# Python for Scientific Computations and Control

FINAL ASSIGNMENT

Andres Gonzalez Padilla



# Contents

1	Introduction	3
2	Analysed Data	
	2.1 Respiratory Time Series	3
	2.2 Electro-cardiogram Signal	4
3	Implementing a Predictive Model	4
	3.1 Artificial Neural Network and Learning Algorithm	4
	3.2 Parameters for the neural network	6
	3.3 Prediction future Values of the Respiration Time Series Signal	6
	3.4 Mean Absolute Error for the Predicting model	9
4	Discord Discovery	9
	4.1 Brute Force algorithm	9
	4.2 Parameters for the Brute Force Discovery	12
5	Working with PyQtGraph library in Python environment	13
	5.1 Comparison in the methods applied	13
	5.2 Using PvQt as Matplot	14
	5.3 Plotting examples	15
6	References	16

## 1 Introduction

Signal processing has become a core feature for the development of new tools in the biomedical ambit. From cardiology to oncology, the analysis of time series can be used for forecasting or finding anomalies. The objective of this work is two investigate two different methods implemented in Python that should be ready to work with any input signal. The first implementation is predictive and consists of a neural network that uses Levenberg-Marquardt with adaptive momentum for training. Levenberg-Marquardt algorithm, also known as damped least-squares, uses an interpolation between Gauss-Newton algorithm and the method of gradient descent for optimization. The second implementation is used to detect anomalies in the given data and is grounded in the Adaptive Window Based Discord Discovery (AWDD) which itself is inspired by Brutal Force Discord Discovery (BFDD) and Heuristic Discord Discovery (HDD). In addition, the new library PyQtgraph will be investigated and compared to the usual Matplotlib.

## 2 Analysed Data

#### 2.1 Respiratory Time Series

The data analysed was selected in accordance to the objective. For the prediction method, a respiratory time series (RTS) representing the position of a lung tumor is analysed and was acquired at Hokkaido University Hospital at 30 Hz sampling. The signals have a length of 18.6 seconds and are shown in figure 1.



Figure 1: Respiratory Time Series for lung tumor.

#### 2.2 Electro-cardiogram Signal

The data for the Discord Discovery and Brute Forces was taken from the Physionet Databank and represents the case of Premature Ventricular Contraction which is suitable for anomaly detection. The length of the data is quite long (1805 seconds) so the code permits the user to select the time range as well as the number of windows for the analysis. The signals for seconds 3 through 10 is shown in figure 2.



Figure 2: Respiratory Time Series for lung tumor.

## 3 Implementing a Predictive Model

#### 3.1 Artificial Neural Network and Learning Algorithm

Artificial neural networks can be used to predict future values of a given signal by analysing to recent history of the signal and by using a learning algorithm.

These models mimic the "real life" behaviour of neurons and the electrical messages they produce between input, processing by the brain and the final output from the brain.

There are several types of Neural Networks, the type of architecture used in the present modelling is called Feedforward neural network. This architecture is the most simple type since information flows in one direction only. It can consist of several layers (perceptrons) or only one. For this method, only one input vector is used for one layer of output nodes.

Although the N.N. used is feedforward, in the method implemented the process is repeated a number of times (epochs) by putting it inside a cycle. The new output will be the input for the next iteration to progressively decrease the error obtained.

An input vector  $x_{(k)}$  consisting of data form the signal's history is used in a weighting function with the coefficients  $w_i$ . The output vector  $y_{(k)}$  of the function is then compared to the real values and the error is calculated. Using the error between real data and model output the weights are updated with the Levenberg-Marquardt algorithm in order to reduce future error.

$$x_{(k)} \Rightarrow \boxed{f(x,w)} \longrightarrow y_{(k)}$$

Specifically for the method implemented, the function of the layer is simply a dot multiplication between the input and weights vectors.

The Levenberg-Marquardt algorithm for a linear model works as follows. The Jacobian matrix is made of all input vectors with the length m.

$$\mathbf{J} = \begin{bmatrix} 1 & x_1(1) & x_1(1) & \dots & x_m(1) \\ 1 & x_2(2) & x_2(2) & \dots & x_m(2) \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & x_1(N) & x_2(N) & \dots & x_m(N) \end{bmatrix}$$
(3-1)

Training the N.N. is by comparing the results to the real data. The weight increment is calculated with the error e between real data and model output and the so called learning rate  $\mu$  in the following way:

$$dw = \left( \left( \left( \mathbf{J}^{\mathsf{T}} \cdot \mathbf{J} \right) + \frac{1}{\mu} \mathbb{1} \right)^{-1} \cdot \mathbf{J}^{\mathsf{T}} \right) \cdot e \tag{3-2}$$

The updated weight after a learning period is then:

$$w = w + dw . ag{3-3}$$

With this procedure the model is able to learn after each learning period and can fit the model better after it.

#### 3.2 Parameters for the neural network

The user defined parameters look like this in the code:

epochs = 30
mu = 0.01
L = 15
pr = 5
The epochs is

The epochs is the number of times the weight vector will be updated. It requires a certain minimum amount of updates until the system is unable to further improve the results.

Mu is the learning rate used in Levenberg-Marquardt. Should be less than the unit.

L is the length of input and output vector; it should be considerably smaller compared to the total length of the data.

The prediction range (pr) is the number of samples ahead the model will predict. It should be big enough to get relevant results but small enough for them to be reliable. The larger the range, the bigger the risk of getting large error.

The weights vector is initialized randomly.

#### 3.3 Prediction future Values of the Respiration Time Series Signal

A simple artificial single-layer neural network is implemented for prediction of future values of the RTS signal. The length of the input vector and so the number of nodes is 15 and the learning rate is set to  $\mu = 0.01$ . The prediction range which represent the number of samples predicted ahead is selected with a value in concordance to the input vector. As the prediction range is further increased the forecast result can get worse; in figure 3 an acceptable result is seen for a prediction range of 5.



Figure 3: Simple predictive model and real data.

To improve results a second model was implemented. This advance model receives the same signal but the input is divided in two. It gets the the recent history in the first half of input and the other half of the input is the history of the signal one period ago which is obtained from the lag value of the autocorrelation function where the correlation coefficient attains its maximum. Taking into consideration the size of data, the period for autocorrelation selected is 30 samples or 1 s. The model output for a prediction range of 5 is shown in figure 4.



Figure 4: Advanced predictive model and real data.



Figure 5: All models and real data with prediction range of 5 samples.

The comparison of performance of both models can be seen in figure 5. From the graph it can be deducted that the advance model gets a better prediction than the simple one. This is confirmed by figure 6 which compares the sum of errors of both implementations. Although both models converge to the same error almost in the same period, for periods before, the difference is considerable.



Figure 6: Sum of squared errors of all models at a prediction range of 5 samples.

#### 3.4 Mean Absolute Error for the Predicting model

The mean absolute error is a statistical measure of how far estimates or forecasts are from actual values. It is most often used in time series and the procedure is two subtract the predicted value from the actual value in each row. Then take the absolute value of each row and add up the absolute values. Finally divide by the total length of data. This means:

$$MAE = \frac{1}{|T|} \sum_{(u,i\in T)} |\hat{r}_{ui} - r_{ui}|$$
(3-4)

Where  $\hat{r}_{ui}$  is the predicted value and  $r_{ui}$  is the true value.

The results for the implemented N.N.:

Simple Model: 0.383658186761

Advance Model: 0.406947280621

The results are almost the same, the advantage of the advance model, however, comes in that it requires less epochs and therefore the N.N. itself is more efficient.

#### 4 Discord Discovery

#### 4.1 Brute Force algorithm

Time series discords are subsequences of longer time series that are maximally different to all the rest of the time series subsequences. They thus capture the sense of the most unusual subsequence within a time series. While discords have many uses for data mining, they are particularly attractive as anomaly detectors because they only require one intuitive parameter: the subsequence length n. In order to analyze the time series, all possible subsequences can be extracted by sliding a window of size n across the whole data.

$$\sum_{n=1}^{10} \frac{1}{n^2}$$

The method uses the Euclidian distance which is defined as a function that has two subsequences C and M as inputs and returns a nonnegative value R, which is said to be the distance from M to C. For subsequent definitions to work we require that the function D be symmetric, that is, Dist(C, M) = Dist(M, C). We also assume that the two subsequences are of equal length n.

The Brute Force Discord Discovery method takes each possible subsequence and find the distance to the nearest non-self match (non-overlapping subsequence). The subsequence that has the greatest such distance value is the discord. The position of this discord is saved and then displayed on top of the original graph. This is achieved with nested loops, where the outer loop considers each possible candidate subsequence, and the inner loop is a linear scan to identify the candidate?s nearest non-self match (check appendix).

For the ECG signal of choice, the data between 3 and 10 seconds was selected since there is an anomaly appearance in the middle. The code receives from the user the number of windows N which is automatically translated into the subsequence length n by dividing the length of data over N. For analysis of the method a number of N = 10 is used since there are 9 peaks in the time period selected. The code is run for the values n through 8n (on-growing windows) to compare the results when different amount of peaks as used as sliding window. The first four plots, produced with the library Matplotlib, can be seen in figure 7. The next four plots, produced with the library PyQtGraph, can be seen in figure 8.



Figure 7: Anomaly detection plotting with matplotlib for the first 4 window lengths.



Figure 8: Anomaly detection plotting with PyQtGraph for window lengths 5 to 8.

It can be seen that the anomaly is detected in every case; the subsequence which corresponds to the greater distance always is in the neighbourhood of the anomaly; however for the first case where the window (subsequence length) is smaller than the anomaly, the method fails. This means that the user should select a window length of at least 3 peaks since the anomaly for Premature Ventricular Contraction occurs in a length between two and three peaks.

#### 4.2 Parameters for the Brute Force Discovery

For the Brute Force method, only the window size is needed. The user can also specify the fragment of the data to be analysed by defining the beginning and end times which turns out to be useful is there is an specific region where an anomaly is suspected. The user defined parameters look like this in the code:

 $N_of_windows = 10$ 

 $t_userin = 3$ 

 $t\_userfin = 10$ 

## 5 Working with PyQtGraph library in Python environment

#### 5.1 Comparison in the methods applied

The PyQtgraph is an alternative to Matplot library for Python. It has been addressed that PyQt is faster and more versatile specially because it includes the feature of plotting live data somewhat easily.

In the previous methods, the plot was done with both libraries and PyQt has both advantages and disadvantages. The annotated code used for plotting the Brute Force method is as follows:

Plotting in MatPlotlib.

#	figure()
#	<pre>plot(t, y, 'b', tbest, ybest, 'r')</pre>
#	<pre>legend(('real data', 'anomaly'))</pre>
#	<pre>grid(), title('Signal 1 - ECG')</pre>
#	show()

Plotting in PyQtGraph (multiple plots).

Create window with title:

```
win = pg.GraphicsWindow(title="Plotting PV-ECG") # plot t vs y in red
#win.setWindowTitle('Plotting PV-ECG') #alternative for title input
```

Enable antialiasing for prettier plots:

```
pg.setConfigOptions(antialias=True)
```

Plot in red and add title:

p1 = win.addPlot(title="Signal 1 PV-ECG and Anomaly")

p1.plot(t, y, pen='r')

Add second plot on top in white color:

```
p1.plot(tbest,ybest, pen=(255,255,255,200))
```

Show grid?

# p1.showGrid(x=True, y=True)

Create Region Selection and add it on plot:

lr = pg.LinearRegionItem([t[Loc],t[Loc + num -1]])
lr.setZValue(-10)
p1.addItem(lr)

The final product of PyQt looks more professional and it is true that the individual plotting of the graphs was quicker which turns out useful for these type of graphs when several features are shown.

Also, it is possible to modify the graph (zoom in, move, change color or export) easier than Matplot library.

However, when generating multiple graphs inside a loop (for example for different window sizes), it would usually skip one or more plots randomly. Running the code several times was necessary, or decreasing the number of plots which made the process slower.

#### 5.2 Using PyQt as Matplot

The following code can be used for simple plotting using Qt for a user already familiar with Matplot library:

User defined function:

```
def qplot(t,y,color, labelx, labely, qtitle):
    ## Plotting in PyQtGraph (multiple plots)
    win = pg.GraphicsWindow(title=qtitle) # plot t vs y in red
    #win.setWindowTitle('Plotting PV-ECG')
    # Enable antialiasing for prettier plots
    pg.setConfigOptions(antialias=True)
    # plot
    p1 = win.addPlot(title=qtitle)
    p1.plot(t, y, pen=color)
    p1.setLabel('left', labely)
    p1.setLabel('bottom', labelx)
Plot command for main:
```

## Plotting with defined function with pyqtgraph
qplot(t,y,'r', 'X axis', 'Y axis', 'Insert Title')

#### 5.3 Plotting examples

These are the basic plots explored. See Appendix for coding. A nice feature is the sliding of region highlighted which can be linked to another graph showing only the region selected. This turns out useful for live showing of results (this can be seen in the last two plots).



Figure 9: Basic plots achieved with PyQtGraph library.

# **6** References

[1] Keogh, E., Lin, j., & Lee, S. (2006). Finding the most unusual time series subsequence: algorithms and applications. Knowledge and Information Systems, 18.

# Appendix

#### Python Code for Brute Force

```
# Python assignment - Andres Gonzalez Padilla
# analyzing position of a lung tumor data, creating a predictive model
# data was acquired at Hokkaido University Hospital at 30 Hz sampling.
## imported in spyder
#from numpy import *
#from matplotlib import *
#from matplotlib.pyplot import *
#from scipy import fft
from numpy.linalg import inv
from numpy.random import randn
from pylab import *
# Respiration data, array
yrts = loadtxt('RTS_30Hz.txt')
tfinal = len(yrts)/30
t = linspace(0, tfinal, len(yrts))
## Neural Network Predictive Model
#Initializing
                 # real data for the model
yr = yrts
mu = 0.01
                # learning rate
L = 15
                 # length of input and output vector
N = len(yr)
                # length of total data
                 # initializing input
x = ones(L)
w = randn(L) / L + initializing weighting with random values
epochs = 30
                  # number of learning epochs
dwdy = zeros((N, L))
                       # initializing weighting derivative
I = eye(L)
                  # identity matrix
pr = 5
                  # prediction range (number of samples ahead)
## levenberg-marquardt method
y = zeros(N)
                # initializing model output
e = y.copy()
                # initializing error
SSE = zeros(epochs)
                      # initializing sum of squared errors
```

```
#y[0:L + 1] = yr[0:L + 1] # initial condition for dynamic model
for epoch in range(epochs):
    for j in range(L - 1 - pr, N - pr):
        x[1:] = yr[range(j, j - L + 1, -1)] #input
        y[j + pr] = dot(x, w) #output
        e[j + pr] = yr[j + pr] - y[j + pr]
        dwdy[j + pr] = x
    J = dwdy
    dw = dot(dot(inv(dot(J.T, J) + 1 / mu * I), J.T), e)
    w = w + dw
    SSE[epoch] = sum(e * e)
print(SSE[epoch])
print('Simple Model Finished')
##MAE simple model
MAEsimple = sum(abs(yr-y))/size(y)
## Advance Model with referring to values one period ago
yadv = zeros(N) # initializing combinated model output
eadv = zeros(N) # initializing error for combimodel
SSEadv = zeros(epochs) # initializing sum of squared errors for combimodel
odd = 0
if (L != 2 * (L / 2)):
    odd = 2 # correction of input at odd length of input vector
for epoch in range(epochs):
    for j in range(L - 1 - pr , N - pr):
        x[1:] = yr[range(j, j - L + 1, -1)]
        if (j > 30 + 3 * L / 4 - 1): # c
            x[L / 2:] = yr[range(j - (30-L/4), j - (30+L/4) - odd, -1)]
        yadv[j + pr] = dot(x, w)
        eadv[j + pr] = yr[j + pr] - yadv[j + pr]
        dwdy[j + pr] = x
    J = dwdy
    dw = dot(dot(inv(dot(J.T, J) + 1 / mu * I), J.T), eadv)
    w = w + dw
    SSEadv[epoch] = sum(eadv * eadv)
print(SSEadv[epoch])
print('Advance Model Finished')
```

```
##MAE advance model
MAEadvance = sum(abs(yr-yadv))/size(y)
print(MAEsimple)
print(MAEadvance)
### plotting
## plotting original data
figure(1)
plot(t, yr, 'b')
grid(), title('Signal 1 - RTS')
figure(2)
plot(t, yr, 'k', t, y, 'b')
grid(), title('Simple predictive Model')
legend(('real data', 'simple model'))
xlabel('time [s]')
figure(3)
plot(t, yr, 'k', t, yadv, 'c')
grid(), title('Advanced predictive Model')
legend(('real data', 'advanced model'))
xlabel('time [s]')
figure(4)
plot(t, yr, 'k', t, y, 'b', t, yadv, 'c')
grid(), title('Predictive models')
legend(('real data', 'simple', 'advanced'))
xlabel('time [s]')
figure(5)
plot(SSE)
plot(SSEadv, 'c')
grid(), title('Sum of squared errors')
legend(('simple', 'advanced'))
xlabel('learning period')
```

show()

# Python Code for Brute Force Python Code for PyQtGraph library

# -\*- coding: utf-8 -\*-..... Created on Sun Apr 27 22:40:13 2014 **@author:** Andres ..... # Basic Plot # -\*- coding: utf-8 -\*-..... This example demonstrates many of the 2D plotting capabilities in pyqtgraph. All of the plots may be panned/scaled by dragging with the left/right mouse buttons. Right click on any plot to show a context menu. ..... #import initExample ## Add path to library (just for examples; you do not need this) from pyqtgraph.Qt import QtGui, QtCore import numpy as np import pyqtgraph as pg #QtGui.QApplication.setGraphicsSystem('raster') #app = QtGui.QApplication([]) #mw = QtGui.QMainWindow() #mw.resize(800,800) win = pg.GraphicsWindow(title="Basic plotting examples") win.resize(1000,600) win.setWindowTitle('pyqtgraph example: Plotting') # Enable antialiasing for prettier plots pg.setConfigOptions(antialias=True)

```
# GRAPH NUMBER 1
p1 = win.addPlot(title="Basic array plotting", y=np.random.normal(size=100))
# GRAPH NUMBER 2
p2 = win.addPlot(title="Multiple curves")
p2.plot(np.random.normal(size=100), pen=(255,0,0))
p2.plot(np.random.normal(size=100)+5, pen=(0,255,0))
p2.plot(np.random.normal(size=100)+10, pen=(0,0,255))
# GRAPH NUMBER 3
p3 = win.addPlot(title="Drawing with points")
p3.plot(np.random.normal(size=100), pen=(200,200,200), symbolBrush=(255,0,0), symbolPe
# GRAPH NUMBER 4
win.nextRow()
p4 = win.addPlot(title="Parametric, grid enabled")
x = np.cos(np.linspace(0, 2*np.pi, 1000))
y = np.sin(np.linspace(0, 4*np.pi, 1000))
p4.plot(x, y)
p4.showGrid(x=True, y=True)
# GRAPH NUMBER 5
p5 = win.addPlot(title="Scatter plot, axis labels, log scale")
x = np.random.normal(size=1000) * 1e-5
y = x*1000 + 0.005 * np.random.normal(size=1000)
y -= y.min()-1.0
mask = x > 1e-15
x = x[mask]
y = y[mask]
p5.plot(x, y, pen=None, symbol='t', symbolPen=None, symbolSize=10, symbolBrush=(100,
p5.setLabel('left', "Y Axis", units='A')
p5.setLabel('bottom', "Y Axis", units='s')
p5.setLogMode(x=True, y=False)
# GRAPH NUMBER 6 ("LIVE" PLOT)
p6 = win.addPlot(title="Updating plot")
curve = p6.plot(pen='y')
data = np.random.normal(size=(10,1000))
```

```
ptr = 0
def update():
    global curve, data, ptr, p6
    curve.setData(data[ptr%10])
    if ptr == 0:
        p6.enableAutoRange('xy', False) ## stop auto-scaling after the first data se
    ptr += 1
timer = QtCore.QTimer()
timer.timeout.connect(update)
timer.start(50)
# GRAPH NUMBER 7
win.nextRow()
p7 = win.addPlot(title="Filled plot, axis disabled")
y = np.sin(np.linspace(0, 10, 1000)) + np.random.normal(size=1000, scale=0.1)
p7.plot(y, fillLevel=-0.3, brush=(50,50,200,100))
p7.showAxis('bottom', False)
# GRAPH NUMBER 8
x2 = np.linspace(-100, 100, 1000)
data2 = np.sin(x2) / x2
p8 = win.addPlot(title="Region Selection")
p8.plot(data2, pen=(255,255,255,200))
#p8.plot(x2, data2, pen=(255,255,255,200))
lr = pg.LinearRegionItem([400,700])
lr.setZValue(-10)
p8.addItem(lr)
# GRAPH NUMBER 9
p9 = win.addPlot(title="Zoom on selected region")
p9.plot(data2)
def updatePlot():
    p9.setXRange(*lr.getRegion(), padding=0)
def updateRegion():
    lr.setRegion(p9.getViewBox().viewRange()[0])
lr.sigRegionChanged.connect(updatePlot)
p9.sigXRangeChanged.connect(updateRegion)
```

```
updatePlot()
## Start Qt event loop unless running in interactive mode or using pyside.
if __name__ == '__main__':
    import sys
    if (sys.flags.interactive != 1) or not hasattr(QtCore, 'PYQT_VERSION'):
        QtGui.QApplication.instance().exec_()
```