A Case-based Explanation Method for Weather Forecasting

Import all necessary libraries

```
import pandas as pd
import numpy
import numpy as np
import math
from scipy import signal
import statsmodels.api as sm
from statsmodels.nonparametric.smoothers_lowess import lowess
import matplotlib.patches as mpatches
import matplotlib.lines as mlines
from matplotlib.legend_handler import HandlerLine2D
from numpy import hstack
from numpy import array
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
import matplotlib.pyplot as plt
from sklearn.model_selection import TimeSeriesSplit
```

All about the Artificial Neural Network

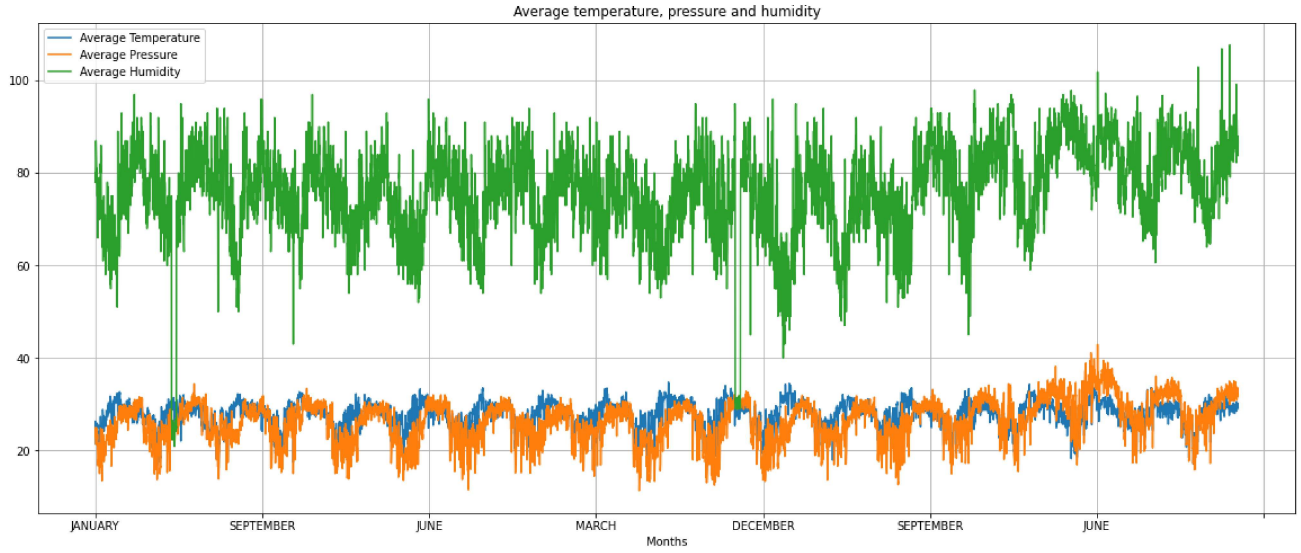
Load dataset

```
df_temp = pd.read_csv("weatherdata.csv", parse_dates= True, index_col= 1)
df_temp.head()
```

MONTH	DATE	TEMP_MAX	TEMP_MIN	TEMP_AVG	PRES_AVG	PRES_MAX	PRES_MIN	HUM_AVG
JANUARY	2000-01-01	29.9	19.2	24.55	23.1	25.3	18.5	80.0
JANUARY	2000-01-02	31.6	21.0	26.30	24.9	27.6	23.2	78.0
JANUARY	2000-01-03	31.2	20.0	25.60	25.0	29.3	23.3	83.0

Global plot

```
muestra = df_temp
plt.figure(figsize=(20,8))
muestra['TEMP_AVG'].plot(legend = True)
muestra['PRES_AVG'].plot(legend = True)
muestra['HUM_AVG'].plot(legend = True)
plt.title('Average temperature, pressure and humidity')
plt.xlabel('Months')
plt.legend(['Average Temperature','Average Pressure','Average Humidity'])
plt.grid()
plt.show()
```



Process dataset

Split dataset into sequences and define train and prediction columns. In this case, predict average temperature through other measures: pressure and humidity

```
step_days = 14
dataset = df_temp.filter(['HUM_MIN','HUM_AVG','HUM_MAX','PRES_MIN','PRES_AVG','PRES_MAX','
dataset = np.array(dataset)

def split_sequences(sequences, n_steps):
```

```
    inputnn, target = list(), list()
····for·i·in·range(len(sequences)):
····    end_ix·=·i·+·n_steps
        if end_ix + 1 > len(sequences):
            break
        seq_x, seq_y = sequences[i:end_ix], sequences[end_ix, (1,4,7)]
        inputnn.append(seq_x)
        target.append(seq_y)
    return array(inputnn), array(target)

inputnn, target = split_sequences(dataset, step_days)
```

Split trainning tests

```
input_train, input_test, target_train, target_test = train_test_split(inputnn, target, tes


model = Sequential()
model.add(LSTM(100, activation = 'relu', return_sequences = True, input_shape = (step_days
model.add(LSTM(32, activation = 'relu', return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(3))
model.compile(optimizer = 'adam', loss = 'mse')
history = model.fit(input_train, target_train, validation_data = (input_test, target_test)
```

```
299/299 [==============================] - 5s 16ms/step - loss: 60.0616 - val_loss
Epoch 27/100
299/299 [==============================] - 5s 16ms/step - loss: 65.0992 - val_loss
Epoch 28/100
299/299 [==============================] - 5s 16ms/step - loss: 61.7708 - val_loss
Epoch 29/100
299/299 [==============================] - 5s 16ms/step - loss: 61.2860 - val_loss
Epoch 30/100
299/299 [==============================] - 5s 16ms/step - loss: 61.2143 - val_loss
Epoch 31/100
299/299 [==============================] - 5s 16ms/step - loss: 61.2655 - val_loss
Epoch 32/100
299/299 [==============================] - 5s 16ms/step - loss: 60.8639 - val_loss
Epoch 33/100
299/299 [==============================] - 5s 16ms/step - loss: 59.6192 - val_loss
Epoch 34/100
299/299 [==============================] - 5s 16ms/step - loss: 58.6818 - val_loss
Epoch 35/100
299/299 [==============================] - 5s 16ms/step - loss: 57.0142 - val_loss
Epoch 36/100
299/299 [==============================] - 6s 20ms/step - loss: 55.1241 - val_loss
Epoch 37/100
299/299 [==============================] - 6s 19ms/step - loss: 56.5936 - val_loss
Epoch 38/100
299/299 [==============================] - 5s 18ms/step - loss: 54.0525 - val_loss
Epoch 39/100
299/299 [==============================] - 5s 18ms/step - loss: 52.5775 - val_loss
Epoch 40/100
299/299 [==============================] - 5s 17ms/step - loss: 51.2739 - val_loss
Epoch 41/100
299/299 [==============================] - 5s 17ms/step - loss: 51.6557 - val_loss
Epoch 42/100
```

```
299/299 [==============================] - 6s 19ms/step - loss: 69.6976 - val_loss
Epoch 43/100
299/299 [==============================] - 5s 17ms/step - loss: 55.7189 - val_loss
Epoch 44/100
299/299 [==============================] - 5s 17ms/step - loss: 53.2360 - val_loss
Epoch 45/100
299/299 [==============================] - 5s 17ms/step - loss: 53.7038 - val_loss
Epoch 46/100
299/299 [==============================] - 5s 17ms/step - loss: 54.6462 - val_loss
Epoch 47/100
299/299 [==============================] - 5s 17ms/step - loss: 54.0778 - val_loss
Epoch 48/100
299/299 [==============================] - 6s 19ms/step - loss: 50.5192 - val_loss
Epoch 49/100
299/299 [==============================] - 5s 17ms/step - loss: 49.4172 - val_loss
Epoch 50/100
299/299 [==============================] - 5s 17ms/step - loss: 49.0884 - val_loss
Epoch 51/100
299/299 [==============================] - 5s 17ms/step - loss: 47.9858 - val_loss
Epoch 52/100
299/299 [==============================] - 5s 17ms/step - loss: 48.5386 - val_loss
Epoch 53/100
299/299 [==============================] - 5s 17ms/step - loss: 49.3637 - val_loss
Epoch 54/100
299/299 [==============================] - 5s 18ms/step - loss: 48.1782 - val_loss
Epoch 55/100
```

Plot global RMSE

```
plt.figure(figsize=(40,8))
for i in range(0,100):
  history.history['loss'][i]= math.sqrt( history.history['loss'][i])
  history.history['val_loss'][i]=math.sqrt( history.history['val_loss'][i])
plt.plot( history.history['loss'], '-o')
plt.plot(history.history['val_loss'], '-o')
plt.title('RMSE')
plt.ylabel('RMSE value')
plt.xlabel('No. epoch')
plt.legend(loc="upper left")
print(history.history['loss'][99])
print(history.history['val_loss'][99])
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start w
6.0587410825896315
3.6604290415786807
```

Plot RMSE of the training split

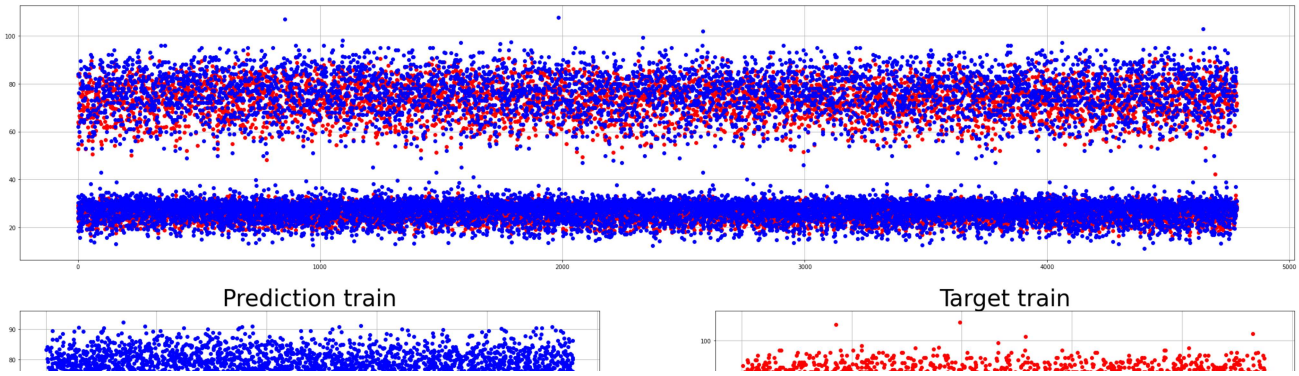
```
prediction_train = model.predict(input_train)

RMSE = math.sqrt(np.square(np.subtract(prediction_train, target_train)).mean())

print("Root Mean Square Error Train:\n",RMSE)
```

```
Root Mean Square Error Train:
 3.720178770889377
```

```
plt.figure(figsize=(40,18))
plt.subplot(2,1,1)
plt.plot(prediction_train, 'ro')
plt.plot(target_train, 'bo')
plt.grid()
plt.subplot(2,2,3)
plt.title("Prediction train", fontsize=40)
plt.plot(prediction_train, 'bo')
plt.grid()
plt.subplot(2,2,4)
plt.title("Target train", fontsize=40)
plt.plot(target_train, 'ro')
plt.grid()
plt.show()
```



Plot RMSE of the test split

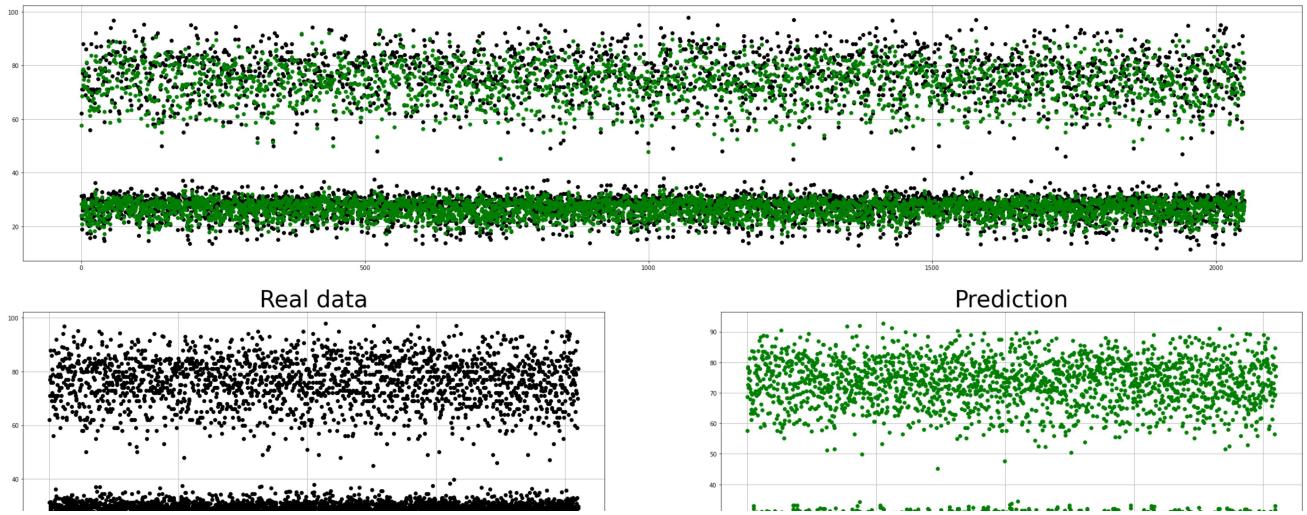
```
prediction = model.predict(input_test)

RMSE = math.sqrt(np.square(np.subtract(prediction, target_test)).mean())

print("Root Mean Square Error Train:\n", RMSE)
```

```
Root Mean Square Error Train:
 3.6604285060522823
```

```
plt.figure(figsize=(40,18))
plt.subplot(2,1,1)
plt.plot(target_test, 'ko')
plt.plot(prediction, 'go')
plt.grid()
plt.subplot(2,2,3)
plt.title("Real data", fontsize=40)
plt.plot(target_test, 'ko')
plt.grid()
plt.subplot(2,2,4)
plt.title("Prediction", fontsize=40)
plt.plot(prediction, 'go')
plt.grid()
plt.show()
```



Analysis Processing

Initialization of variables and constants

The following cell defines all constants and variables with the data to process.

- Windows variable holds the temperature average, humidity average and vapor pressure average data, avoiding the rest of the data.
- The second line emulates the pop method. It stores the last element of the window variable as the targetWindow variable and then that one is deleted by overwritten the windows variable.
- It defines the windowsLen variable by windows variable length.
- It defines the componentesLen variable by the third dimension of windos variable. In this case, there are 3 components, which are the average temperature, average humidity and vapor average pressure.

```
nonOutputColumns=[0,2,3,5,6,8]
windows = np.delete(inputnn, nonOutputColumns, 2)
targetWindow, windows = windows[-1], windows[:-1]
windowsLen = len(windows)
componentsLen = windows.shape[2]
windowLen = windows.shape[1]
actualPrediction = prediction[-1]
titleColumns = ["Humidity", "Vapor Pressure" ,"Temperature"]
titleIndexes = ["Window Index {0}".format(index) for index in range(windowsLen)]
smoothnessFactor = .03
punishedSumFactor = .5
finalWindowNumber = 30
#0 number of results, 1 average, 2 Max values, 3 min values, 4 median
explicationMethodResult = 1
```

Obtaining the Pearson's correlation

The Pearson's correlation is calculated using comprehensive lists. The pearsonCorrelation variable has the same shape as windows variable.

- It uses the corrcoef method from Numpy to acomplish this.
- The results are obtained by performing the operation as if it was the axis = 1.
- It's necessary to reshape in order to get the same shape as it was mentioned before.

```
pearsonCorrelation = np.array(([np.corrcoef(windows[currentWindow,:,currentComponent], tar
for currentWindow in range(len(windows)) for currentComponent in range(componentsLen)])).r
```

Pearson´s correlation print

```
df = pd.DataFrame(data = pearsonCorrelation, columns= titleColumns, index = titleIndexes)
df
```

	Humidity	Vapor Pressure	Temperature
Window Index 0	0.465358	-0.006805	0.077588
Window Index 1	0.257798	-0.041508	0.278657
Window Index 2	0.289051	-0.118465	-0.038017
Window Index 3	0.127883	-0.155410	-0.347418
Window Index 4	0.101848	-0.242479	-0.245252
...	...	...	...
Window Index 6827	-0.147868	-0.381464	0.110078
Window Index 6828	0.207013	-0.296889	0.097895
Window Index 6829	0.085850	-0.126142	-0.043407
Window Index 6830	-0.178847	0.360006	-0.382395
Window Index 6831	-0.195869	0.705340	0.166994

6832 rows × 3 columns

Obtaining the Euclidean distance

The Euclidian distance is also calculated using comprehensive lists.

- It uses the linalg.norm function from Numpy.

As Pearson's correlation, the results are obtained by performing the operation as if it was the axis = 1.

- It's necessary applying the reshape just like Pearson´s correlation is applied.

```
euclideanDistance = np.array(([np.linalg.norm(targetWindow[:,currentComponent] - windows[c
for currentWindow in range(windowsLen) for currentComponent in range(componentsLen)])).res
```

Euclidean distance print

```
df = pd.DataFrame(data = euclideanDistance, columns= titleColumns, index = titleIndexes)
df
```

	Humidity	Vapor Pressure	Temperature
Window Index 0	29.142680	35.048009	18.276761
Window Index 1	32.760435	36.124597	18.285513
Window Index 2	37.263248	38.831628	19.480696
Window Index 3	42.100640	41.205146	21.038180
Window Index 4	45.135287	43.253922	21.982891
...	...	...	...
Window Index 6827	22.946183	6.556789	3.114482
Window Index 6828	18.940411	6.270030	2.873587
Window Index 6829	19.804747	5.899212	2.846050
Window Index 6830	22.692295	4.472908	3.383785
Window Index 6831	22.756445	2.841319	2.631539

6832 rows × 3 columns

Normalized Euclidean distance

Euclidean distance is normalized in order to get values from 0 o 1. The normalization is applied for each component respectively.

```
normalizedEuclideanDistance = euclideanDistance / np.amax(euclideanDistance,axis=0)
```

Normalized Euclidean distance print

```
df = pd.DataFrame(data = normalizedEuclideanDistance, columns= titleColumns, index = title
df
```

	Humidity	Vapor Pressure	Temperature
Window Index 0	0.122758	0.556000	0.489366
Window Index 1	0.137997	0.573079	0.489600
Window Index 2	0.156965	0.616023	0.521601
Window Index 3	0.177341	0.653676	0.563304
Window Index 4	0.190124	0.686178	0.588598
...	...	...	...
Window Index 6827	0.096657	0.104017	0.083391
Window Index 6828	0.079783	0.099467	0.076941
Window Index 6829	0.083424	0.093585	0.076204
Window Index 6830	0.095587	0.070958	0.090602
Window Index 6831	0.095857	0.045075	0.070460

Obtaining the normalization between the Pearson's correlation and Euclidean distance

To do so, the following equation was used:

```
normalizedCorrelation = (.5+(pearsonCorrelation-2*normalizedEuclideanDistance+1)/4)
```

Normalized Correlation print

```
df = pd.DataFrame(data = normalizedCorrelation, columns= titleColumns, index = titleIndexe
df
```

	Humidity	Vapor Pressure	Temperature
Window Index 0	0.804960	0.470299	0.524714
Window Index 1	0.745451	0.453084	0.574864

Normalized Correlation sum and punished sum

As is possible in Python, first it's applied the punished sum to normalizedCorrelation variable and then the Numpy's sum function is used by setting axis = 1 as an argument. Finally, the result is stored in the correlationPerWindow variable.

In the second line, the correlationPerWindow variable is normalized in order to get values from 0 to 1.

```
correlationPerWindow = np.sum(((normalizedCorrelation+punishedSumFactor)**2), axis=1)
correlationPerWindow /= max(correlationPerWindow)
```

6832 rows × 3 columns

Normalized Correlation per Window sum print

```
df = pd.DataFrame(data = correlationPerWindow, columns= ["Correlation per window"], index
df
```

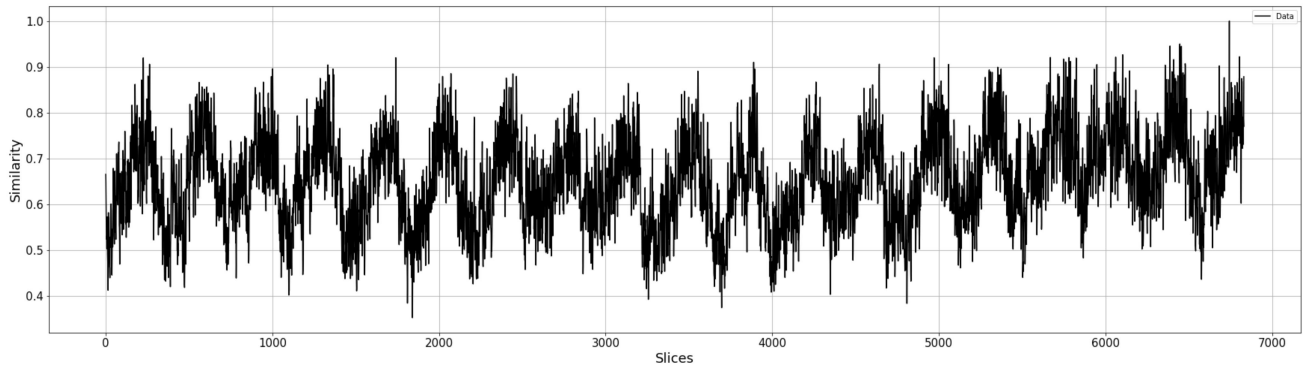
	Correlation per window
Window Index 0	0.665207
Window Index 1	0.650876
Window Index 2	0.601246
Window Index 3	0.537107
Window Index 4	0.523870
...	...
Window Index 6827	0.738155
Window Index 6828	0.790082
Window Index 6829	0.779214
Window Index 6830	0.769142
Window Index 6831	0.878518

6832 rows × 1 columns

Correlation per window plot

The following figure shows the correlation value of each window, being the correlation the similarity compared to targetWindow variable.

```
plt.figure(figsize=(30,8))
plt.plot(correlationPerWindow,'k', label='Data')
plt.ylabel('Similarity',fontsize=18)
plt.xlabel('Slices',fontsize=18)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



Highest values filtering

As a first step, the time series tendency is discovered.

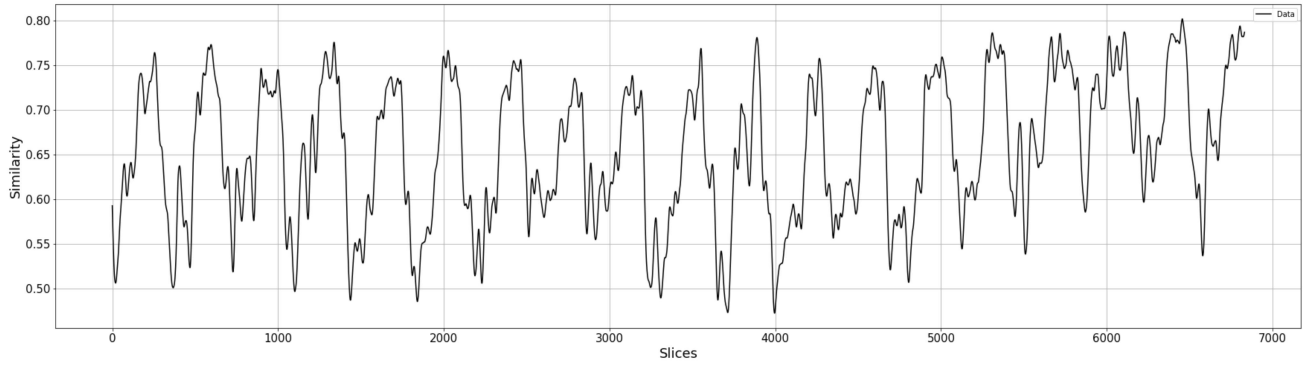
- To acomplish this the Hodrick-Prescott filter implemented as a function in the Statsmodel module is used.
- In the following cell the tendency component and cyclic component are obtained.

```
cyclicComponent, tendencyComponent = sm.tsa.filters.hpfilter(correlationPerWindow)
```

Tendency time series plot

```
plt.figure(figsize=(30,8))
```

```
plt.plot(tendencyComponent,'k', label='Data')
plt.ylabel('Similarity',fontsize=18)
plt.xlabel('Slices',fontsize=18)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



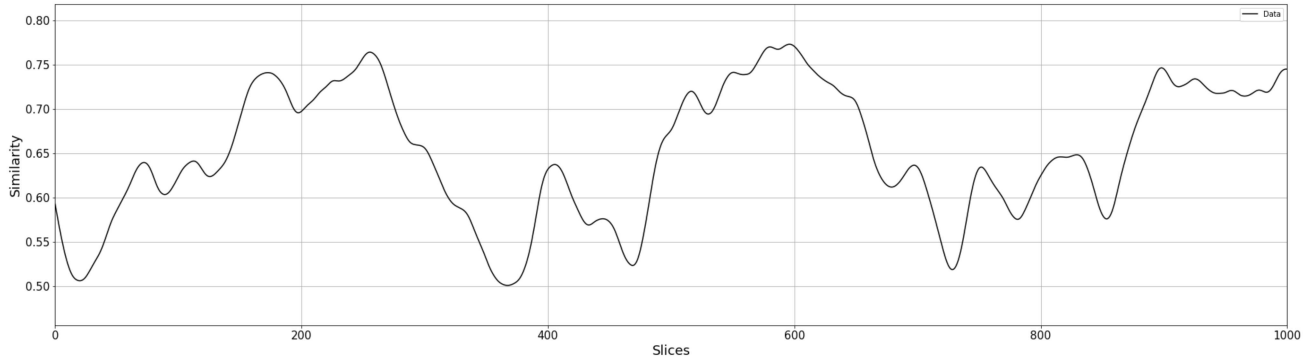
Noise Reduction

It can be observed the kind of pattern of the time series.

Now it's a little more obvious where and which the peaks are, but there are still some peaks so near by others that they could be considered as noise.

The following figure shows the problem mentioned before.

```
plt.figure(figsize=(30,8))
plt.plot(tendencyComponent,'k', label='Data')
plt.ylabel('Similarity',fontsize=18)
plt.xlabel('Slices',fontsize=18)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.xlim(0,1000)
plt.grid()
plt.legend()
plt.show()
```



Noise removing

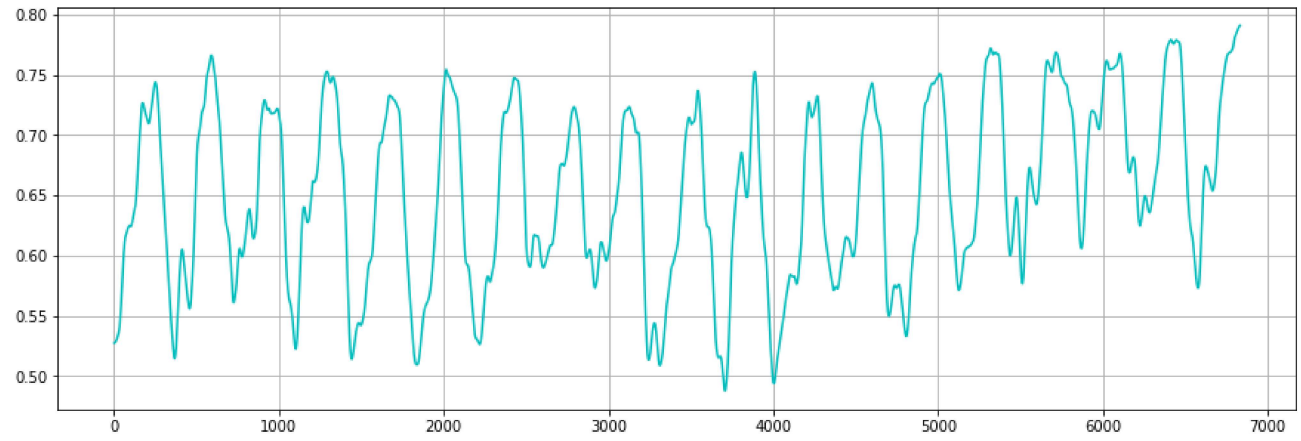
To remove the noise, the Local Regression implemented as a function in the same statsmodels module is used,this is performed in order to get a better results without many peaks near others.

- In the first time the argument in the frac parameter of lowess function is 1%.

```
df_loess_1 = pd.DataFrame(lowess(correlationPerWindow, np.arange(len(correlationPerWindow)
```

In the figure below it can be observed tha the result is so similar to the tendency component, what it means that is workable way to continue smoothing but there is still some noise.

```
plt.figure(figsize=(15,5))
plt.plot(df_loess_1, 'c', label='Data')
plt.grid()
plt.show()
```



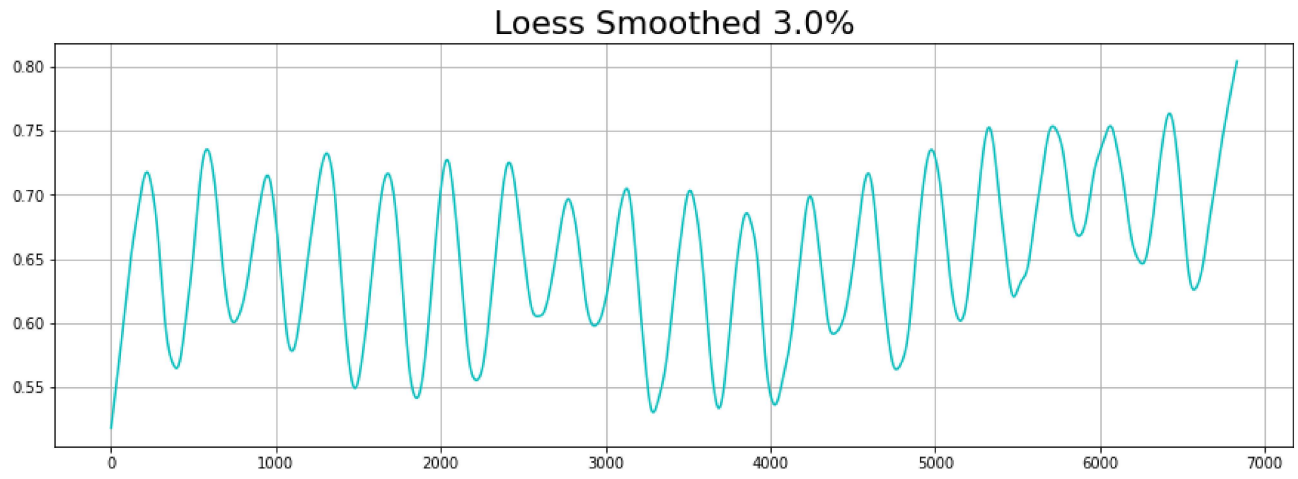
To get better results, the frac parameter is setted to 3%.

- The smoothedCorrelation variable is overwritten to store only the data needed which is the smoothed time series data.

```
smoothedCorrelation = lowess(correlationPerWindow, np.arange(len(correlationPerWindow)), s
df_loess_3 = pd.DataFrame(smoothedCorrelation)
```

Now the result presented in the following figure has no peaks near by others such as the figures above.

```
plt.figure(figsize=(15,5))
plt.plot(df_loess_3, 'c', label='Data')
plt.title("Loess Smoothed {0}%".format(smoothnessFactor*100), fontsize=22)
plt.grid()
plt.show()
```



For purposes of this problem, all noise peaks have been removed.

Time series peaks finding

The peaks and valleys are useful to determine the best and worst comparison cases to windowTarget variable.

- To know the peaks and valleys the argrelextrema function from scipy is used.
- The the results of the following cell is the peak indexes and valleys indexes.
- The function returns a lots information but the only want needed are the index of peaks and valleys.

```
valleyIndex, peakIndex = signal.argrelextrema(smoothedCorrelation, np.less)[0], signal.arg
```

```
The peaks and valleys indexes are shown in th
```

The peaks and valleys indexes are shown in the figure below.

```
print(peakIndex)
print(valleyIndex)
```

```
[ 217  581  948 1307 1678 2038 2414 2773 3128 3511 3855 4240 4593 4977
 5326 5713 6062 6421]
[ 395  744 1097 1479 1853 2215 2584 2931 3289 3687 4027 4381 4764 5152
 5477 5872 6256 6570]
```

Due to regression local nature, smoothedCorrelation variable do not match with the original peaks and valleys of normalizedCorrelation. To know the real peaks or correlationPerWindow, it's necessary do the following:

- Use peak indexes to split and for each segment detect the highest correlation value.
- Use valley indexes to spplit and for each segment detect the lowes correlation value.
- peakSegment variable contains all the peak segments.
- valleySegment variable contains all the valley segments.

Split into segments based on worst and best indices

It creates to arrays which contains the segments.

- The concaveSegments contains the segments with the highest values of correlationPerWindow.
- The convexSegments contains the segments with the lowest values of correlationPerWindow.

```
concaveSegments = np.split(np.transpose(np.array((np.arange(windowsLen), correlationPerWin
convexSegments = np.split(np.transpose(np.array((np.arange(windowsLen), correlationPerWind
```

Peak segments and valley segments plot

Detection of the highest and lowest value of each segment

It creates two list which store the indexes.

- The bestWindowIndex variable stores the indices with the highest correlation values.
- The worstWindowIndex variable stores the indices with the lowest correlation values.

```
bestWindowsIndex, worstWindowsIndex = list(), list()

for split in concaveSegments:
    bestWindowsIndex.append(int(split[np.where(split == max(split[:,1]))[0][0],0]))
for split in convexSegments:
    worstWindowsIndex.append(int(split[np.where(split == min(split[:,1]))[0][0],0]))


bestDic = {index: correlationPerWindow[index] for index in bestWindowsIndex}
worstDic = {index: correlationPerWindow[index] for index in worstWindowsIndex}


bestSorted = sorted(bestDic.items(),reverse=True, key=lambda x:x[1])
worstSorted = sorted(worstDic.items(), key=lambda x:x[1])
```

- *maxComp* and *minComp* are the maximum and minimum values respectively for each component.
- *lims* is just for the plot and figures.

```
maxComp,minComp,lims=[],[],[]
for i in range(componentsLen):
    maxComp.append(int(max(max(a) for a in windows[:,:,i])))
    minComp.append(int(min(min(a) for a in windows[:,:,i])))
    lims.append(range(minComp[i],maxComp[i],int((maxComp[i]-minComp[i])/8)))
```

- *bestMAE* and *worstMAE* are lists that contains the closeness evaluation by applying the MAE formula to best and worst windows.

```
bestMAE,worstMAE = [],[]
for i in range(len(bestSorted)):
    rawBestMAE=rawWorstMAE=0
    for f in range(componentsLen):
        rawBestMAE+=(windows[bestSorted[i][0]][windowLen-1][f]-minComp[f])/maxComp[f]
        rawWorstMAE+=(windows[worstSorted[i][0]][windowLen-1][f]-minComp[f])/maxComp[f]
    bestMAE.append(rawBestMAE/componentsLen)
    worstMAE.append(rawWorstMAE/componentsLen)
```

- This is a table showing the best windows on the right and worst windows on the left with the CCI and MAE respectively.

```
d= {'index': dict(bestSorted).keys(), 'CCI': dict(bestSorted).values(), "MAE":bestMAE,'ind
df = pd.DataFrame(data = d)
```

```
df
```

	index	CCI	MAE	index.1	CCI.1	MAE.1
0	6744	1.000000	0.473211	1841	0.352121	0.304575
1	6447	0.949990	0.506829	3697	0.374093	0.267785
2	6104	0.926356	0.513824	4809	0.383365	0.241105
3	5670	0.920444	0.491887	3257	0.392146	0.318216
4	1741	0.919793	0.483111	1099	0.401556	0.297968
5	4972	0.919789	0.465373	4349	0.403002	0.388419
6	225	0.919493	0.475584	3995	0.407915	0.326055
7	3889	0.909912	0.476766	1506	0.410679	0.228984
8	4643	0.905761	0.437775	14	0.412140	0.329896
9	1333	0.904259	0.465644	473	0.418147	0.393364
10	5354	0.899087	0.495920	2205	0.425984	0.344752
11	1002	0.895869	0.452547	6577	0.435851	0.306392
12	3554	0.890581	0.458063	783	0.438815	0.369519
13	2072	0.885151	0.473449	5504	0.440066	0.290233
14	2444	0.884774	0.450999	2865	0.447963	0.343282
15	4263	0.866803	0.478725	2518	0.457332	0.421092
16	561	0.865987	0.492642	5130	0.461043	0.435484
17	3136	0.863786	0.463030	5868	0.482672	0.296242
18	2836	0.847121	0.447713	6220	0.502002	0.380086

- It defines the notCombinedOption function which prints all the windows that user defines quantity as variable.

```
def notCombinedOption():
    for i in range(len(bestSorted)):
        plt.figure(figsize=(12,8))
        for f in range(componentsLen):
            plt.subplot(componentsLen,1,f+1)
            plt.title(titleColumns[f])
            plt.plot(cont,targetWindow[:,f], '.-k', label = "Target")
            plt.plot(cont,windows[bestSorted[i][0],:,f], '.-g' ,label= "Data")
            plt.plot(windowLen+1,actualPrediction[f], 'dk', label = "Prediction")
            plt.plot(windowLen+1,windows[bestSorted[i][0]][windowLen-1][f], 'dg', label =
            plt.grid()
            plt.xticks(range(1,windowLen+2,1))
            plt.yticks(lims[f])
```

```
        plt.tight_layout()
        plt.show()
```

It defines a function which it has the task of selecting if the general case are going to be handled by maximum, minimum or average.

```
def subOptions(op):
    if op==1:
        newCase = np.sum(windows[list(dict(bestSorted).keys())], axis=0)/len(bestSorted)
    elif op==2:
        newCase = np.max(windows[list(dict(bestSorted).keys())], axis=0)
    elif op==3:
        newCase = np.min(windows[list(dict(bestSorted).keys())], axis=0)
    elif op==4:
        newCase = np.median(windows[list(dict(bestSorted).keys())], axis=0)
    return newCase
```

This function invoke the subOptions function to define the way of the general case and the rest of the code is only for printing the windows.

```
def combinedOption():
    plt.figure(figsize=(12,8))
    newCase = np.zeros((windowLen,componentsLen))
    try:
        newCase = subOptions(explicationMethodResult)
    except:
        print("Unavailable option")
    for f in range(componentsLen):
        plt.subplot(componentsLen,1,f+1)
        plt.title(titleColumns[f])
        plt.plot(cont,targetWindow[:,f], '.-k', label = "Target")
        plt.plot(cont,newCase[:,f], '.-g' ,label= "Data")
        plt.plot(windowLen+1,actualPrediction[f], 'dk', label = "Prediction")
        plt.plot(windowLen+1,newCase[windowLen-1][f], 'dg', label = "Next day")
        plt.grid()
        plt.xticks(range(1,windowLen+2,1))
        plt.yticks(lims[f])
    plt.tight_layout()
    plt.show()
```

This code handles if the results are going to be printed depending on the selected option by the user.

```
cont=np.arange(1,windowLen+1)
if explicationMethodResult == 0:
    notCombinedOption()
else:
    combinedOption()
```