AnomalyDetection_4_MovingAverage

September 29, 2020

1 AnomalyDetection_4_MovingAverage

Updates from previous notebook: - this notebook follows the moving average idea from AnomalyDetection_3_Problem_1

1.1 Libraries and Configuration

[17]:
```
""" Libraries """

#file / system libraries
import os
import datetime as dt

# mathematical

from numpy.fft import ifft
from numpy.fft import fft
import numpy as np

# data exploration

import pandas as pd

# data visualization

import matplotlib.pyplot as plt

""" Configuration """

# pandas

pd.set_option('display.max_columns', None)
```

1.2 Functions

[18]:
```
def polynomial(x):
    """ takes an array and returns it after our polynomial function has been␣
 ↪applied to it"""
    C = [0.7741697399557282,-0.15839741967042406,0.09528795099596377,-0.
 ↪004279871380772796]
    y = C[0]*np.power(x,4)+C[1]*np.power(x,2)+C[2]*x+C[3]
    return y

def directory_to_df(paths, exclude = [None], filetype = '.csv',ignore_index =␣
 ↪True, exception = '_repet'):
    """ concatenates all files in a directory into a dataframe
    components:
    path: path to the directory (must end with /)
    exclude: array of directories to excludes from the treatment
    filetype: a string of the file extension (must include .)
    ignore_index: boolean that tells pandas to ignore the index or not
    exception: takes a string. Any time a filename includes this string it is␣
 ↪treated differently (for cases when you have
    more than one )
    """
    filenames = []
    file_column = []
    frames = []
    test_index = 1

    for path in paths:
        for filename in os.listdir(path):
            print(path)
            if filetype in filename and filename not in exclude:
                if exception in filename:
                    curr_df = pd.read_csv(path+filename)
                    curr_df = special_treatment(curr_df)

                else:
                    curr_df = pd.read_csv(path+filename)
                frames.append(curr_df)
                filenames.append(filename.replace(filetype,''))
                for i in range(curr_df.shape[0]):
                    file_column.append(test_index)
                test_index+=1

    df = pd.concat(frames,ignore_index = ignore_index)
    df['files'] = file_column
    return df, filenames
```

```
def special_treatment(df):
    """ performs a custom operation on a dataframe
    components:
    df: dataframe to play on
    """
    columns = df.columns.values.tolist()
    columns.remove('date')
    df.drop('gyrZ',inplace = True, axis = 1)
    df.columns = columns
    df.reset_index(inplace = True)
    df.rename(columns= {'index':'date'},inplace = True)
    return df

class seasonality():
    """ takes in a dataframe, outputting it with two extra columns: seasonality␣
↪(but column name = seasonality
    inputted) and times, where 'times' is a plottable version of date with␣
↪reference to a prespecified start time
    (day_start)
    Components:
    df: the dataframe, must have the dates column as 'date' and in np.
↪datetime64 timeformat
    seasonality (optional): defaults to 'day'. This is the criteria for␣
↪splitting the data
    day_start (optional): this signifies what is the 'start time' of the day (i.
↪e. the 0 point on the x axis). Defaults
    for midnight.
    time_delta (optional): this defines the units for the time delta between␣
↪data points. Defaults to seconds.
    EDIT THIS MSG
    NEED TO FIX THIS
    """
    def __init__(self,df,seasonality='day',day_start = '00:00:00', time_delta =␣
↪'s'):

        if seasonality not in ['hour','day','month','year']:
            raise ValueError("you can only input the following for seasonality:␣
↪'day', 'month', or 'year'")
        self.df = df
        self.seasonality = 'seasonality_{}'.format(seasonality)
        try:
            self.day_start = dt.datetime.strptime(day_start,'%H:%M:%S')
        except:
            raise ValueError('Please enter your day_start in the correct format:
↪ "HH:MM:SS". "{}" is not acceptable'\
```

```
                                .format(day_start))
        self.time_delta = time_delta

    def find_seasonal_trends(self):
        if 'hour' in self.seasonality:
            self.df[self.seasonality] = self.df.date.dt.hour
        elif 'day' in self.seasonality:
            self.df[self.seasonality] = self.df.date.dt.day
        elif 'month' in self.seasonality:
            self.df[self.seasonality] = self.df.date.dt.month
        else:
            self.df[self.seasonality] = self.df.date.dt.year

        self.create_times()


        return self.df

    def create_times(self):
        times = []
        for season in self.df[self.seasonality].unique():
            temp_dates = self.df.date[self.df[self.seasonality] == season].
↪values
            date = dt.datetime.strptime(str(temp_dates[0])[:-3], '%Y-%m-%dT%H:
↪%M:%S.%f')
            # 'date' is wrong: this will not work for when you have a lower␣
↪order seasonality.
            # it needs to adapt such that it starts recording when the␣
↪beginning of the year
            start_day = dt.datetime(date.year,
                                    date.month,
                                    date.day,
                                    self.day_start.hour,
                                    self.day_start.minute,
                                    self.day_start.second)
            start_day = np.datetime64(start_day)

            for index, date in enumerate(temp_dates):
                times.append((date - start_day)/np.timedelta64(1, self.
↪time_delta))
        self.df['times'] = times
```

1.3 Data

[19]:
```
base = '/Users/yousefnami/KinKeepers/ProjectAI/Kin-Keepers/Data/{}'
names = ['rohan','ignacio']
end_labels = ['_filtered.csv']
dfs = []

for index,name in enumerate(names):
    dfs.append(pd.read_csv(base.format(names[index]+end_labels[0]),index_col =
 ↪0))
```

[20]:
```
dfs[0].head()
```

[20]:

	date	accX	accY	accZ	gyrX	gyrY	gyrZ	files	accTotal	gyrTotal
220	2020-09-14 19:19:26	0.01	0.02	0.00	3.62	1.04	1.38	1	0.022361	4.011284
319	2020-09-14 19:20:39	0.09	0.16	0.14	36.11	25.84	67.85	1	0.230868	81.087978
320	2020-09-14 19:20:40	0.09	0.16	0.09	22.98	15.43	16.45	1	0.204450	32.198879
321	2020-09-14 19:20:41	0.05	0.07	0.09	22.98	15.43	16.45	1	0.124499	32.198879
322	2020-09-14 19:20:42	0.12	0.07	0.07	29.44	39.83	27.27	1	0.155563	56.540210

[21]:
```
dfs[1].head()
```

[21]:

	date	accX	accY	accZ	gyrX	gyrY	gyrZ	files	accTotal	gyrTotal
0	2020-09-13 17:09:25	0.02	0.12	0.03	1.47	3.32	2.22	1	0.125300	4.255784
1	2020-09-13 17:09:26	0.02	0.12	0.03	1.47	3.32	2.22	1	0.125300	4.255784
2	2020-09-13 17:09:27	0.01	0.01	0.00	7.43	6.82	10.10	1	0.014142	14.273307
12	2020-09-13 17:09:34	0.01	0.01	0.00	6.64	7.07	12.45	1	0.014142	15.782173
13	2020-09-13 17:09:34	0.01	0.01	0.00	4.12	3.61	5.81	1	0.014142	7.985149

[22]:
```
import datetime as dt

class moving_avg:
    """
```

```
    a class used to store a moving average values, parameters

    Dependencies:
    -------------


    Attributes:
    -----------

    averages: [*float]
        stores the values of the moving average at each datapoint

    time_frame ( optional - 5 ): int
        the window for the moving average, in hours

    weight ( optional - (0.0, 0.75) ): (float, float)
        weight to apply to numbers greater than the specified quartile

    time_frame_start: datetime
        the start of the moving average window

    Methods:
    --------

    """
    averages = [[0.0,0.0]]

    num_points = []

    points = []
    sum_points = [[0.0, 0.0]]
    time_frame_start = [dt.datetime.strptime('1999-07-24 00:00:00','%Y-%m-%d %H:
↪%M:%S')]
    time_stamps = []

    def __init__( self, time_frame = 5, weight = (0.0, 0.75)):

        self.time_frame = time_frame*3600
        self.weight = weight

    def plot():
        pass # for plotting purposes


class average(moving_avg):
    """
```

```
    Dependencies:
    -------------



    Attributes:
    -----------

    data: [float, float, float, float, float, float]
        represents the list of the seven readings, averaged out over the second:
        [AccX, AccY, AccZ, GyrX, GyrY, GyrZ, Fall]
        Acceleration units are in g, Gyration in degrees per second



    """

    def __init__(self,data,time):
        super().__init__()

        self.data = data
        self.time_stamps.append(dt.datetime.strptime(time,'%Y-%m-%d %H:%M:%S'))

        self.points.append([
            data_point for data_point in self.data
        ]
        )



        if not moving_avg.num_points:
            self.time_frame_start[-1] = self.time_stamps[-1]

        if (self.time_stamps[-1] - self.time_frame_start[-1]).total_seconds()␣
↪>= self.time_frame:
            self.update_attributes()


        self.average()

    def average(self):
        # need to know the index that we are dealing with!
        # need to know index of self.time_frame_start !
        # so, from index of self.time_frame_start[-1], until the end of the␣
↪list, sum all thevalues

        our_index = len(self.time_frame_start) - 1
```

```
        our_range = len(self.points) - our_index
        self.num_points.append(our_range)


        for i in range(len(self.sum_points[-1])):

            for j in range(our_index, len(self.points) - 1):

                self.sum_points[-1][i] += self.points[j][i]

        #print(self.points)
        #print(self.sum_points)

        self.averages[-1] = [sum_point/self.num_points[-1] for sum_point in␣
↪self.sum_points[-1]]

    def update_attributes(self):

        self.time_frame_start.append(
            self.time_stamps[
                self.time_stamps.index(
                    self.time_frame_start[-1]
                ) + 1
            ]
        )
        our_index = self.time_stamps.index(self.time_frame_start[-1]) - 1
        self.num_points.append(len(self.points[our_index:]))
        print(self.num_points)
        self.sum_points.append([0.0,0.0])
        self.averages.append(0)



data = [2,3]
instance = average(data,'1999-07-24 00:00:00')
data = [3,4]
instance = average(data,'1999-07-24 04:00:00')
data = [90,10]
instance = average(data,'1999-07-24 05:00:00')

data = [5,6]
instance = average(data,'1999-07-24 5:01:00')
data = [8,9]

instance = average(data,'1999-07-24 5:02:00')
```

```
moving_avg.averages
```

```
[1, 2, 3]
```

[22]:
```
[[1.0, 1.5], [48.5, 9.5]]
```

[380]:
```
import datetime as dt
import numpy as np
import matplotlib.pyplot as plt

class moving_avg:

    """
    a class used to store a moving average values, parameters and methods

    Dependencies:
    -------------
    import datetime as dt
    import numpy as np
    import matplotlib.pyplot as plt

    Attributes:
    -----------

    data ( class var ): [*[*float]]
        stores all the datapoints for each window

    time_frame_start ( class var ): [datetime]
        the start of the moving average window

    time_stamps ( class var ): [*[*datetime]]
        stores the timestamps for each data point within it's window

    averages ( class var ): [*float]
        stores the values of the moving average for each window

    time_frame ( optional - 5 ): int
        the length of the moving window in units of hours

    weight ( optional - (0.0, 0.75) ): (float, float)
        weight to apply to numbers greater than the specified quartile


    Methods:
    --------
    __init__( self, time_frame = 5, weight = (0.0, 0.75)):
    initialises class based on inputs; converts 'time_frame' to seconds
```

```
    average( self ):
        calculates the averages for each moving window

    plot( self, figsize = (16,8), labels = ('gyrTotal', ' accTotal') ):
        plots the averages against the start time of the moving moving

    """
    data = [[]]
    time_frame_start = []
    time_stamps = [[]]
    averages = []
    # note there is a danger in using class variables because they 'save' every␣
↪instantiations values!

    def __init__( self, time_frame = 5, weight = (0.0, 0.75)):
        self.time_frame = time_frame*3600
        self.weight = weight

    def average( self ):
        for window in self.data:
            window = np.asarray(window).reshape(-2,2)
            self.averages.append([
                window[:,index].mean() for index in range(window.shape[1])
            ])


    def plot( self, figsize = (16,8), labels = ('gyrTotal', ' accTotal') ):
        averages = np.asarray(self.averages).reshape((-2,2))
        fig = plt.figure(figsize = figsize)
        for i in range(averages.shape[1]):
            fig.add_subplot(1,averages.shape[1],i+1)
            #plt.plot(self.time_frame_start,averages[:,i],'.')
            #plt.plot([j for j in range(len(self.time_frame_start))],averages[:
↪,i],'.')

            plt.plot([j for j in range(averages.shape[0])],averages[:,i],'.')
            #plt.xticks(self.time_frame_start)
            plt.xlabel('date')
            plt.ylabel('average {}'.format(labels[i]))

        plt.show()


class average(moving_avg):
    """
```

```
    Dependencies:
    -------------
    moving_avg (class)

    Attributes:
    -----------

    datapoint: [*float]
        datapoint to be considered for averaging, length --> degrees of freedom

    time: str
        time data point is recorded in the format 'YYYY-mm-dd HH:MM:SS'

    Methods:
    --------

    __init__(self, datapoint, time):
        initilises class; converts time to datetime; stores new datapoint and time;
        if new time exceeds average window, creates new storage location

    """

    def __init__(self,datapoint,time):
        super().__init__() # is this necessary?

        self.datapoint = datapoint
        self.time_stamps[-1].append(dt.datetime.strptime(time,'%Y-%m-%d %H:%M:%S'))

        if not self.time_frame_start:
            self.time_frame_start.append(self.time_stamps[-1][-1])
        if (self.time_stamps[-1][-1] - self.time_frame_start[-1]).total_seconds() < self.time_frame:
            pass
        else:
            for i,time in enumerate(self.time_stamps[-1]):
                if time not in self.time_frame_start:
                    self.data.append(
                        self.data[-1][1:]
                    )

                    self.time_frame_start.append(time)

            self.data.append([])
```

```
        self.data[-1].append([
            point for point in datapoint
        ]) # should account for the 'weights' that you've specified here, might␣
 ↪require moving the average method

# what the class is still missing is the 'decision making process', so if an␣
 ↪average is lower than
```

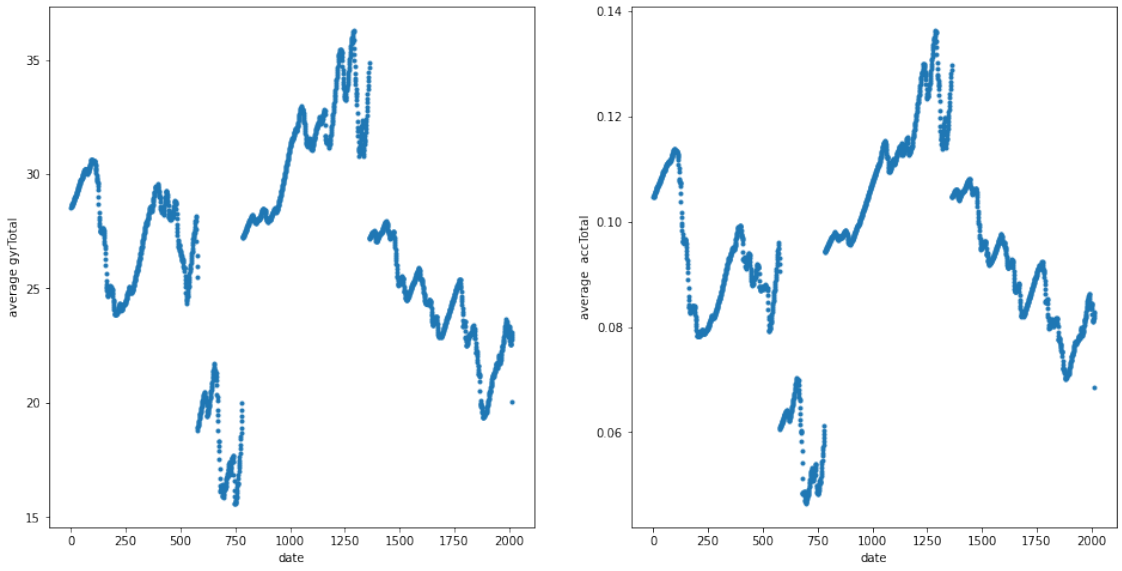
1.4 On read data

[331]:
```
# shows values which are MUCH lower than their previous value

averages = np.asarray(m_avg_instance.averages).reshape(-2,2)
print(averages.shape)
for index in range(1,(averages[:,1]).shape[0]):
    if averages[index,1] < 0.1*averages[index - 1, 1]:
        plt.plot(index,averages[index,1],'+')
```

```
(69, 2)
```

[377]:
```
m_avg_instance = moving_avg()
for item in dfs[1][['gyrTotal','accTotal','date']].values.tolist():
    avg_instance = average(item[0:2],item[2])
m_avg_instance.average()
m_avg_instance.plot()
```

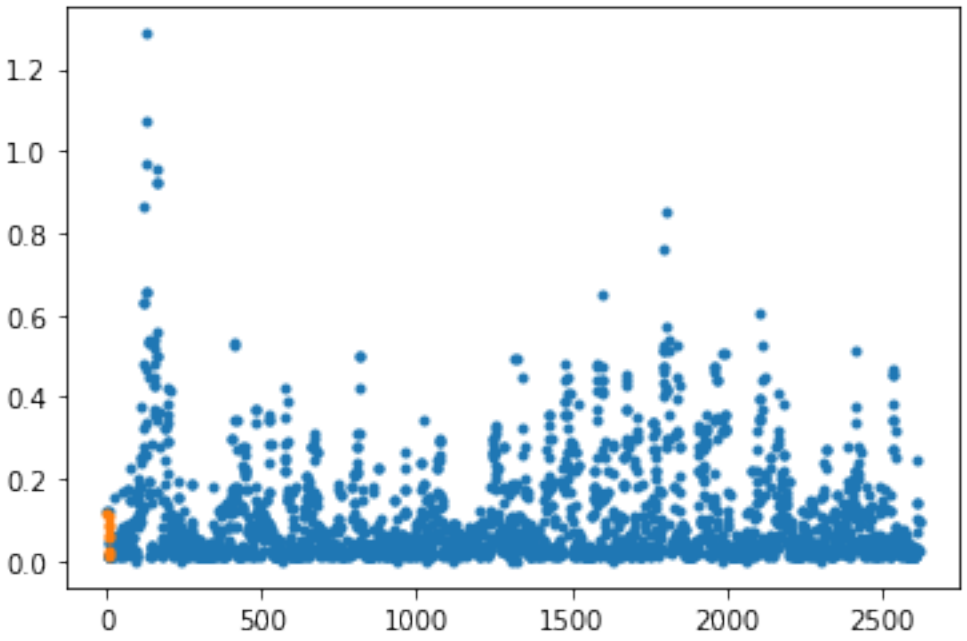


[297]:
```
#dfs[0].reset_index(inplace = True)
#dfs[1].reset_index (inplace = True)
averages = np.asarray(m_avg_instance.averages).reshape(-2,2)

plt.plot(dfs[1].index,dfs[1].accTotal,'.')
plt.plot([i for i in range(len(m_avg_instance.averages))],averages[:,1],'.')


# does not seem to be working... the graphs seems to have just 'shifted place...
 ↪' what is the meaning of this?
# this is very confusing, because it was working well earlier...
```

[297]:
```
[<matplotlib.lines.Line2D at 0x139d5e510>]
```



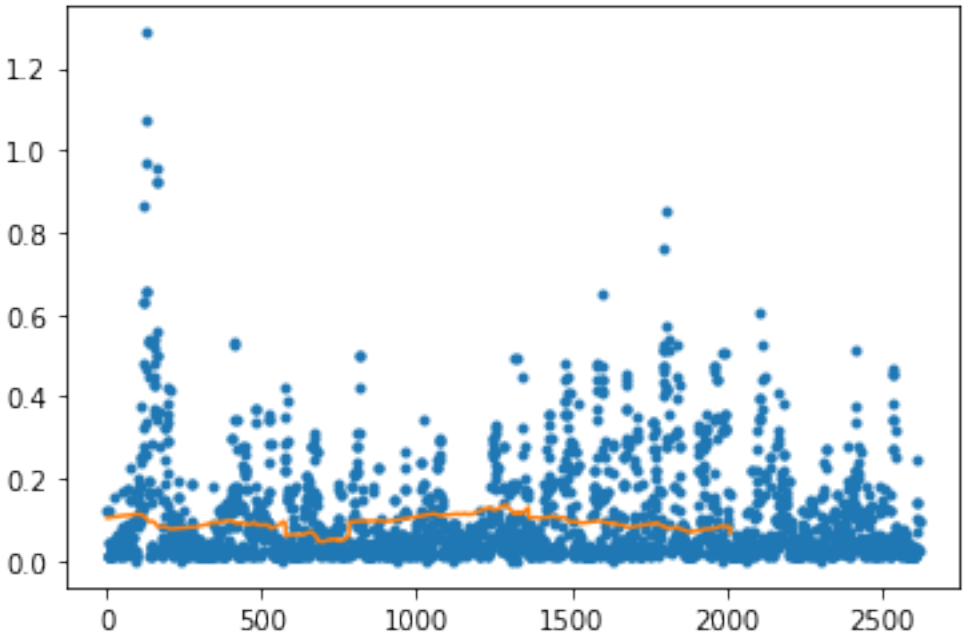
[382]:
```
newinstance = moving_avg()
for item in dfs[1][['gyrTotal','accTotal','date']].values.tolist():
    avg_instance = average(item[0:2],item[2])

newinstance.average()

plt.plot(dfs[1].index,dfs[1][['gyrTotal','accTotal','date']].values[:,1],'.')

plt.plot([i for i in range(len(newinstance.averages))],np.asarray(newinstance.
 ↪averages).reshape(-2,2)[:,1])
```

[382]:
```
[<matplotlib.lines.Line2D at 0x134032190>]
```



2 Conclusion

The average class works, at least in determining the correct average.

There are some changes you need to make in terms of the actual class though, these are summarised below: 1. Currently, you cannot choose to plot the average, with the data points, or average on it's own 2. Currently, the average is calculated at the end, as opposed to at every stage (this was done to save memory, but when the model is deployed, you will need to calculate it every time) 3. You need to think about where everything will be stored, and how this will work in conjunction with Rohan's API (best wait for him to come back from holiday before starting this) 4. You need to account for the weightage when calculating the averages 5. You need to add meaningful xticks, in terms of date and time 6. You need to add functionality to be able to determine when there is a 'break' in the sequence (i.e. much lower values?) 7. You need to fix the way your class handles stuff in memory: it currently saves the values from previous instantiations as well