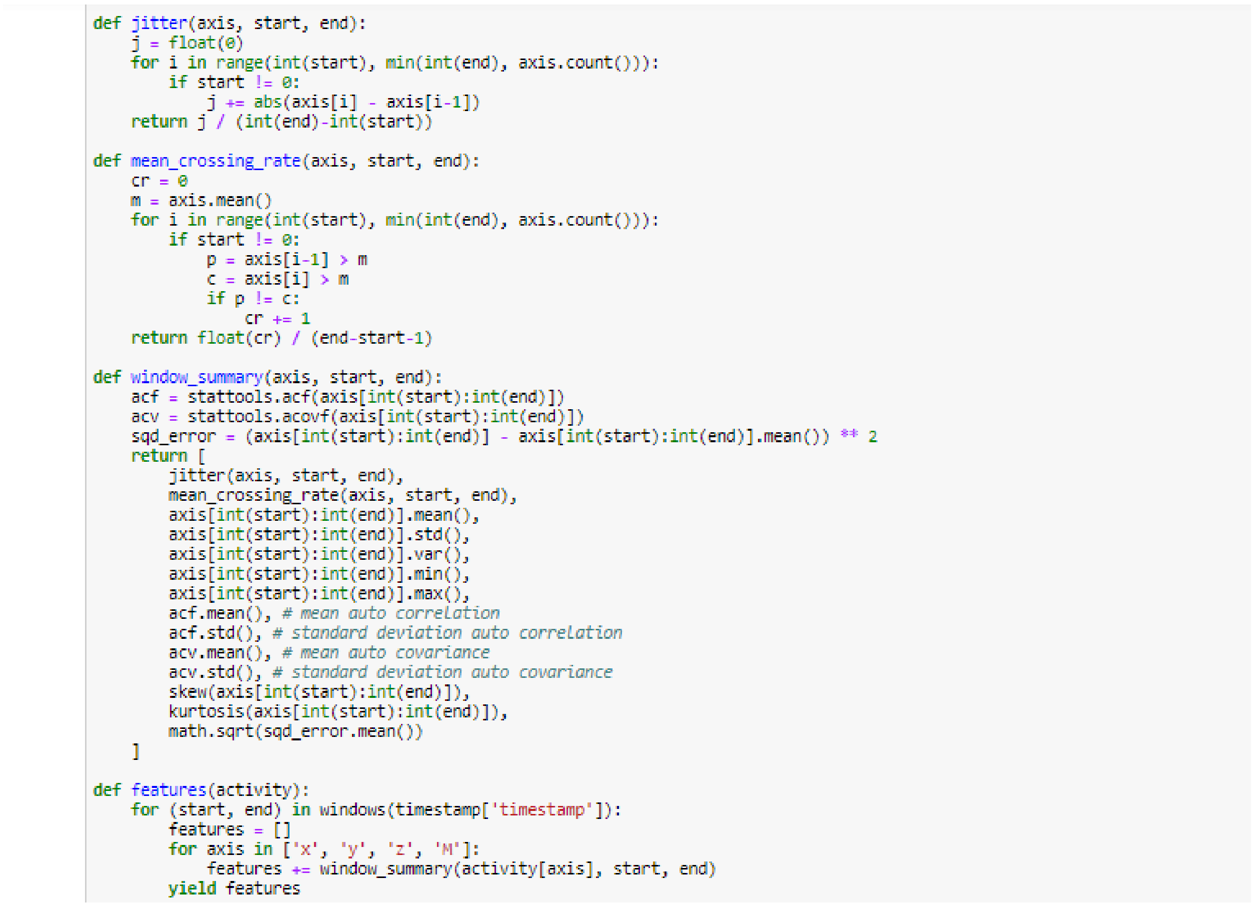


```
In [3]:

def magnitude(activity):
    x2 = activity['x'] * activity['x']
    y2 = activity['y'] * activity['y']
    z2 = activity['z'] * activity['z']
    m2 = x2 + y2 + z2
    m = m2.apply(lambda x: math.sqrt(x))
    return m
```

Not enough, we can breakeven the interval for the wave signal, so we use a window function of the intervals from the magnitude

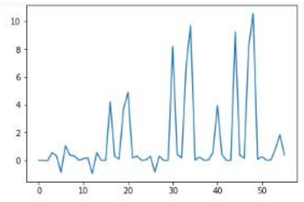


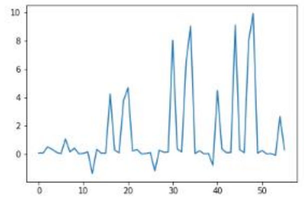
After dividing we apply mathematical function like jitter, mean_crossing_rate, mean,standard, squared mean error ,deviation, variation, min, max, statt tools, skew, kurtosis etc. Well, a lot a functions for the data, I got this idea from pydata 2016, which was presented a method for clasifying user states based on accelerometer which applies those functions on windows of data.

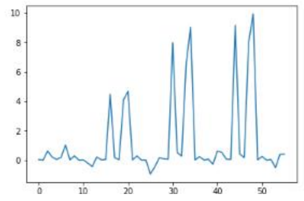


```
def jitter(axis, start, end):
    j = float(0)
    for i in range(int(start), min(int(end), axis.count())):
        if start != 0:
            j += abs(axis[i] - axis[i-1])
    return j / (int(end)-int(start))

def mean_crossing_rate(axis, start, end):
    cr = 0
    m = axis.mean()
    for i in range(int(start), min(int(end), axis.count())):
        if start != 0:
            p = axis[i-1] > m
            c = axis[i] > m
            if p != c:
                cr += 1
    return float(cr) / (end-start-1)

def window_summary(axis, start, end):
    acf = stattools.acf(axis[int(start):int(end)])
    acv = stattools.acovf(axis[int(start):int(end)])
    sqd_error = (axis[int(start):int(end)] - axis[int(start):int(end)].mean()) ** 2
    return [
        jitter(axis, start, end),
        mean_crossing_rate(axis, start, end),
        axis[int(start):int(end)].mean(),
        axis[int(start):int(end)].std(),
        axis[int(start):int(end)].var(),
        axis[int(start):int(end)].min(),
        axis[int(start):int(end)].max(),
        acf.mean(), # mean auto correlation
        acf.std(), # standard deviation auto correlation
        acv.mean(), # mean auto covariance
        acv.std(), # standard deviation auto covariance
        skew(axis[int(start):int(end)]),
        kurtosis(axis[int(start):int(end)]),
        math.sqrt(sqd_error.mean())
    ]

def features(activity):
    for (start, end) in windows(timestamp['timestamp']):
        features = []
        for axis in ['x', 'y', 'z', 'M']:
            features += window_summary(activity[axis], start, end)
        yield features
```

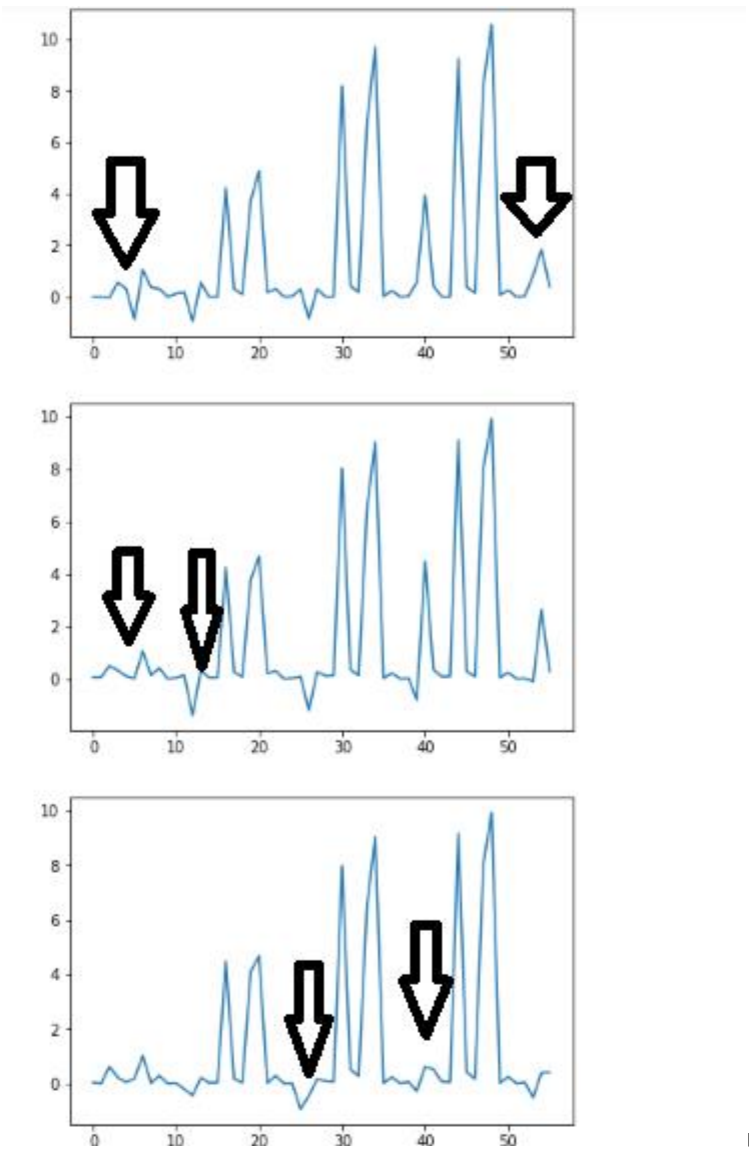
Pretty complicated, the basic idea it is that we can have more uniques properties for a better classifcation for the timestamps. I will add the video in the bibliografy.







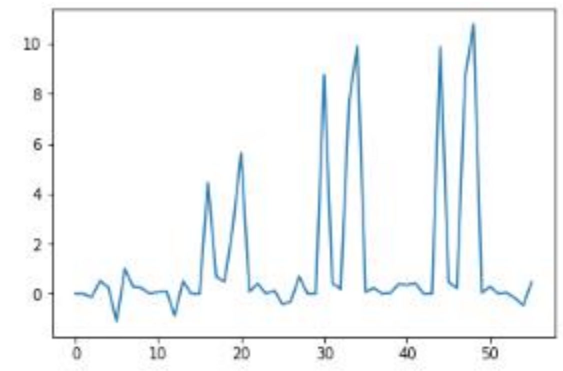
They kinda look the same right ?



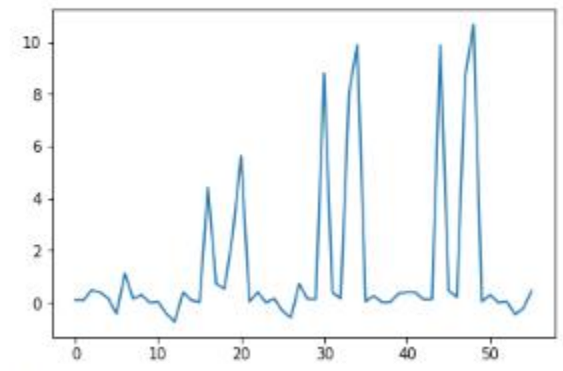
Nope, here are the differences not all but, some of them.

Which is very good because it would be easier to understand the differences for classifying

Here is a example for two datasets with the same labels to see the differences in clasifying



```
[1]
```



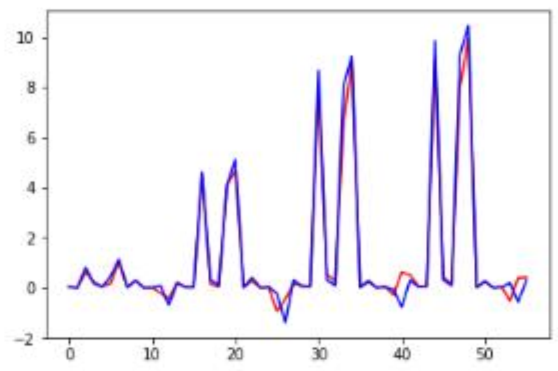
```
[1]
```

Wow, very look alike

We loaded the data and got this

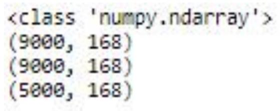
```
(9000, 3, 56)
(5000, 3, 56)
```

```
In [40]: plt.plot(train_images[0][2],'r')
         plt.plot(train_images[1][2],'b')
         plt.show()
```

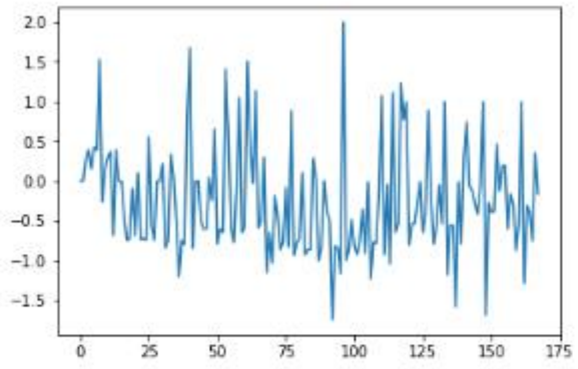


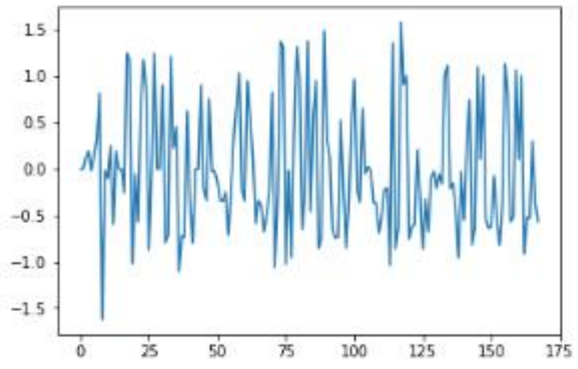
3 is the number of windows and 56 are the unique features from creating unique features functions for each file-per window

Standardized the files to be easier to load and flatten the data which gives us



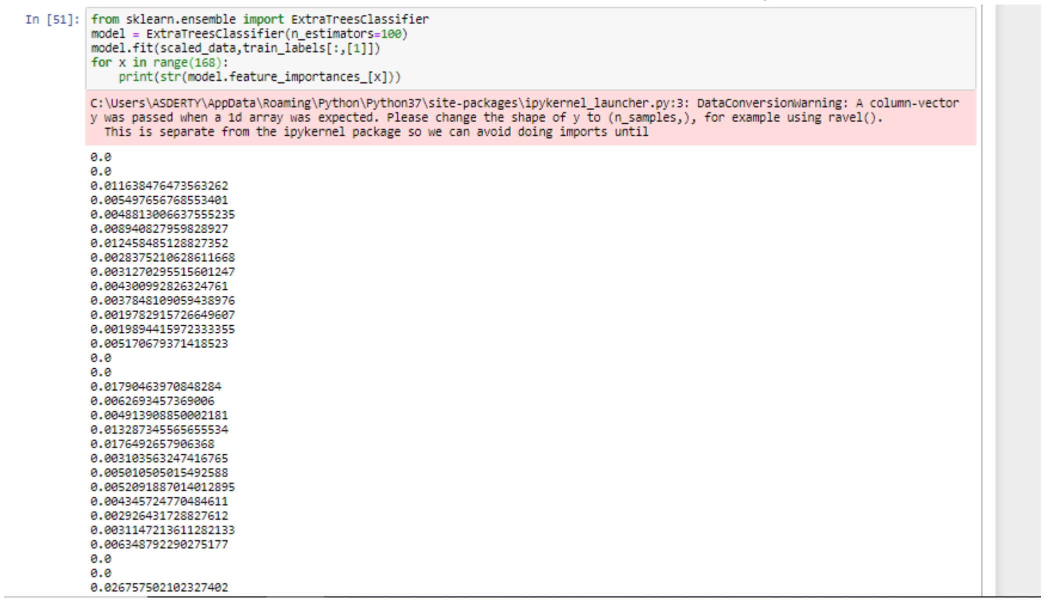
```
<class 'numpy.ndarray'>
(9000, 168)
(9000, 168)
(5000, 168)
```





Features importance

Cool, now we have our data scaled and flat now lets see the importances of the features that we have



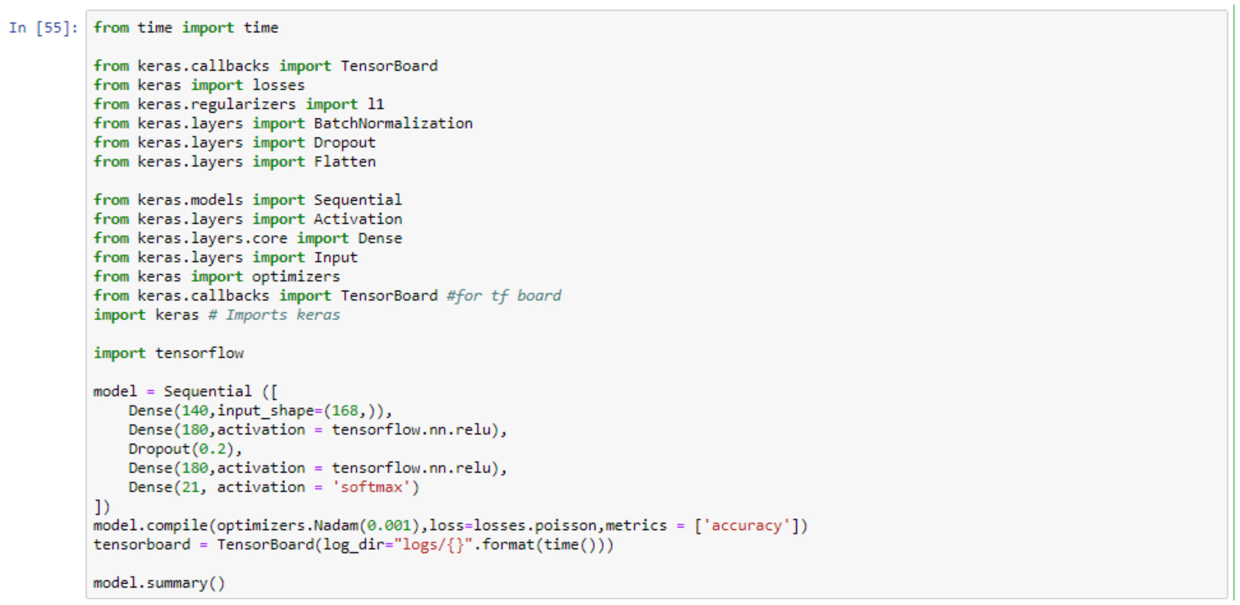
```
In [51]: from sklearn.ensemble import ExtraTreesClassifier
         model = ExtraTreesClassifier(n_estimators=100)
         model.fit(scaled_data,train_labels[:,[1]])
         for x in range(168):
             print(str(model.feature_importances_[x]))
```

```
C:\Users\ASDERTY\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.py:3: DataConversionWarning: A column-vector
y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using ravel().
  This is separate from the ipykernel package so we can avoid doing imports until
```

```
0.0
0.0
0.011638476473563262
0.005497656768553401
0.004881300663755235
0.008940827959828927
0.012458485128827352
0.0028375210628611668
0.0031270295515601247
0.004300992826324761
0.0037848109059438976
0.0019782915726649607
0.0019894415972333355
0.005170679371418523
0.0
0.0
0.01790463970848284
0.0062693457369006
0.004913908850002181
0.013287345565655534
0.0176492657906368
0.003103563247416765
0.005010505015492588
0.0052091887014012895
0.004345724770484611
0.002926431728827612
0.0031147213611282133
0.006348792290275177
0.0
0.0
0.026757502102327402
```

Well, some features are not that useful it seems, the extras trees clasifier made the importances on the data but for safety reason, I would like to keep them because it can affect the graph

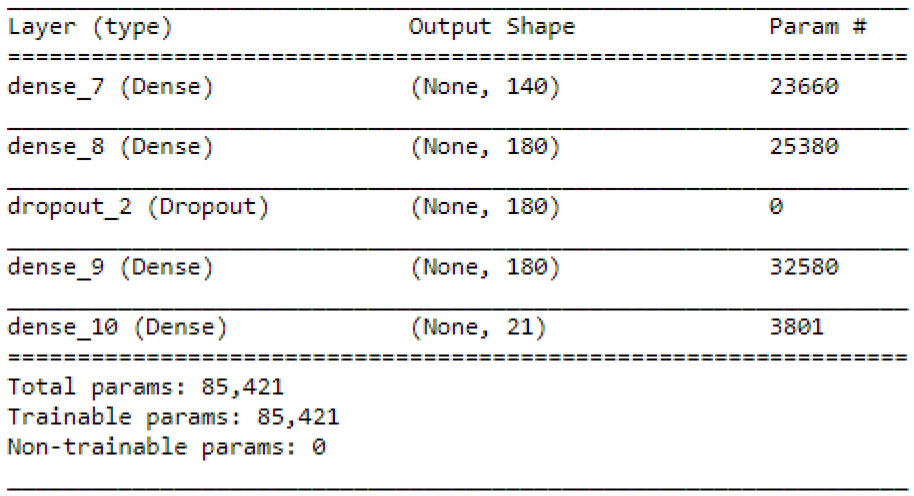
Now the interesting part , the Keras Neural Network with Tensorflow activation, here is my model of choice



```
In [55]: from time import time

         from keras.callbacks import TensorBoard
         from keras import losses
         from keras.regularizers import l1
         from keras.layers import BatchNormalization
         from keras.layers import Dropout
         from keras.layers import Flatten

         from keras.models import Sequential
         from keras.layers import Activation
         from keras.layers.core import Dense
         from keras.layers import Input
         from keras import optimizers
         from keras.callbacks import TensorBoard #for tf board
         import keras # Imports keras

         import tensorflow

         model = Sequential ([
             Dense(140,input_shape=(168,)),
             Dense(180,activation = tensorflow.nn.relu),
             Dropout(0.2),
             Dense(180,activation = tensorflow.nn.relu),
             Dense(21, activation = 'softmax')
         ])
         model.compile(optimizers.Nadam(0.001),loss=losses.poisson,metrics = ['accuracy'])
         tensorboard = TensorBoard(log_dir="logs/{}".format(time()))

         model.summary()
```

Well, I have choosen a bigger model to hold more param to learn from them and apply on the next set of data, I have choosen as activation relu(rectified liniar unit), because it seems relu was my best shot for it and used Dropout for reducing overfiting of data.

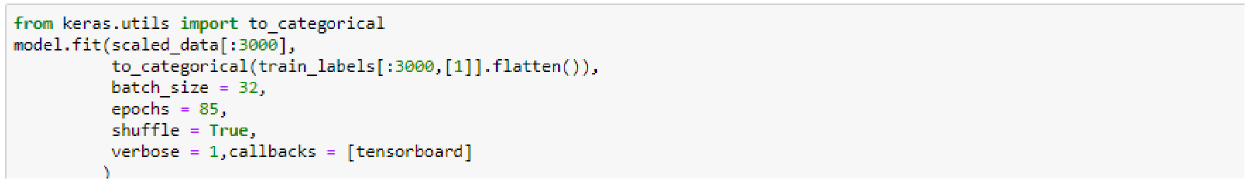
As compiler I choose Nadam due to its performance on my data sets and lr = 0.001 to be better on the gradient descending and for loss function chose poisson due to its performance on the data



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 140)               23660     
_________________________________________________________________
dense_8 (Dense)              (None, 180)               25380     
_________________________________________________________________
dropout_2 (Dropout)          (None, 180)               0         
_________________________________________________________________
dense_9 (Dense)              (None, 180)               32580     
_________________________________________________________________
dense_10 (Dense)             (None, 21)                3801      
=================================================================
Total params: 85,421
Trainable params: 85,421
Non-trainable params: 0
_________________________________________________________________
```

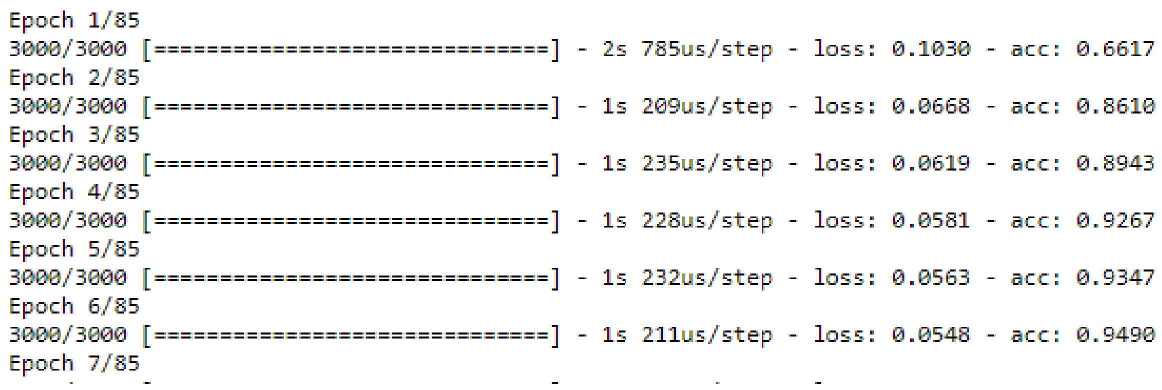
Pretty neat for my task

Now the training part, I used as first subset the first 3000 data and started training on it with 85 epoch and batch size of 32 for better understanding, used keras to_categorical for labeling the labels as hot ones enconding for the model to be able to run.



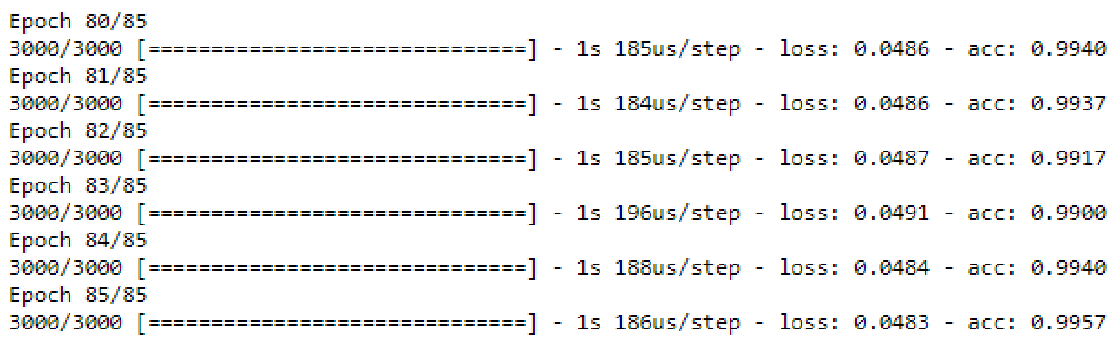
```
from keras.utils import to_categorical
model.fit(scaled_data[:3000],
          to_categorical(train_labels[:3000,[1]].flatten()),
          batch_size = 32,
          epochs = 85,
          shuffle = True,
          verbose = 1,callbacks = [tensorboard]
         )
```

And here it starts



```
Epoch 1/85
3000/3000 [==============================] - 2s 785us/step - loss: 0.1030 - acc: 0.6617
Epoch 2/85
3000/3000 [==============================] - 1s 209us/step - loss: 0.0668 - acc: 0.8610
Epoch 3/85
3000/3000 [==============================] - 1s 235us/step - loss: 0.0619 - acc: 0.8943
Epoch 4/85
3000/3000 [==============================] - 1s 228us/step - loss: 0.0581 - acc: 0.9267
Epoch 5/85
3000/3000 [==============================] - 1s 232us/step - loss: 0.0563 - acc: 0.9347
Epoch 6/85
3000/3000 [==============================] - 1s 211us/step - loss: 0.0548 - acc: 0.9490
Epoch 7/85
```

Pretty well for 7 epochs and for the final epoch

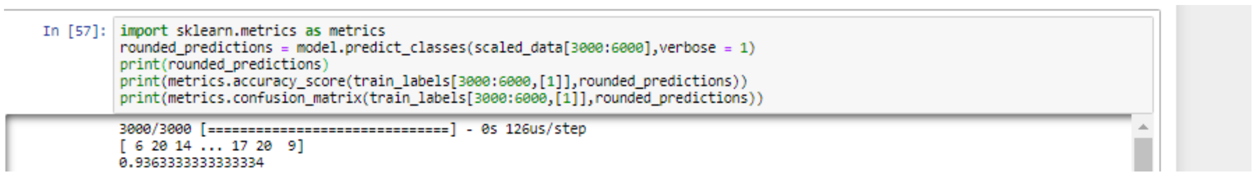


```
Epoch 80/85
3000/3000 [==============================] - 1s 185us/step - loss: 0.0486 - acc: 0.9940
Epoch 81/85
3000/3000 [==============================] - 1s 184us/step - loss: 0.0486 - acc: 0.9937
Epoch 82/85
3000/3000 [==============================] - 1s 185us/step - loss: 0.0487 - acc: 0.9917
Epoch 83/85
3000/3000 [==============================] - 1s 196us/step - loss: 0.0491 - acc: 0.9900
Epoch 84/85
3000/3000 [==============================] - 1s 188us/step - loss: 0.0484 - acc: 0.9940
Epoch 85/85
3000/3000 [==============================] - 1s 186us/step - loss: 0.0483 - acc: 0.9957
```

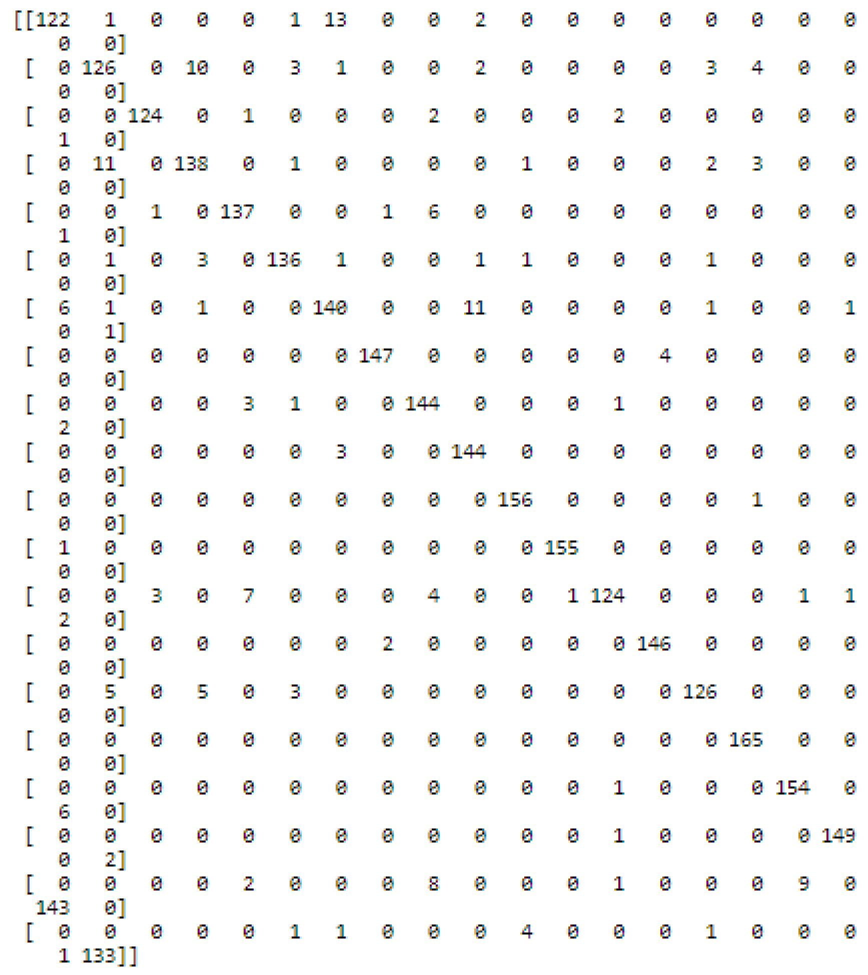
That's it awesome, %99.5

Lets test on the data 3000:6000, 6000:9000

First set we see

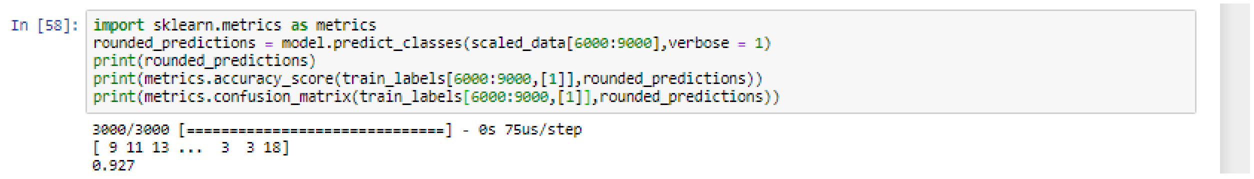


```
In [57]: import sklearn.metrics as metrics
         rounded_predictions = model.predict_classes(scaled_data[3000:6000],verbose = 1)
         print(rounded_predictions)
         print(metrics.accuracy_score(train_labels[3000:6000,[1]],rounded_predictions))
         print(metrics.confusion_matrix(train_labels[3000:6000,[1]],rounded_predictions))

         3000/3000 [==============================] - 0s 126us/step
         [ 6 20 14 ... 17 20  9]
         0.9363333333333334
```



```
[[122   1   0   0   0   1  13   0   0   2   0   0   0   0   0   0   0   0
    0   0]
 [  0 126   0  10   0   3   1   0   0   2   0   0   0   0   3   4   0   0
    0   0]
 [  0   0 124   0   1   0   0   0   2   0   0   0   2   0   0   0   0   0
    1   0]
 [  0  11   0 138   0   1   0   0   0   0   1   0   0   0   2   3   0   0
    0   0]
 [  0   0   1   0 137   0   0   1   6   0   0   0   0   0   0   0   0   0
    1   0]
 [  0   1   0   3   0 136   1   0   0   1   1   0   0   0   1   0   0   0
    0   0]
 [  6   1   0   1   0   0 140   0   0  11   0   0   0   0   1   0   0   1
    0   1]
 [  0   0   0   0   0   0   0 147   0   0   0   0   0   4   0   0   0   0
    0   0]
 [  0   0   0   0   3   1   0   0 144   0   0   0   1   0   0   0   0   0
    2   0]
 [  0   0   0   0   0   0   3   0   0 144   0   0   0   0   0   0   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0 156   0   0   0   0   1   0   0
    0   0]
 [  1   0   0   0   0   0   0   0   0   0   0 155   0   0   0   0   0   0
    0   0]
 [  0   0   3   0   7   0   0   0   4   0   0   1 124   0   0   0   1   1
    2   0]
 [  0   0   0   0   0   0   0   2   0   0   0   0   0 146   0   0   0   0
    0   0]
 [  0   5   0   5   0   3   0   0   0   0   0   0   0   0 126   0   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 165   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0 154   0
    6   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0 149
    0   2]
 [  0   0   0   0   2   0   0   0   8   0   0   0   1   0   0   0   9   0
  143   0]
 [  0   0   0   0   0   1   1   0   0   0   4   0   0   0   1   0   0   0
    1 133]]
```

Oh well, that’s %93.6, that’s really nice and the confussion matrix looks awesome, having a little error for each label itself now lets predict 6000:9000



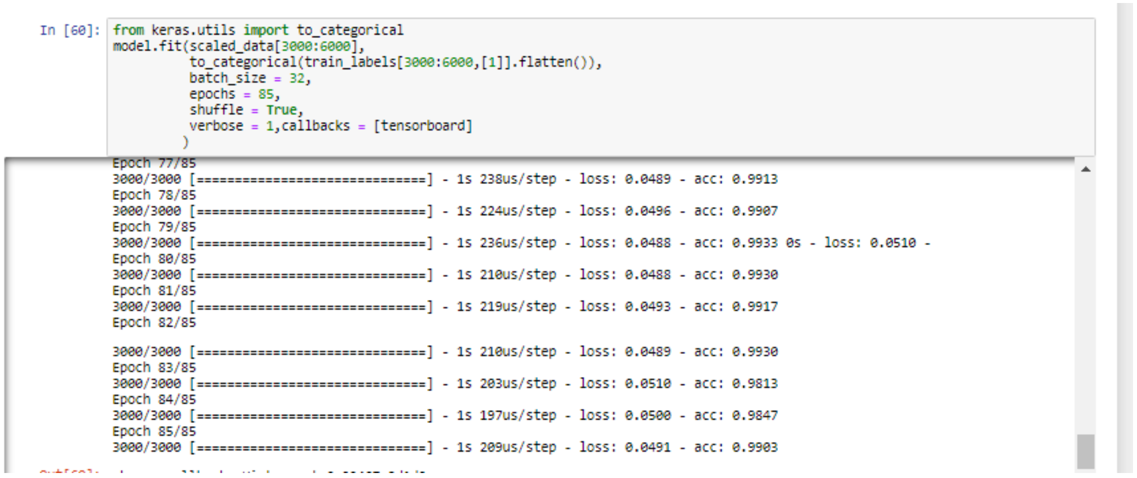
```
In [58]: import sklearn.metrics as metrics
         rounded_predictions = model.predict_classes(scaled_data[6000:9000],verbose = 1)
         print(rounded_predictions)
         print(metrics.accuracy_score(train_labels[6000:9000,[1]],rounded_predictions))
         print(metrics.confusion_matrix(train_labels[6000:9000,[1]],rounded_predictions))

         3000/3000 [==============================] - 0s 75us/step
         [ 9 11 13 ...  3  3 18]
         0.927
```

```
[[117   1   0   0   0   0   6   0   0   3   0   0   0   0   0   0   0   0
    0   0]
 [  0 125   0  10   0   2   0   0   0   0   0   0   0   0   7   0   0   0
    0   0]
 [  0   0 140   0   1   0   0   0   3   0   0   0   3   0   0   0   0   0
    2   0]
 [  0  12   0 127   0   0   0   0   0   0   2   0   0   0   0   3   0   0
    0   0]
 [  0   0   1   0 154   0   0   1   9   0   0   0   2   2   0   0   0   0
    0   0]
 [  1   1   0   2   0 140   0   0   0   0   0   0   0   0   3   1   0   0
    0   0]
 [  3   0   0   2   0   0 139   0   1  15   0   0   0   0   1   0   0   1
    0   3]
 [  0   0   0   0   0   0   0 149   0   0   0   0   0   7   0   0   0   1
    0   0]
 [  0   0   0   0   5   0   0   0 141   0   0   0   0   0   0   0   0   0
    0   0]
 [  0   0   0   2   0   1   0   0   0 145   1   0   0   0   4   0   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0 141   0   0   0   0   1   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 129   0   0   0   0   1   0
    0   0]
 [  0   0   9   0   9   0   0   0   5   0   0   0 132   0   0   0   7   0
    0   0]
 [  0   0   0   0   0   0   0   3   0   0   0   0   0 142   0   0   0   0
    0   0]
 [  0   3   0   8   0   3   0   0   0   2   0   0   0   0 131   2   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 153   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0 151   0
    6   0]
 [  0   0   0   0   2   0   1   0   0   2   0   0   0   0   0   0   0 141
    0   1]
 [  0   0   0   0   0   0   0   0   5   0   0   0   2   0   0   0   6   0
  139   0]
 [  0   1   0   5   0   0   0   0   0   2   5   0   0   0   1   0   0   0
    1 145]]
```

And voila, %92.7, which Is close to the expected,it seems it has the most error on the label 13 clasifying as 2 4

Now overall let’s see the model tested with the data 3000:6000 to predict 0:3000 and 6000:9000

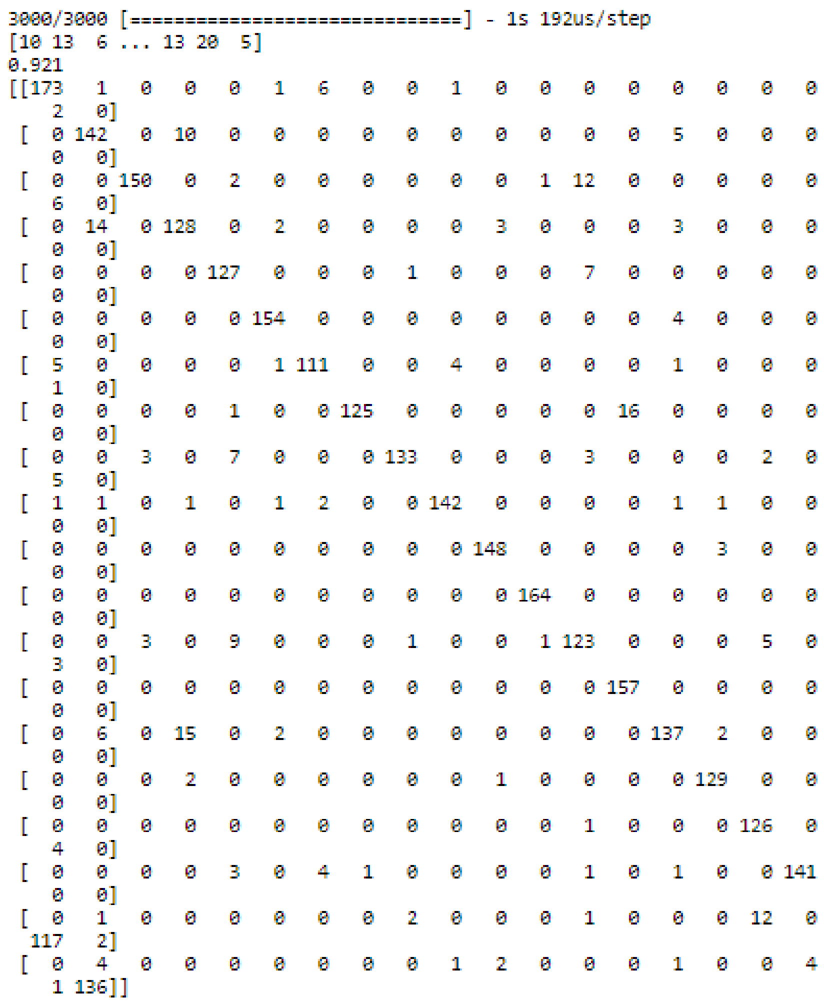
Here it is



```
In [60]: from keras.utils import to_categorical
         model.fit(scaled_data[3000:6000],
                   to_categorical(train_labels[3000:6000,[1]].flatten()),
                   batch_size = 32,
                   epochs = 85,
                   shuffle = True,
                   verbose = 1,callbacks = [tensorboard]
                  )
```

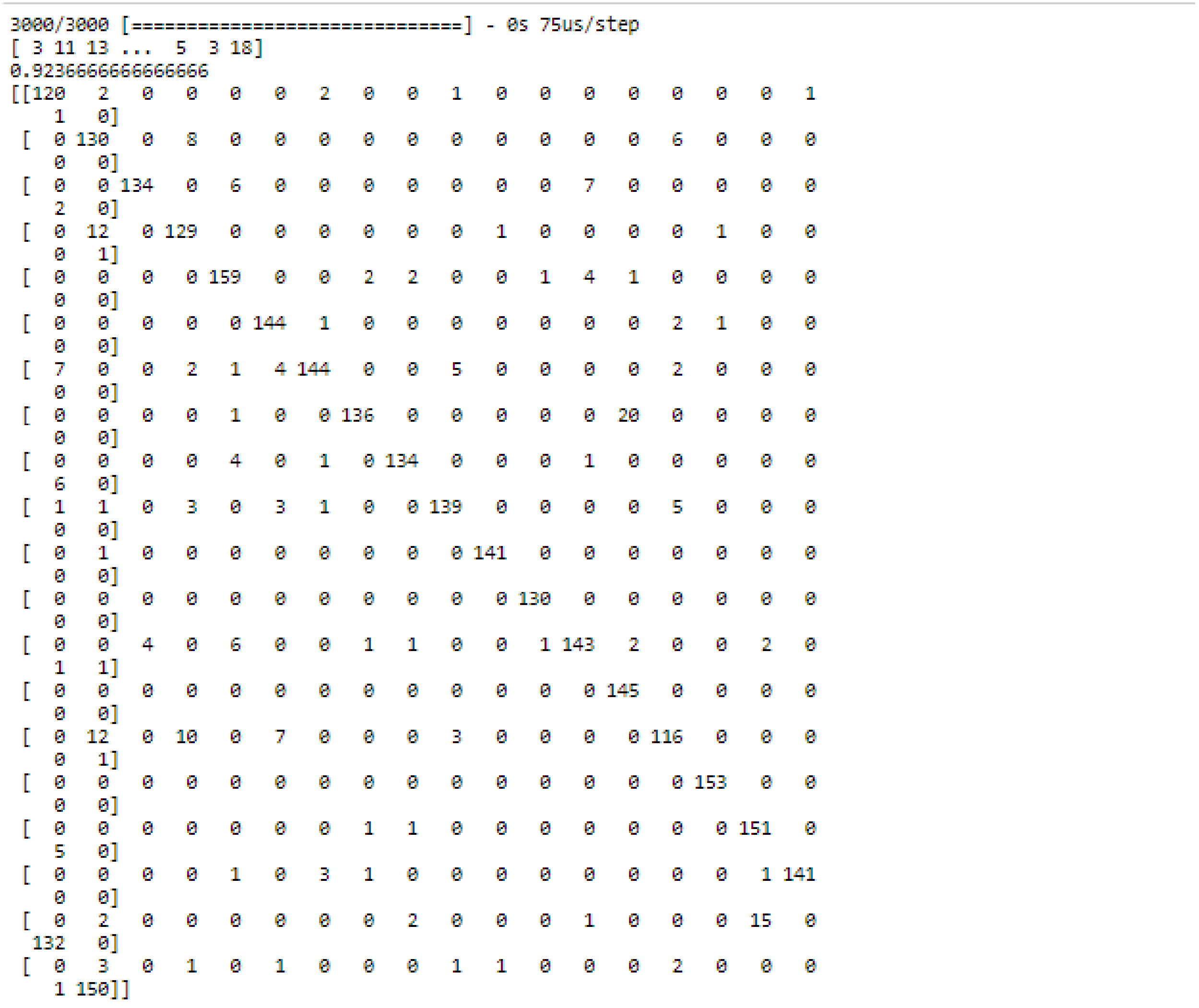
```
Epoch 77/85
3000/3000 [==============================] - 1s 238us/step - loss: 0.0489 - acc: 0.9913
Epoch 78/85
3000/3000 [==============================] - 1s 224us/step - loss: 0.0496 - acc: 0.9907
Epoch 79/85
3000/3000 [==============================] - 1s 236us/step - loss: 0.0488 - acc: 0.9933 0s - loss: 0.0510 -
Epoch 80/85
3000/3000 [==============================] - 1s 210us/step - loss: 0.0488 - acc: 0.9930
Epoch 81/85
3000/3000 [==============================] - 1s 219us/step - loss: 0.0493 - acc: 0.9917
Epoch 82/85

3000/3000 [==============================] - 1s 210us/step - loss: 0.0489 - acc: 0.9930
Epoch 83/85
3000/3000 [==============================] - 1s 203us/step - loss: 0.0510 - acc: 0.9813
Epoch 84/85
3000/3000 [==============================] - 1s 197us/step - loss: 0.0500 - acc: 0.9847
Epoch 85/85
3000/3000 [==============================] - 1s 209us/step - loss: 0.0491 - acc: 0.9903
```

Started with %64 and got to %99.0, now lets put it to the test



```
3000/3000 [==============================] - 1s 192us/step
[10 13  6 ... 13 20  5]
0.921
[[173   1   0   0   0   1   6   0   0   1   0   0   0   0   0   0   0   0
    2   0]
 [  0 142   0  10   0   0   0   0   0   0   0   0   0   0   5   0   0   0
    0   0]
 [  0   0 150   0   2   0   0   0   0   0   0   1  12   0   0   0   0   0
    6   0]
 [  0  14   0 128   0   2   0   0   0   0   3   0   0   0   3   0   0   0
    0   0]
 [  0   0   0   0 127   0   0   0   1   0   0   0   7   0   0   0   0   0
    0   0]
 [  0   0   0   0   0 154   0   0   0   0   0   0   0   0   4   0   0   0
    0   0]
 [  5   0   0   0   0   1 111   0   0   4   0   0   0   0   1   0   0   0
    1   0]
 [  0   0   0   0   1   0   0 125   0   0   0   0   0  16   0   0   0   0
    0   0]
 [  0   0   3   0   7   0   0   0 133   0   0   0   3   0   0   0   2   0
    5   0]
 [  1   1   0   1   0   1   2   0   0 142   0   0   0   0   1   1   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0 148   0   0   0   0   3   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 164   0   0   0   0   0   0
    0   0]
 [  0   0   3   0   9   0   0   0   1   0   0   1 123   0   0   0   5   0
    3   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 157   0   0   0   0
    0   0]
 [  0   6   0  15   0   2   0   0   0   0   0   0   0   0 137   2   0   0
    0   0]
 [  0   0   0   2   0   0   0   0   0   0   1   0   0   0   0 129   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0 126   0
    4   0]
 [  0   0   0   0   3   0   4   1   0   0   0   0   1   0   1   0   0 141
    0   0]
 [  0   1   0   0   0   0   0   0   2   0   0   0   1   0   0   0  12   0
  117   2]
 [  0   4   0   0   0   0   0   0   0   1   2   0   0   0   1   0   0   4
    1 136]]
```

The accuracy due seem to follow up the previous, this is the test on 0:3000

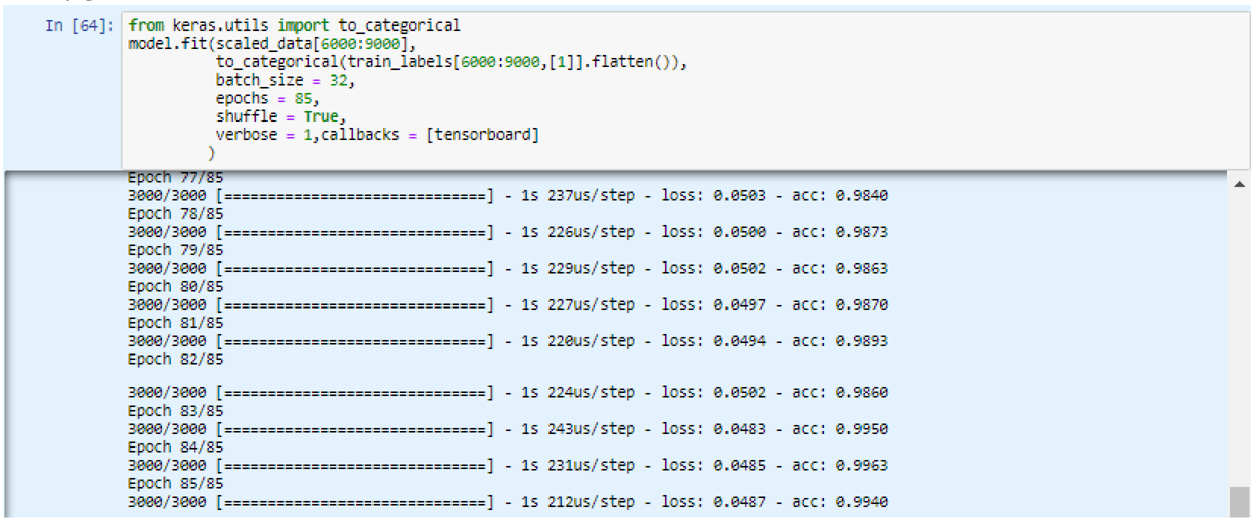


```
3000/3000 [==============================] - 0s 75us/step
[ 3 11 13 ...  5  3 18]
0.9236666666666666
[[120   2   0   0   0   0   2   0   0   1   0   0   0   0   0   0   0   1
    1   0]
 [  0 130   0   8   0   0   0   0   0   0   0   0   0   0   6   0   0   0
    0   0]
 [  0   0 134   0   6   0   0   0   0   0   0   0   7   0   0   0   0   0
    2   0]
 [  0  12   0 129   0   0   0   0   0   0   1   0   0   0   0   1   0   0
    0   1]
 [  0   0   0   0 159   0   0   2   2   0   0   1   4   1   0   0   0   0
    0   0]
 [  0   0   0   0   0 144   1   0   0   0   0   0   0   0   2   1   0   0
    0   0]
 [  7   0   0   2   1   4 144   0   0   5   0   0   0   0   2   0   0   0
    0   0]
 [  0   0   0   0   1   0   0 136   0   0   0   0   0  20   0   0   0   0
    0   0]
 [  0   0   0   0   4   0   1   0 134   0   0   0   1   0   0   0   0   0
    6   0]
 [  1   1   0   3   0   3   1   0   0 139   0   0   0   0   5   0   0   0
    0   0]
 [  0   1   0   0   0   0   0   0   0   0 141   0   0   0   0   0   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 130   0   0   0   0   0   0
    0   0]
 [  0   0   4   0   6   0   0   1   1   0   0   1 143   2   0   0   2   0
    1   1]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 145   0   0   0   0
    0   0]
 [  0  12   0  10   0   7   0   0   0   3   0   0   0   0 116   0   0   0
    0   1]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 153   0   0
    0   0]
 [  0   0   0   0   0   0   0   1   1   0   0   0   0   0   0   0 151   0
    5   0]
 [  0   0   0   0   1   0   3   1   0   0   0   0   0   0   0   0   1 141
    0   0]
 [  0   2   0   0   0   0   0   0   2   0   0   0   1   0   0   0  15   0
  132   0]
 [  0   3   0   1   0   1   0   0   0   1   1   0   0   0   2   0   0   0
    1 150]]
```

It still good on 6000:9000 and some errors on the confusion matrix but more concentrated

Now lets see training on 6000:9000 data and test on 0:6000

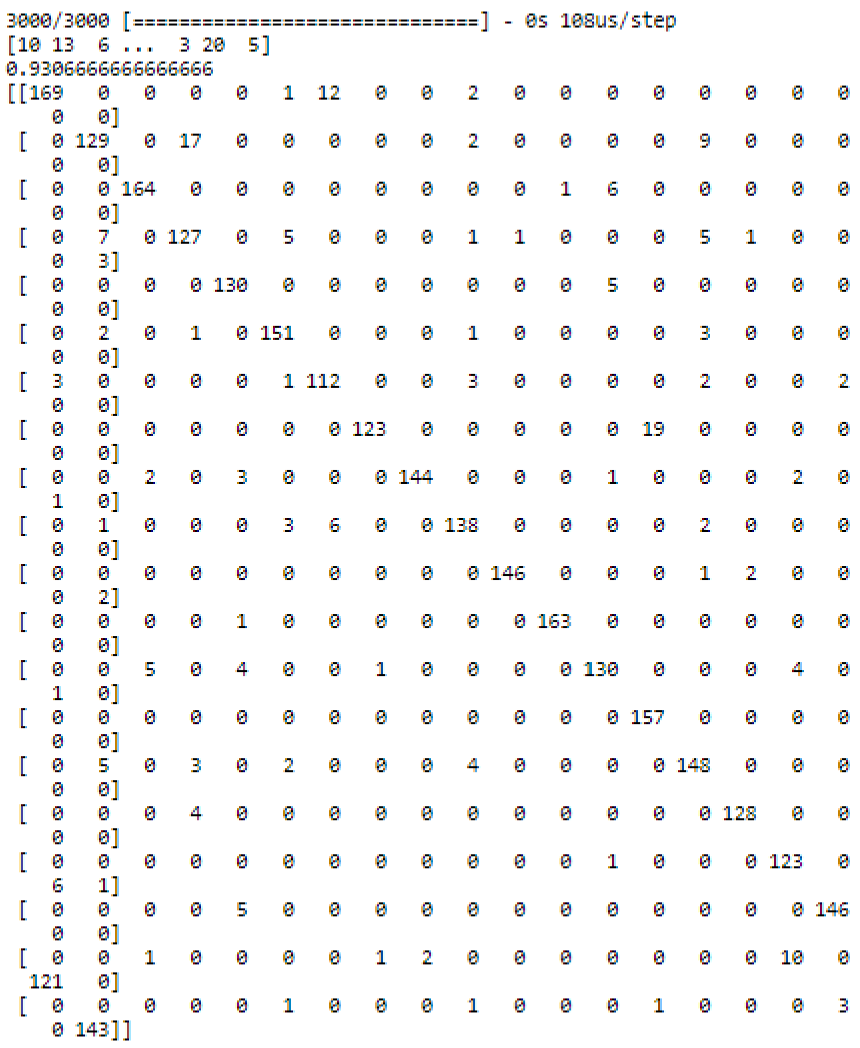
Pretty good, %99.4



```
In [64]: from keras.utils import to_categorical
         model.fit(scaled_data[6000:9000],
                   to_categorical(train_labels[6000:9000,[1]].flatten()),
                   batch_size = 32,
                   epochs = 85,
                   shuffle = True,
                   verbose = 1,callbacks = [tensorboard]
                  )
```

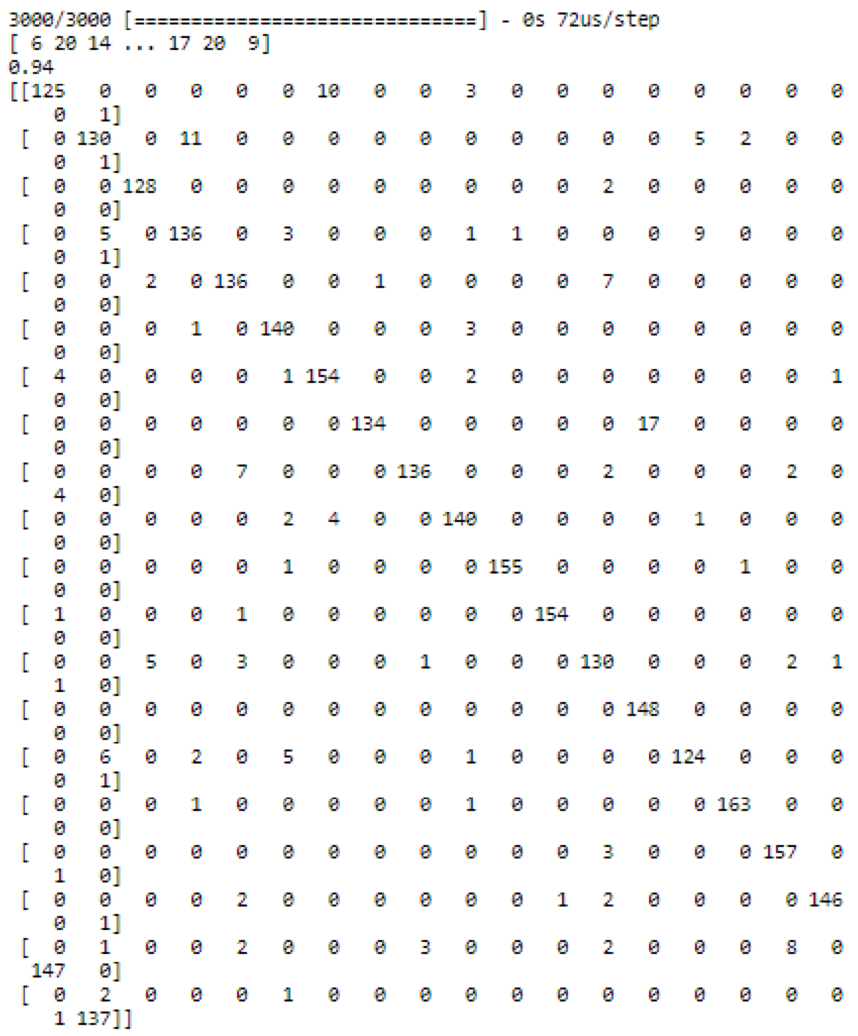
```
Epoch 77/85
3000/3000 [==============================] - 1s 237us/step - loss: 0.0503 - acc: 0.9840
Epoch 78/85
3000/3000 [==============================] - 1s 226us/step - loss: 0.0500 - acc: 0.9873
Epoch 79/85
3000/3000 [==============================] - 1s 229us/step - loss: 0.0502 - acc: 0.9863
Epoch 80/85
3000/3000 [==============================] - 1s 227us/step - loss: 0.0497 - acc: 0.9870
Epoch 81/85
3000/3000 [==============================] - 1s 220us/step - loss: 0.0494 - acc: 0.9893
Epoch 82/85

3000/3000 [==============================] - 1s 224us/step - loss: 0.0502 - acc: 0.9860
Epoch 83/85
3000/3000 [==============================] - 1s 243us/step - loss: 0.0483 - acc: 0.9950
Epoch 84/85
3000/3000 [==============================] - 1s 231us/step - loss: 0.0485 - acc: 0.9963
Epoch 85/85
3000/3000 [==============================] - 1s 212us/step - loss: 0.0487 - acc: 0.9940
```

Lets lets test it



```
3000/3000 [==============================] - 0s 108us/step
[10 13  6 ...  3 20  5]
0.9306666666666666
[[169   0   0   0   0   1  12   0   0   2   0   0   0   0   0   0   0   0
    0   0]
 [  0 129   0  17   0   0   0   0   0   2   0   0   0   0   9   0   0   0
    0   0]
 [  0   0 164   0   0   0   0   0   0   0   0   1   6   0   0   0   0   0
    0   0]
 [  0   7   0 127   0   5   0   0   0   1   1   0   0   0   5   1   0   0
    0   3]
 [  0   0   0   0 130   0   0   0   0   0   0   0   5   0   0   0   0   0
    0   0]
 [  0   2   0   1   0 151   0   0   0   1   0   0   0   0   3   0   0   0
    0   0]
 [  3   0   0   0   0   1 112   0   0   3   0   0   0   0   2   0   0   2
    0   0]
 [  0   0   0   0   0   0   0 123   0   0   0   0   0  19   0   0   0   0
    0   0]
 [  0   0   2   0   3   0   0   0 144   0   0   0   1   0   0   0   2   0
    1   0]
 [  0   1   0   0   0   3   6   0   0 138   0   0   0   0   2   0   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0 146   0   0   0   1   2   0   0
    0   2]
 [  0   0   0   0   1   0   0   0   0   0   0 163   0   0   0   0   0   0
    0   0]
 [  0   0   5   0   4   0   0   1   0   0   0   0 130   0   0   0   4   0
    1   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 157   0   0   0   0
    0   0]
 [  0   5   0   3   0   2   0   0   0   4   0   0   0   0 148   0   0   0
    0   0]
 [  0   0   0   4   0   0   0   0   0   0   0   0   0   0   0 128   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0 123   0
    6   1]
 [  0   0   0   0   5   0   0   0   0   0   0   0   0   0   0   0   0 146
    0   0]
 [  0   0   1   0   0   0   0   1   2   0   0   0   0   0   0   0  10   0
  121   0]
 [  0   0   0   0   0   1   0   0   0   1   0   0   0   1   0   0   0   3
    0 143]]
```

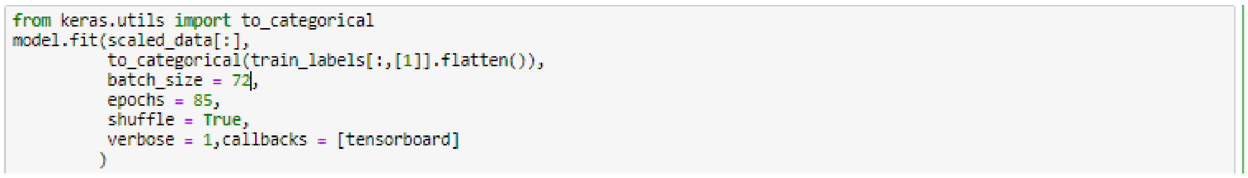
Stable on 0:3000 with %93.06 and concetrated error on the same label.



```
3000/3000 [==============================] - 0s 72us/step
[ 6 20 14 ... 17 20  9]
0.94
[[125   0   0   0   0   0  10   0   0   3   0   0   0   0   0   0   0   0
    0   1]
 [  0 130   0  11   0   0   0   0   0   0   0   0   0   0   5   2   0   0
    0   1]
 [  0   0 128   0   0   0   0   0   0   0   0   0   2   0   0   0   0   0
    0   0]
 [  0   5   0 136   0   3   0   0   0   1   1   0   0   0   9   0   0   0
    0   1]
 [  0   0   2   0 136   0   0   1   0   0   0   0   7   0   0   0   0   0
    0   0]
 [  0   0   0   1   0 140   0   0   0   3   0   0   0   0   0   0   0   0
    0   0]
 [  4   0   0   0   0   1 154   0   0   2   0   0   0   0   0   0   0   1
    0   0]
 [  0   0   0   0   0   0   0 134   0   0   0   0   0  17   0   0   0   0
    0   0]
 [  0   0   0   0   7   0   0   0 136   0   0   0   2   0   0   0   2   0
    4   0]
 [  0   0   0   0   0   2   4   0   0 140   0   0   0   0   1   0   0   0
    0   0]
 [  0   0   0   0   0   1   0   0   0   0 155   0   0   0   0   1   0   0
    0   0]
 [  1   0   0   0   1   0   0   0   0   0   0 154   0   0   0   0   0   0
    0   0]
 [  0   0   5   0   3   0   0   0   1   0   0   0 130   0   0   0   2   1
    1   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 148   0   0   0   0
    0   0]
 [  0   6   0   2   0   5   0   0   0   1   0   0   0   0 124   0   0   0
    0   1]
 [  0   0   0   1   0   0   0   0   0   1   0   0   0   0   0 163   0   0
    0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3   0   0   0 157   0
    1   0]
 [  0   0   0   0   2   0   0   0   0   0   0   1   2   0   0   0   0 146
    0   1]
 [  0   1   0   0   2   0   0   0   3   0   0   0   2   0   0   0   8   0
  147   0]
 [  0   2   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
    1 137]]
```

Wow, %94 and little error with concetrated ones

This seems pretty solid, now for the final stand I have gone with this set



```python
from keras.utils import to_categorical
model.fit(scaled_data[:],
          to_categorical(train_labels[:,[1]].flatten()),
          batch_size = 72,
          epochs = 85,
          shuffle = True,
          verbose = 1,callbacks = [tensorboard]
         )
```

Due to the filesize 9000, used batch size 72 for the balance

It does have a good performance on the data and test, which is good that our model is not overfiting

In conclusion the Neural Network seems to have a good performance and it does not overfit on the test data which it pretty nice and the results we're amazing, I could add more recall,loss graph functions to see more clearly the evolution and to have a proper understanding of the differences between paramters and models.This was my best solution, I also tried low pass filter, gaussian filters and used wavelets with scipy signals for this matter, in place this model got the best approach and a more unique tackle of the problems with a more mathematical and logical approach by dividing and applying mathematical functions to get more unicity for each labels itself making It easier to understandl

Biblography

- https://gist.github.com/junzis/e06eca03747fc194e322 -(Low Pass Filter)
- https://www.youtube.com/watch?v=BidI8_1ikiQ -(Neal Lathia - Mining smartphone sensor data with python) – PyData 2016
- https://pywavelets.readthedocs.io/en/latest/ref/wavelets.html – Wavelets for wave processing