exemplar_code_for_appendix

May 12, 2023

[ ]:
```
### Importing the necessary Python libraries and data needed for the␣
↪calculations of the time lag

# Python lib import
import csv
import PyQt5
import warnings
import scipy as sc
import os
import math
import sys
from functools import reduce
from math import log
from scipy.stats import rankdata as rd
from scipy.stats import norm as nm
from scipy.stats import ttest_ind as tt
from scipy import optimize as op
from mpl_toolkits.mplot3d import Axes3D
from sklearn.neighbors import NearestNeighbors
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from scipy.signal import argrelmin, argrelmax, find_peaks, detrend
from scipy.optimize import curve_fit, minimize
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
from astropy.io import fits
from astropy.table import Table, Column

# Ignoring warnings printed to screen
warnings.filterwarnings("ignore")

hdulist = fits.open('../../../Real Data/Healthy_Control_Data/Four_dots_static.
↪fits')
data = hdulist[0].data
```

```
# Getting t data
t = np.linspace(0,500,500)

# Extracting x data
x = [ [] for i in range(46)]
x11 = [[] for i in range(46)]

for i in range(46):

    x[i] = data[0,i,0,:]

for i in range(46):

    x11[i] = detrend(x[i], axis=-1, type='linear')

x1 = np.array(x11)

# Extracting y data
y = [ [] for i in range(46)]
y11 = [[] for i in range(46)]

for i in range(46):

    y[i] = data[0,i,1,:]

for i in range(46):

    y11[i] = detrend(y[i], axis=-1, type='linear')

y1 = np.array(y11)

# Finding r data
r1 = np.sqrt(x1**2 + y1**2)
```

```
[ ]: ### Creating a function to calculate mutual information between a time series
     ↪x(t) and x(t + tau)

def MI_single(x0, x1, h='sturge', ranking=True):

    # Determining number of points for each input x0 and x1
    Nx0 = len(x0)
    Nx1 = len(x1)
    if Nx0 == Nx1:
        N = Nx0
    else:
        N = min([Nx0, Nx1])
```

```
    # Performing ranking of data
    if ranking == True:
        x0 = rd(x0, method='ordinal')
        x1 = rd(x1, method='ordinal')

    # Calculating the number of bins to use for the histograms x0, x1 abnd␣
 ↪(x0,x1)
    if h == 'sturge':
        Bx0 = np.log2(Nx0) + 1
        Bx1 = np.log2(Nx1) + 1
        B = [Bx0, Bx1]
        B = np.round(np.array(B)).astype(int)     # Ensuring bin is an integer␣
 ↪value

    # Calculating 1D histograms for x0 and x1
    Hx0 = np.histogram(x1, bins = int(B[0])) [0]
    Hx1 = np.histogram(x1, bins = int(B[1])) [0]

    #  Calculating 2D histogram for (x0, x1)
    Hx0x1 = np.histogram2d(x0, x1, bins = B) [0]

    # Working out the probabilities needed for the AMI factor
    Px0 = Hx0/N
    Px1 = Hx1/N
    Px0x1 = Hx0x1/N

    # Performing the AMI factor calculation
    I_initial = Px0x1*np.log(Px0x1/(Px1*Px1))
    NaN_check = np.isnan(I_initial)                       # Checking which␣
 ↪numbers in data are NaN  np.nansum
    I_initial[NaN_check] = 0                              # Changing NaN entries␣
 ↪in series to 0
    I = sum(sum(I_initial))

    # Finishing off function and returning value I
    return(I)
```

[ ]:

```
### Creating a function to evaluate mutual information factor as a function of␣
 ↪lag

def MI_time_delay(timeseries, plotting=True):

    # Initialising values and constants
    max_delay = 150
    I = []                          # Initialising AMI array
    tau = []                        # Initialising tau array
    delay = 0
```

```
    # Performing the analysis
    while delay < max_delay:
        delay = delay + 1
        x0 = timeseries[:-delay]                    # All terms besides last term
        x1 = timeseries[delay:]                     # All terms besides first term
        I.append(MI_single(x0,x1))
        tau.append(delay)

    return(tau, I)
```

[ ]:
```
### Function to compute takens emedded data for a specified delay and embedding␣
↪dimension

def takens_embedding(data, delay, dimension):

    if delay*dimension > len(data):
        raise NameError('Delay times dimension exceeds length of data')       #␣
↪Ensures that delay is not going to be too large such that it is larger than␣
↪the data length

    embedded_data = np.array([data[0:len(data)-delay*dimension]])

    for i in range(1, dimension):
        embedded_data = np.append(embedded_data, [data[i*delay:len(data) -␣
↪delay*(dimension - i)]], axis=0)

    return embedded_data;
```

[ ]:
```
### Function to calculate percentage of false nearest neighbours for a range of␣
↪embedding dimensions

def false_nearest_neighbors(data,delay,embedding_dimension):

    embedded_data = takens_embedding(data, delay, embedding_dimension);

    nbrs = NearestNeighbors(n_neighbors=2, algorithm='auto').fit(embedded_data.
↪transpose())
    distances, indices = nbrs.kneighbors(embedded_data.transpose())

    epsilon = np.std(distances.flatten())
    nFalseNN = 0

    for i in range(0, len(data)-delay*(embedding_dimension+1)):
```

```
        if (0 < distances[i, 1]) and (distances[i, 1] < epsilon) and (␣
↪(abs(data[i+embedding_dimension*delay] -␣
↪data[indices[i,1]+embedding_dimension*delay]) / distances[i,1]) > 10):
            nFalseNN += 1;
    return nFalseNN
```

[ ]:
```
### Function to compute the power specttrum of the phase space

def power_spectrum(data_array,time):

    fouriert_1 = sc.fft.rfft(data_array, len(time))
    fourier_freq = sc.fft.rfftfreq(len(time), d = 1e-3)
    power_spec = np.abs(fouriert_1)**2

    return power_spec, fourier_freq
```

[ ]:
```
### Performing averaged mutual information in x for each repetition

# Setting up arrays
tau_x = [[] for i in range(46)]
I_x = [[] for i in range(46)]

# Determining averaged mutual information for each repetition
for i in range(46):

    tau_x[i], I_x[i] = MI_time_delay(x1[i])

# Calculating mean averaged mutual information
tau_xav = np.sum(tau_x, axis=0)/46
I_xav = np.sum(I_x, axis=0)/46

# # Plotting mean averaged mutual information
plt.plot(tau_xav, I_xav)
plt.plot(tau_xav[16], I_xav[16], 'x', markersize=12, color='r')
plt.xlabel('$Lag$', fontsize=16)
plt.ylabel('<Mutual information> ($nats$)', fontsize=16)
plt.title('Mutual information plot - x data', fontsize=20, pad=20)
plt.xlim(0,60)
```

[ ]:
```
### Determining the values of tau for x data repetitions

tau_xvalues = [[] for i in range(46)]

for i in range(46):

    tau_xvalues[i] = MI_minima(tau_x[i], I_x[i])
```

```
# Calculating mean value of tau
tau_xmean = np.sum(tau_xvalues)/46

# Calculating standard deviation of tau
tau_sdx = np.std(tau_xvalues)

print('Mean lag in x direction =', np.int(tau_xmean))
print(tau_sdx)
```

[ ]:
```
### False Nearest Neighbours for each x repetition

# Setting up arrays
nFNN_x = [[] for i in range(46)]

# Calculating percentage of false nearest neighbours for each repetition
for i in range(46):

    for j in range (1,7):

        nFNN_x[i].append(false_nearest_neighbors(x1[i],tau_xvalues[i],j) /␣
 ↪len(x1[i]))

# Calculating mean false nearest neighbours
nFNN_xav = np.sum(nFNN_x, axis=0)/46

nFNN_xav1 = [0.65, nFNN_xav[1], nFNN_xav[2], nFNN_xav[3], nFNN_xav[4],␣
 ↪nFNN_xav[5]]

print(nFNN_xav)

# Plotting mean false nearest neighbours
plt.plot(range(1,7), nFNN_xav1)
plt.plot(3,nFNN_xav1[2],'x', color='red', markersize=10)
plt.xlabel('embedding dimension', fontsize=16);
plt.ylabel('<Fraction of fNN>', fontsize=16);
plt.title('False nearest neighbours vs dimension - x data', fontsize=20, pad=20)
```

[ ]:
```
### Performing phase space reconstruction for x data

# Embedding data
embedded_x_final = []

for i in range(46):

    embedded_x_final.append(takens_embedding(x1[i],tau_xvalues[i],3))
```

```
# Plotting repetiton specific phase spaces
fig = plt.figure()
fig.set_size_inches(8,6)
ax = plt.axes(projection='3d')
ax.plot3D(embedded_x_final[10][0], embedded_x_final[10][1],␣
 ↪embedded_x_final[10][2])
ax.set_xlabel('$x(t)$', fontsize=14)
ax.set_ylabel('$x(t+$'+str(tau_xvalues[0])+')', fontsize=14)
ax.set_zlabel('$x(t+$'+str(2*tau_xvalues[0])+')', fontsize=14)
ax.set_title('Parkinsons - x', fontsize=20, pad=40)
```

[ ]:
```
### Power spectrum for each x repetition

# Setting up arrays
power_spectrum_x = []
freq_x = [[] for i in range(46)]
power_x = [[] for i in range(46)]
sum_fpx = [[] for i in range(46)]
sum_fx = [[] for i in range(46)]
mean_freqx = [[] for i in range(46)]
period_x = [[] for i in range(46)]


# Calculating the power spectrum for x data
for i in range(46):

    power_spectrum_x.append(power_spectrum(x1[i],t))

for i in range(46):

    freq_x[i] = power_spectrum_x[i][:][:][:][1]
    power_x[i] = power_spectrum_x[i][:][:][:][0]

for i in range(46):

    sum_fpx[i] = np.sum(freq_x[i]*power_x[i])
    sum_fx[i] = np.sum(freq_x[i])

for i in range(46):

    mean_freqx[i] = sum_fpx[i]/sum_fx[i]
    period_x[i] = 1/mean_freqx[i]
```

[ ]:
```
### Nearest neighbours for x data

# Setting up arrays
```

```
nearest_neighbours_x = [[] for i in range(46)]
distances_x = [[] for i in range(46)]
indices_x = [[] for i in range(46)]

# Calculating nearest neighbours
for i in range(46):

    for j in range(len(x1[i])):

        nearest_neighbours_x[i] = NearestNeighbors(n_neighbors=200,␣
↪algorithm='auto').fit(embedded_x_final[i].transpose())
        distances_x[i], indices_x[i] = nearest_neighbours_x[i].
↪kneighbors(embedded_x_final[i].transpose())
```

[ ]:
```
### Divergence for x data

# Initialising arrays needed for divergence calculation
N = 300
separation_x1 = [ [] for i in range(N)]
lags_x = []
xx_1 = [ [] for i in range(N)]
lyapunovs_x = []
lyapunovs_xerr = []
all_divergencex = [[] for i in range(46)]


for k in range(46):

    separation_x1 = [[] for i in range(N)]
    lags_x = []
    xx_1 = [[] for i in range(N)]

    eps = period_x[k]
    oooo = embedded_x_final[k]
    indi = indices_x[k]

    # Extracting time differences between nearest neighbours
    for i in range(N):
        xx_1[i] = indi[i] - i

    xx_2 = np.array(xx_1)
    times_x = xx_2*1e-3

    for i in range(N):

        m_x = 0
```

```
        while np.abs(times_x[i][m_x]) < eps and m_x < 199:

            m_x = m_x + 1

        lags_x.append(times_x[i][m_x])

    lags_x1 = np.array(lags_x)/1e-3
    lags_x2 = lags_x1.astype(int)


    # Calculating the divergence for Lyapunov exponent calculation
    for i in range(0,N):

        for j in range(0,N):

            try:

                divv1 = np.sqrt((oooo[0,i+j+lags_x2[i]] - oooo[0,i+j])**2 +␣
↪(oooo[1,i+j+lags_x2[i]] - oooo[1,i+j])**2 + (oooo[2,i+j+lags_x2[i]] -␣
↪oooo[2,i+j])**2)
                separation_x1[j].append(divv1)

            except IndexError:
                separation_x1[j].append(np.nan)


    sep_x1 = np.array(separation_x1)

    # Taking the logarithm of the divergence array
    logsep_x1 = [ [] for i in range(len(sep_x1))]

    for i in range(N):

        logsep_x1[i] = np.log(sep_x1[i])

    logsep_x1 = np.array(logsep_x1)
    df_x=pd.DataFrame(logsep_x1)
    df_x_interpolate=df_x.interpolate(limit_direction='both')
    logsep_x2=pd.DataFrame.to_numpy(df_x_interpolate)

    # Calculating the averaged ln(divergence)
    av_log_div_x = np.mean(logsep_x2, axis = 1)
    df_xx=pd.DataFrame(av_log_div_x)
    df_xx=df_xx.replace(-np.inf, np.nan)
    df_xx_interpolate=df_xx.interpolate(limit_direction='both')
    av_log_div_x2=pd.DataFrame.to_numpy(df_xx_interpolate)
```

```
    # Performing the linear regression calculation
    t_regx = t[0:60].reshape(-1,1)
    divx_reg = av_log_div_x2[0:60].reshape(-1,1)

    reg_x = LinearRegression().fit(t_regx, divx_reg)
    grad_x = reg_x.coef_.item()
    intercept_x = reg_x.intercept_.item()

    resx = av_log_div_x2[0:60] - (t[0:60]*grad_x + intercept_x)
    resx_sq = np.sum(resx**2)
    tmeanx = np.mean(t[0:60])
    ttx = np.sum((t[0:60]-tmeanx)**2)

    error_x = np.sqrt((1/58)*(resx_sq/ttx))
    lyapunovs_x.append(grad_x)
    lyapunovs_xerr.append(error_x)
    all_divergencex[k].append(av_log_div_x2)

lyapunovs_xmean = np.mean(lyapunovs_x)
lyapunovs_sdx = np.std(lyapunovs_x)

print('The mean value of lyapunov exponents for x repetitions is',␣
 ↪lyapunovs_xmean)
print(lyapunovs_sdx)
```

[ ]:
```
### Plotting averaged divergence for x data

av_divx = np.sum(all_divergencex, axis=0)/46

av_divx1=av_divx.reshape(300,)

# Performing the linear regression calculation
t_regx1 = t[2:40].reshape(-1,1)
divx_reg1 = av_divx1[2:40].reshape(-1,1)

reg_x1 = LinearRegression().fit(t_regx1, divx_reg1)
grad_x1 = reg_x1.coef_.item()
intercept_x1 = reg_x1.intercept_.item()

resx1 = av_divx1[2:40] - (t[2:40]*grad_x1 + intercept_x1)
resx_sq1 = np.sum(resx**2)
tmeanx = np.mean(t[2:40])
ttx1 = np.sum((t[2:40]-tmeanx)**2)

error_x1 = np.sqrt((1/38)*(resx_sq1/ttx1))

plt.plot(t[1:100], av_divx1[1:100])
```

```
plt.plot(t[2:40], t[2:40]*grad_x1 + intercept_x1)
plt.xlabel('Time (ms)', fontsize=16)
plt.ylabel('<ln(divergence)> (arcmin)', fontsize=16)
plt.title('Divergence of x data trajectories', fontsize=20, pad=20)

rscore_x = r2_score(av_divx1[2:40], t[2:40]*grad_x1 + intercept_x1)
mean_errx = mean_squared_error(av_divx1[2:40], t[2:40]*grad_x1 + intercept_x1)

print(grad_x1)
print(mean_errx)
print(rscore_x)
print(error_x1)
```

[ ]:
```
### Performing T-test between Parkinso's and control participant for x data

t_patientx = lyapunovs_x
t_controlx = [0.01765627331455689, 0.012681150466437757, 0.020516774227012393,␣
↪0.01497865336011384, 0.024826263834485114, 0.003848403480972742, 0.
↪0075960087742760205, 0.005290220126460295, 0.010024605869547747, 0.
↪0046611347492231375, 0.01438726685116858, 0.00778425412962644l, 0.
↪012539216669978632, 0.0165496486747158, 0.019613956695817572, 0.
↪014048506763748882, 0.013387101499709127, 0.0031957656049769246, 0.
↪011314532168966975, 0.012329519111002822, 0.014217457729683282, 0.
↪012502313802872785, 0.010575841251180645, -0.0018117458155710 69, 0.
↪007358029269456156, 0.01482539304261262, 0.015682957303024214, 0.
↪007349580045030007, 0.016981321085743385, 0.008148401209969791, 0.
↪007409589823571574, 0.01650054938199119, 0.012896785302120472, 0.
↪008804631259729022, 0.013786305113871196, 0.015854212036596618, 0.
↪009345630588459495, 0.02129832636742016, 0.014684630889070334, 0.
↪010536314699870123, 0.012130646252518248, 0.010468336607669083, 0.
↪0033527344517116434, 0.024080092867893536, 0.013161449552121143, 0.
↪013348949519345897]

mu_patientx = np.mean(t_patientx)
mu_controlx = np.mean(t_controlx)

sd_patientx = np.std(t_patientx)
sd_controlx = np.std(t_controlx)

ttest_x = tt(t_patientx, t_controlx, equal_var=False)

print(ttest_x)
```

[ ]:
```
### Plotting the histograms and fitted distributions for x data

_, bins_xp, _ = plt.hist(t_patientx, 30, density=1, alpha=0.2, color='r',␣
↪histtype='bar', ec='r')
```

```python
_, bins_xc, _ = plt.hist(t_controlx, 30, density=1, alpha=0.2, color='b',
    histtype='bar', ec='b')

mu_xp, sigma_xp = nm.fit(t_patientx)
mu_xc, sigma_xc = nm.fit(t_controlx)

best_xp = nm.pdf(bins_xp, mu_xp, sigma_xp)
best_xc = nm.pdf(bins_xc, mu_xc, sigma_xc)

plt.plot(bins_xp, best_xp, color='r', linewidth=2.5)
plt.plot(bins_xc, best_xc, color='b', linewidth=2.5)
plt.legend(['Parkinson', 'Control'])
plt.xlabel('LLE (arcmin/ms)', fontsize=16)
plt.ylabel('Density', fontsize=16)
```