Příloha 6

Kódové provedení ptool

change_order – algoritmus ke změně řádu dat (vrátí rozdíl mezi datovými body)
learning_entropy_2 – algoritmus pro výpočet learning entropy
buffer – transformuje vektor na matici
debuffer – transformuje matici na vektor
clear_noise_rough – filtrace dat pomoci PCA
fourier_transformation – FFT
find_fft_peaks – hledá zadaný počet nejvíce významných frekvenci v FFT
new_fft_peaks – porovná frekvence ve dvou vektorech a vrátí ty, které se odlišují
return_roughness – algoritmus, který vrací šum ze signálu pro detailnější analýzu
calculate_Ra – výpočet koeficientu drsnosti Ra
calculate_Rq – výpočet koeficientu drsnosti Rq
calculate_Rdq – výpočet koeficientu drsnosti Rdq
calculate_Skew – výpočet koeficientu drsnosti šikmosti
calculate_Kurt – výpočet koeficientu drsnosti špičatnosti

```
# This code contains partial tools that are used for the main
# predictor code called mtools.py

import numpy as np
import itertools
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from scipy import signal, stats


def change_order(data, order: int, drop_na: bool = True):
    """
    Differentiates the data by shifting and subtracting two offset datasets
from
    each other.

    Parameters
    ----------------
    data : array-like
        Integer or float array.

    order : int
        Sets up what difference order will be used.

    drop_na : bool
        Allows to drop nan values.

    """
    ordered_data = np.array(data).astype(float)

    for step in range(order - 1):
        ordered_data[1:] = abs(ordered_data[1:] - ordered_data[:-1])
        if step == 0: ordered_data[0] = float('NaN')

    if drop_na:
        ordered_data = ordered_data[order:]
```

```
    return ordered_data


def learning_entropy_2(data, beta_range=np.arange(0.92, 2.23, 0.41),
order=1):
    """
    Learning entropy calculator based on second method of calculation.

    Parameters
    ------------
    data : array-like
        Array representing learning effort. Means absolute value of
        overall weight differences in each step.
        1st dimension of array must be time dimension
        2nd dimension of array is individual for number of weights

    beta_range : range-like
        Bandwidth levels. Every value of dw that is higher that mean +
beta*std
        is added to Entropy level.
        Corresponds to sensitivity.

    order : int
        Order of data that will be counted. Min value = 1
    """

    ordered_data = change_order(data, order)

    E = np.zeros((max(ordered_data.shape), 1))
    normalized_data = (ordered_data - np.mean(ordered_data)) /
np.std(ordered_data)

    mean = [np.mean(X) for X in normalized_data.T]
    std = [np.std(X) for X in normalized_data.T]

    for k in range(normalized_data.shape[0]):
        for beta in beta_range:
            for i in range(normalized_data.shape[1]):
                dw = normalized_data[k][i]
                if dw > mean[i] + beta*std[i]:
                    E[k] += 1

    E = E / (len(beta_range) * max(normalized_data.shape))
    return E


def buffer(n, numpy_array, nan_row=True):
    """
    Takes 1D array and stacks it up to 2D matrix.

    Example
    ---------
    #1
    Call:
        buffer(2, [0.1, 0.2, 0.3], True)

    Out1:
        [[0.1, 0.2],
         [0.3, nan]]

    Out2:
```

```
        None


    #2
    Call:
        buffer(2, [0.1, 0.2, 0.3], False)

    Out1:
        [[0.1, 0.2]]

    Out2:
        [0.3]
    """
    args = [iter(numpy_array.flatten())] * n
    buffed = np.array(list(itertools.zip_longest(fillvalue=float('nan'),
*args)))

    if (not nan_row) & (max(numpy_array.shape) % n > 0):
        return buffed[:-1], buffed[-1][~np.isnan(buffed[-1])]
    else:
        return buffed, None


def debuffer(numpy_array, suffix=None):
    """
    Takes 2D matrix and de-stacks it into 1D array.
    This is a counter function to buffer().

    Example
    ----------
    #1
    Call:
        debuffer([[0.1, 0.2],
                  [0.3, nan]])

    Out:
        [0.1, 0.2, 0.3]


    #2
    Call:
        debuffer([[0.1, 0.2], suffix=[0.3])

    Out:
        [0.1, 0.2, 0.3]
    """

    if type(suffix) == type(None):
        return numpy_array[~np.isnan(numpy_array)]
    else:
        return np.concatenate((numpy_array[~np.isnan(numpy_array)],
suffix), axis=0)


def clear_noise_rough(data, **kwargs):
    """
    PCA based noise filter. This filter is rough i.e., the output still
contains
    noise and unusual peaks.

    Parameters
```

```
    --------------
    data : 1D array
        One dimensional array vector.

    Keyword Arguments
    -------------------
    buff : int, Optional
        Parameter that corresponds to the strength of the
        filter. Buff is the size of one side of the matrix
        that is made from data for PCA process.

        If buff==None then it is selected automatically.

    normalize : bool
        Normalization of the data could enhance and equalize the
        unusialities in signal.
        Default: True

    pca_par : float
        Parameter that changes the filter roughness.
        Default: 0.3

    buff_par : float
        Parameter that changes the filter strength. When buff size is
        selected automatically.
        Default: 0.05

    """

    # Definition of filter parameters (kwargs decoding)
    buff = None if type(kwargs.get('buff')) == type(None) else
int(kwargs.get('buff'))
    pca_par = float(0.3 if type(kwargs.get('pca_par')) == type(None) else
kwargs.get('pca_par'))
    buff_par = float(0.05 if type(kwargs.get('buff_par')) == type(None)
else kwargs.get('buff_par'))
    normalize = bool(True if type(kwargs.get('normalize')) == type(None)
else kwargs.get('normalize'))

    data_len = max(data.shape)
    buff = int(buff_par * data_len) if type(buff) == type(None) else buff

    # Normalization of the data can enhance the results
    if normalize:
        scale = StandardScaler().fit(data)
        data_normalized = scale.transform(data)
    else:
        data_normalized = data

    # In order to use PCA we need to stack up 1D data into 2D matrix
    data_buffed, suff = buffer(buff, data_normalized, False)

    # Components selection is based on algorithm to create rough PCA filter
    components = int(pca_par * min(buff, int(data_len / buff)))

    # PCA filtering
    pca = PCA(components).fit(data_buffed)
    data_compressed = pca.transform(data_buffed)
    data_decompressed = pca.inverse_transform(data_compressed)

    data_debuffed = debuffer(data_decompressed, suff).reshape(-1, 1)
```

```
    if normalize:
        data_denormalized = scale.inverse_transform(data_debuffed)
    else:
        data_denormalized = data

    return data_denormalized


def fourier_transformation(data, timestep):
    """
    Creates fourier transformation of provided data (1D) and
    returns power spectral density (psd) and frequencies (freq).

    """
    r = data.shape[0]
    freq = np.fft.fftfreq(r, timestep)[0:int(r / 2)]
    psd = abs(np.fft.fft(data))[0:int(r / 2)]
    return psd, freq


def find_fft_peaks(psd, freq, rarity: float = 6):
    """
    Return list of frequencies and their magnitudes.

    Parameters
    -------------
    psd : array-like
        Power spectral density that is an output from
        fourier_transformation

    freq : array-like
        Frequencies form fourier_transformation

    rarity : float
        To recognize what is a peak and what is a noise
        we use rarity estimation with standard deviation.
        Rarity corresponds to multiplication of std i.e.,
        probability that interval is normal.


    Returns
    ------------
    magnitudes : numpy-array
        Array of magnitudes of recognized frequencies.

    frequencies : numpy-array
        Array of recognized frequencies.

    """
    top_psd = psd >= np.mean(psd) + rarity * np.std(psd)
    frequencies = np.round(freq[top_psd], 1)
    magnitudes = psd[top_psd]

    return magnitudes, frequencies


def new_fft_peaks(freqs_1, freqs_2, psds_2):
    """
    Finds peaks that are in freqs_2, but not in freqs_1.
    Returns new frequencies and its magnitudes.
```

```
    Note:   Repetitive frequencies will be ignored and frequency
            with highest magnitude will be considered.

    Parameters
    ------------
    freqs_1 : array-like
        Output from find_fft_peaks. This is array of
        original frequencies.

    freqs_2 : array-like
        Output from find_fft_peaks. This is array of
        actual frequencies.

    psds_2 : array-like
        Output from find_fft_peaks. This is array of
        actual power spectral density.

    Returns
    -----------
    new_psds : numpy-array
    new_frequencies : numpy-array

    """
    list1_norep = list(dict.fromkeys(freqs_1))
    list2_norep = list(dict.fromkeys(freqs_2))
    new_frequencies = np.array([x for x in list2_norep if x not in
list1_norep])
    new_psds = np.zeros(len(new_frequencies))

    for n, frq in enumerate(new_frequencies):
        new_psds[n] = np.max(psds_2[freqs_2 == frq])
    return new_psds, new_frequencies


def return_roughness(dataset, filter_order=5, crit_freq=.06):
    """
    Return roughness of a function, using lowpass filter to eliminate
wavines.
    roughness, waviness = return_roughness() is returned.
    """

    pars = signal.butter(filter_order, crit_freq)

    try:
        sig = np.zeros((dataset.shape[0], dataset.shape[1]))
    except IndexError:
        raise IndexError(' Dataset should be at least 1D. Try
add .reshape(-1,1)')

    for i in range(dataset.shape[1]):
        sig[:, i] = signal.filtfilt(pars[0], pars[1], dataset[:, i])

    return dataset - sig, sig


def calculate_Ra(roughness):
    return np.mean(abs(roughness))


def calculate_Rq(roughness):
```

```
    return np.sqrt(np.mean(roughness ** 2))


def calculate_Rdq(roughness):
    slopes = np.zeros(len(roughness) - 1)
    for x in range(1, len(roughness)):
        x0 = x - 1
        x1 = x
        y0 = roughness[x0]
        y1 = roughness[x1]
        slopes[x - 1] = np.arctan((y1 - y0) / (x1 - x0)) / np.pi * 180
    return np.sqrt(np.mean(slopes ** 2))


def calculate_Skew(roughness):
    return 3 * (np.mean(roughness) - np.median(roughness)) /
np.std(roughness)


def calculate_Kurt(roughness):
    return stats.kurtosis(roughness)[0]
```