PyEEG Reference Guide

Release 0.02 r1

Forrest Sheng Bao

August 30, 2010

CONTENTS

Release 0.02 r1

Date August 30, 2010

PyEEG, a Python module to extract EEG features, v 0.02_r1

Project homepage: http://pyeeg.org

Data structure

PyEEG only uses standard Python and numpy data structures, so you need to import numpy before using it. For numpy, please visit http://numpy.scipy.org

Naming convention

I follow "Style Guide for Python Code" to code my program http://www.python.org/dev/peps/pep-0008/

Constants: UPPER_CASE_WITH_UNDERSCORES, e.g., SAMPLING_RATE, LENGTH_SIGNAL.

Function names: lower_case_with_underscores, e.g., spectrum_entropy.

Variables (global and local): CapitalizedWords or CapWords, e.g., Power.

If a variable name consists of one letter, I may use lower case, e.g., x, y.

CHAPTER

ONE

FUNCTIONS LISTED ALPHABETICALLY

ap_entropy (*X, M, R*)

Computer approximate entropy (ApEN) of series X, specified by M and R.

Suppose given time series is X = [x(1), x(2), ... , x(N)]. We first build embedding matrix Em, of dimension (N-M+1)-by-M, such that the i-th row of Em is x(i),x(i+1), ... , x(i+M-1). Hence, the embedding lag and dimension are 1 and M-1 respectively. Such a matrix can be built by calling pyeeg function as Em = embed_seq(X, 1, M). Then we build matrix Emp, whose only difference with Em is that the length of each embedding sequence is M + 1

Denote the i-th and j-th row of Em as Em[i] and Em[j]. Their k-th elments are Em[i][k] and Em[j][k] respectively. The distance between Em[i] and Em[j] is defined as 1) the maximum difference of their corresponding scalar components, thus, max(Em[i]-Em[j]), or 2) Euclidean distance. We say two 1-D vectors Em[i] and Em[j] *match* in *tolerance* R, if the distance between them is no greater than R, thus, max(Em[i]-Em[j]) <= R. Mostly, the value of R is defined as 20% - 30% of standard deviation of X.

Pick Em[i] as a template, for all j such that 0 < j < N - M + 1, we can check whether Em[j] matches with Em[i]. Denote the number of Em[j], which is in the range of Em[i], as k[i], which is the i-th element of the vector k. The probability that a random row in Em matches Em[i] is simga_1^{N-M+1} k[i] / (N - M + 1), thus sum(k)/ (N - M + 1), denoted as Cm[i].

We repeat the same process on Emp and obtained Cmp[i], but here 0<i<N-M since the length of each sequence in Emp is M + 1.

The probability that any two embedding sequences in Em match is then sum(Cm)/ (N - M +1). We define Phi_m = sum(log(Cm)) / (N - M + 1) and Phi_mp = sum(log(Cmp)) / (N - M).

And the ApEn is defined as Phi_m - Phi_mp.

See Also:

samp_entropy sample entropy of a time series

Notes

Extremely slow implementation. Do NOT use if your dataset is not small.

References

Costa M, Goldberger AL, Peng CK, Multiscale entropy analysis of biolgical signals, Physical Review E, 71:021906, 2005

bin_power(*X, Band, Fs*)

Compute power in each frequency bin specified by Band from FFT result of X. By default, X is a real signal.

Parameters **Band** :

list

boundary frequencies (in Hz) of bins. They can be unequal bins, e.g. [0.5,4,7,12,30] which are delta, theta, alpha and beta respectively. You can also use range() function of Python to generate equal bins and pass the generated list to this function.

Each element of Band is a physical frequency and shall not exceed the Nyquist frequency, i.e., half of sampling frequency.

X :

list

a 1-D real time series.

Fs :

integer

the sampling rate in physical frequency

Returns **Power** :

list

spectral power in each frequency bin.

Power_ratio :

list

spectral power in each frequency bin normalized by total power in ALL frequency bins.

dfa(*X, Ave=None, L=None*)

Compute Detrended Fluctuation Analysis from a time series X and length of boxes L.

The first step to compute DFA is to integrate the signal. Let original seres be X= [x(1), x(2), ..., x(N)].

The integrated signal Y = [y(1), y(2), ..., y(N)] is otained as follows y(k) = sum_{i=1}^{k}{x(i)-Ave} where Ave is the mean of X.

The second step is to partition/slice/segment the integrated sequence Y into boxes. At least two boxes are needed for computing DFA. Box sizes are specified by the L argument of this function. By default, it is from 1/5 of signal length to one (x-5)-th of the signal length, where x is the nearest power of 2 from the length of the signal, i.e., 1/16, 1/32, 1/64, 1/128, ...

In each box, a linear least square fitting is employed on data in the box. Denote the series on fitted line as Yn. Its k-th elements, yn(k), corresponds to y(k).

For fitting in each box, there is a residue, the sum of squares of all offsets, difference between actual points and points on fitted line.

F(n) denotes the square root of average total residue in all boxes when box length is n, thus Total_Residue = sum_{k=1}^{N}{(y(k)-yn(k))} F(n) = sqrt(Total_Residue/N)

The computing to F(n) is carried out for every box length n. Therefore, a relationship between n and F(n) can be obtained. In general, F(n) increases when n increases.

Finally, the relationship between F(n) and n is analyzed. A least square fitting is performed between log(F(n)) and log(n). The slope of the fitting line is the DFA value, denoted as Alpha. To white noise, Alpha should be 0.5. Higher level of signal complexity is related to higher Alpha.

Parameters **X:** :

1-D Python list or numpy array a time series

Ave: :

integer, optional The average value of the time series

L: :

1-D Python list of integers A list of box size, integers in ascending order

Returns **Alpha:** :

integer the result of DFA analysis, thus the slope of fitting line of log(F(n)) vs. log(n). where n is the

Notes

This value depends on the box sizes very much. When the input is a white noise, this value should be 0.5. But, some choices on box sizes can lead to the value lower or higher than 0.5, e.g. 0.38 or 0.58.

Based on many test, I set the box sizes from 1/5 of signal length to one (x-5)-th of the signal length, where x is the nearest power of 2 from the length of the signal, i.e., 1/16, 1/32, 1/64, 1/128, ...

You may generate a list of box sizes and pass in such a list as a parameter.

Examples

```
>>> import pyeeg
>>> from numpy.random import randn
>>> print pyeeg.dfa(randn(4096))
0.490035110345
```

embed_seq(*X, Tau, D*)

Build a set of embedding sequences from given time series X with lag Tau and embedding dimension DE. Let X = [x(1), x(2), ... , x(N)], then for each i such that 1 < i < N - (D - 1) * Tau, we build an embedding sequence, Y(i) = [x(i), x(i + Tau), ... , x(i + (D - 1) * Tau)]. All embedding sequence are placed in a matrix Y.

Parameters **X** :

list

a time series

Tau :

integer

the lag or delay when building embedding sequence

D :

integer

the embedding dimension

Returns **Y** :

2-D list

embedding matrix built

Examples

```
>>> import pyeeg
>>> a=range(0,9)
>>> pyeeg.embed_seq(a,1,4)
array([[ 0.,  1.,  2.,  3.],
       [ 1.,  2.,  3.,  4.],
       [ 2.,  3.,  4.,  5.],
       [ 3.,  4.,  5.,  6.],
       [ 4.,  5.,  6.,  7.],
       [ 5.,  6.,  7.,  8.]])
>>> pyeeg.embed_seq(a,2,3)
array([[ 0.,  2.,  4.],
       [ 1.,  3.,  5.],
       [ 2.,  4.,  6.],
       [ 3.,  5.,  7.],
       [ 4.,  6.,  8.]])
>>> pyeeg.embed_seq(a,4,1)
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.],
       [ 6.],
       [ 7.],
       [ 8.]])
```

first_order_diff(*X*)

Compute the first order difference of a time series.

For a time series X = [x(1), x(2), ... , x(N)], its first order difference is: Y = [x(2) - x(1) , x(3) - x(2), ..., x(N) - x(N-1)]

fisher_info(*X, Tau, DE, W=None*)

Compute Fisher information of a time series from either two cases below: 1. X, a time series, with lag Tau and embedding dimension DE (default) 2. W, a list of normalized singular values, i.e., singular spectrum (if W is

provided, recommended to speed up.)

If W is None, the function will do as follows to prepare singular spectrum:

First, computer an embedding matrix from X, Tau and DE using pyeeg function embed_seq():

M = embed_seq(X, Tau, DE)

Second, use scipy.linalg function svd to decompose the embedding matrix M and obtain a list of singular values:

W = svd(M, compute_uv=0)

At last, normalize W: W /= sum(W)

Parameters **X** :

list

a time series. X will be used to build embedding matrix and compute singular values if W or M is not provided.

Tau :

integer

the lag or delay when building a embedding sequence. Tau will be used to build embedding matrix and compute singular values if W or M is not provided.

DE :

integer

the embedding dimension to build an embedding matrix from a given series. DE will be used to build embedding matrix and compute singular values if W or M is not provided.

W :

list or array

the set of singular values, i.e., the singular spectrum

Returns **FI** :

integer

Fisher information

See Also:

`embed_seq` embed a time series into a matrix

Notes

To speed up, it is recommended to compute W before calling this function because W may also be used by other functions whereas computing it here again will slow down.

`hfd`(*X, Kmax*)

Compute Hjorth Fractal Dimension of a time series X, kmax is an HFD parameter

`hjorth`(*X, D=None*)

Compute Hjorth mobility and complexity of a time series from either two cases below:

1.X, the time series of type list (default)

2.D, a first order differential sequence of X (if D is provided, recommended to speed up)

In case 1, D is computed by first_order_diff(X) function of pyeeg

Parameters **X** :

list

a time series

D :

list

first order differential sequence of a time series

Returns **As indicated in return line** :

Hjorth mobility and complexity :

Notes

To speed up, it is recommended to compute D before calling this function because D may also be used by other functions whereas computing it here again will slow down.

hurst(*X*)

Compute the Hurst exponent of X. If the output H=0.5,the behavior of the time-series is similar to random walk. If H<0.5, the time-series cover less "distance" than a random walk, vice verse.

Parameters **X** :

list

a time series

Returns **H** :

float

Hurst exponent

Examples

```
>>> import pyeeg
>>> from numpy.random import randn
>>> a = randn(4096)
>>> pyeeg.hurst(a)
>>> 0.5057444
```

in_range(*Template, Scroll, Distance*)

Determines whether one vector is the the range of another vector.

The two vectors should have equal length.

Parameters **Template** :

list The template vector, one of two vectors being compared

Scroll :

list The scroll vector, one of the two vectors being compared

D :

float Two vectors match if their distance is less than D

Bit :

Notes

The distance between two vectors can be defined as Euclidean distance according to some publications.

The two vector should of equal length

pfd(*X, D=None*)

Compute Petrosian Fractal Dimension of a time series from either two cases below:

1.X, the time series of type list (default)

2.D, the first order differential sequence of X (if D is provided, recommended to speed up)

In case 1, D is computed by first_order_diff(X) function of pyeeg

To speed up, it is recommended to compute D before calling this function because D may also be used by other functions whereas computing it here again will slow down.

samp_entropy(*X, M, R*)

Computer sample entropy (SampEn) of series X, specified by M and R.

SampEn is very close to ApEn.

Suppose given time series is X = [x(1), x(2), ... , x(N)]. We first build embedding matrix Em, of dimension (N-M+1)-by-M, such that the i-th row of Em is x(i),x(i+1), ... , x(i+M-1). Hence, the embedding lag and dimension are 1 and M-1 respectively. Such a matrix can be built by calling pyeeg function as Em = embed_seq(X, 1, M). Then we build matrix Emp, whose only difference with Em is that the length of each embedding sequence is M + 1

Denote the i-th and j-th row of Em as Em[i] and Em[j]. Their k-th elments are Em[i][k] and Em[j][k] respectively. The distance between Em[i] and Em[j] is defined as 1) the maximum difference of their corresponding scalar components, thus, max(Em[i]-Em[j]), or 2) Euclidean distance. We say two 1-D vectors Em[i] and Em[j] *match* in *tolerance* R, if the distance between them is no greater than R, thus, max(Em[i]-Em[j]) <= R. Mostly, the value of R is defined as 20% - 30% of standard deviation of X.

Pick Em[i] as a template, for all j such that 0 < j < N - M , we can check whether Em[j] matches with Em[i]. Denote the number of Em[j], which is in the range of Em[i], as k[i], which is the i-th element of the vector k.

We repeat the same process on Emp and obtained Cmp[i], 0 < i < N - M.

The SampEn is defined as log(sum(Cm)/sum(Cmp))

See Also:

ap_entropy approximate entropy of a time series

Notes

Extremely slow computation. Do NOT use if your dataset is not small and you are not patient enough.

References

Costa M, Goldberger AL, Peng C-K, Multiscale entropy analysis of biolgical signals, Physical Review E, 71:021906, 2005

spectral_entropy(*X, Band, Fs, Power_Ratio=None*)

Compute spectral entropy of a time series from either two cases below: 1. X, the time series (default) 2. Power_Ratio, a list of normalized signal power in a set of frequency bins defined in Band (if Power_Ratio is provided, recommended to speed up)

In case 1, Power_Ratio is computed by bin_power() function.

Parameters **Band** :

list

boundary frequencies (in Hz) of bins. They can be unequal bins, e.g. [0.5,4,7,12,30] which are delta, theta, alpha and beta respectively. You can also use range() function of Python to generate equal bins and pass the generated list to this function.

Each element of Band is a physical frequency and shall not exceed the Nyquist frequency, i.e., half of sampling frequency.

X :

list

a 1-D real time series.

Fs :

integer

the sampling rate in physical frequency

Returns **As indicated in return line** :

See Also:

bin_power pyeeg function that computes spectral power in frequency bins

Notes

To speed up, it is recommended to compute Power_Ratio before calling this function because it may also be used by other functions whereas computing it here again will slow down.

svd_entropy(*X, Tau, DE, W=None*)

Compute SVD Entropy from either two cases below: 1. a time series X, with lag tau and embedding dimension dE (default) 2. a list, W, of normalized singular values of a matrix (if W is provided, recommend to speed up.)

If W is None, the function will do as follows to prepare singular spectrum:

First, computer an embedding matrix from X, Tau and DE using pyeeg function embed_seq():

M = embed_seq(X, Tau, DE)

Second, use scipy.linalg function svd to decompose the embedding matrix M and obtain a list of singular values:

W = svd(M, compute_uv=0)

At last, normalize W: W /= sum(W)

Notes

To speed up, it is recommended to compute W before calling this function because W may also be used by other functions whereas computing it here again will slow down.

Contents:

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

INDEX