```
import os
import math
import numpy as np
import pandas as pd
from tqdm import tqdm
from pathlib import Path
from random import randrange
import matplotlib.pyplot as plt

from scipy.fft import fft
from scipy import signal, ndimage
from scipy.signal import argrelextrema, hilbert
from scipy.stats import moment, zscore, mode, iqr

from scripts.data_handling import resample_axis
from scripts.preprocessing import preprocess_ecg, preprocess_gsr, preprocess_resp, preprocess_emg, butter_filter


'''
This file implements several hand-crafted-feature approaches to analyze physiological sensors in the context of pain recognition.


For example the paper 'Automatic Pain Recognition from Video and Biomedical Signals' is used.
https://www.researchgate.net/publication/268513442_Automatic_Pain_Recognition_from_Video_and_Biomedical_Signals

'''
```

```
#-----------------------------------------------------------------------------------------------------

# Supporting functions

#-----------------------------------------------------------------------------------------------------


def moving_average(x, w) :

    """

    Moving average for a one dimensional numpy vector.

    Each element in x is updated by calculating the mean of a window of elements with size 'w' centered around the element.

    Window is cropped to the start and end of 'x'.


    Parameters

    ----------


    x: np. Vector which will be modified by the moving_average.


    w: int. Size of the window.


    Returns

    -------

    x: np. Vector.

    """

    assert x.ndim==1


    x = np.pad(x, (w//2, w//2 - 1), 'constant', constant_values=(x[0], x[-1]))

    x = np.convolve(x, np.ones(w)/w, mode='valid')


    return x
```

```
def RMS_envelope(x, w) :
    """
    RMS enevlope for a one dimensional numpy vector.
    Each element in x is updated by calculating the RMS of a window of elements with size 'w' centered around the element.
    Window is cropped to the start and end of 'x'.

    Parameters
    ----------

    x: np. Vector which will be modified by the moving_average.

    w: int. Size of the window.

    Returns
    -------
    x: np. Vector.
    """
    assert x.ndim==1

    l = len(x)
    result = np.zeros(l)
    for i in np.arange(l):
        start = max(0, i - w//2)
        end = min(l, i + w//2)
        result[i] = math.sqrt(np.mean(x[start:end])**2)
    return result

def normalize(x):
```

'''

Function to normalize a dataframe.

Dataframe can be of type list, Pandas or Numpy.

If a Pandas Dataframe or list is given, it is converted to numpy.

Expects either 1 or 2 dimensional data. If a dataframe with two dimensions is given normalization is done along the first axis.

Parameters

----

x: np, df or list. Data to normalize.

Returns

Normalized data.

'''

```
if isinstance(x, pd.Series):
    x = x.to_frame()

if isinstance(x, pd.DataFrame):
    result = x.copy()
    for feature_name in result.columns:
        max_value = result[feature_name].max()
        min_value = result[feature_name].min()
        if max_value != min_value:
            result[feature_name] = (result[feature_name] - min_value) / (max_value - min_value)
```

```
                else:
                        result[feature_name] = 0
        return result


    if isinstance(x, list):
        x = np.array(x)
    if isinstance(x, np.ndarray):
        x = x.astype(float)
        if len(x.shape) == 1:
            x = x[np.newaxis, ...]
        if len(x.shape) == 2:
            for i, column in enumerate(x):
                max_value = np.max(column)
                min_value = np.min(column)
                if max_value != min_value:
                    x[i, :] = (column - min_value) / (max_value - min_value)
                else:
                    x[i, :] = 0
            if x.shape[0]==1:
                return x[0]
            return x
        print("Shape of len 1 or 2 was expected. Not implemented for shape: ", x.shape)
        return None


def best_fit_slope(xs,ys):
    '''
    Simple solution to fit linear curve through data and return the slope of it.
    Found on https://pythonprogramming.net/how-to-program-best-fit-line-machine-learning-tutorial/.
```

```
        Parameters
        ----------

        xs: np. x data

        xy: np. y data.

        Returns
        -------
        int slope.
        '''
        m = (((np.mean(xs)*np.mean(ys)) - np.mean(xs*ys)) /
                ((np.mean(xs)*np.mean(xs)) - np.mean(xs*xs))) if (np.mean(xs)*np.mean(xs)) -
np.mean(xs*xs) != 0 else 0
        return m


def calc_hr_peaks(x, sampling_rate, plot= False):
        '''
        ECG R-peak segmentation algorithm.
        Follows the approach by Engelse and Zeelenberg [EnZe79]_ with the
        modifications by Lourenco *et al.* [LSLL12]_.

        .. [EnZe79] W. Engelse and C. Zeelenberg, "A single scan algorithm for
                QRS detection and feature extraction", IEEE Comp. in Cardiology,
                vol. 6, pp. 37-42, 1979
        .. [LSLL12] A. Lourenco, H. Silva, P. Leite, R. Lourenco and A. Fred,
                "Real Time Electrocardiogram Segmentation for Finger Based ECG
                Biometrics", BIOSIGNALS 2012, pp. 49-54, 2012
```

```
Further information:
# https://biosignalsplux.com/learn/notebooks/Categories/Detect/r_peaks_rev.php
# https://www.biorxiv.org/content/biorxiv/early/2019/08/06/722397.full.pdf

Parameters
----------

x: np. 1D ecg signal.

plot: bool. Whether a plot should be shown.

Returns
-------
np peaks.
'''

assert x.ndim==1

df = pd.DataFrame()
df["input"]= x

# --- Differentiated - x[n]-x[n-4]
y1 = np.zeros(len(x))
y1[4:] = x.flat[4:] - x.flat[:-4]
df["Diff"]= y1

# Low pass filter - five tap FIR windowed smoothing filter with the coefficients= [1,4,6,4,1]
# https://github.com/PIA-Group/BioSPPy/blob/master/biosppy/signals/ecg.py
```

```
        # y2[n]= sum ci*y1[n-1]
        #c = [1, 4, 6, 4, 1, -1, -4, -6, -4, -1]
        y2 = np.zeros(len(df["Diff"]))
        y2[:-4] = np.convolve(df["Diff"].values,[1, 4, 6, 4, 1], 'valid')
        df["fir"] = y2

        # Other approach in NeuroKit:
        #
https://github.com/neuropsychology/NeuroKit/blob/a04760bd83c544253ca919d214a46eb70cc06c39/neurokit2/ecg/ecg_findpeaks.py#L653
        '''
        ci = [1, 4, 6, 4, 1]
        low_pass = signal.lfilter(ci, 1, df["Diff"])
        low_pass[: int(0.2 * sampling_rate)] = 0
        df["fir"] = low_pass
        '''

        # peak detection: y2[n]>Th, with Th=0.6*max(y2[n])
        # Changed this value from 0.6 to 0.4
        Th = 0.4 * max(y2)
        i=0
        peaks = []
        nthi = None
        nthf = None
        while i < len(y2):

                if y2[i] > Th and nthi is None:
                        nthi = i
```

```
        if y2[i] < Th and nthi is not None:
                nthf = i

        # found interval [nthi;nthf]
        if nthf is not None and nthi is not None:

                # 160 ms window to the right of nthf
                start, end = nthf, int(nthf + (0.160 * sampling_rate))
                if plot:
                        plt.axvline(x=start, color='slategrey', linestyle='dashed')
                        plt.axvline(x=end, color='slategrey', linestyle='dashed')
                window = y2[start : end]
                # at least 10 ms of consecutive points with condition y2[n]< -Th
                # condition
                cond = window < - Th
                # consecutive
                if sum(cond) >= math.floor(0.010 *sampling_rate):
                        peak = (start + np.argmax(x[start:end]), np.max(x[start:end]))
                        peaks.append(peak)

                i= end
                nthi = None
                nthf = None
        i += 1

peaks_np = np.array(peaks)
p_list = []
q_list = []
s_list = []
```

```
t_list = []

# Return empty lists if there are no peaks
if peaks_np.size == 0:
        return p_list, q_list, peaks_np, s_list, t_list

# find p,q,s and t
qs_range = (0.100 * sampling_rate) # QRS complex is 100 ms for a healthy person
for r in peaks_np[:, 0]:
        half_qrs = int(qs_range//2)
        q_window_start = max(0, int(r - half_qrs))
        q = np.argmin(x[q_window_start : int(r)]) + q_window_start
        q_list.append([q, x[q]])

        s = np.argmin(x[int(r) : int(r + half_qrs)]) + int(r)
        s_list.append([s, x[s]])

        p_window_start = max(0, q - sampling_rate//4)
        p_window_end = max(1, q - sampling_rate//20)
        p = np.argmax(x[p_window_start : p_window_end]) + p_window_start
        p_list.append([p, x[p]])

        t_window_end = min(len(x), s + sampling_rate//3)
        t = np.argmax(x[s : t_window_end]) + s
        t_list.append([t, x[t]])

[p_list, q_list, s_list, t_list] = [np.array(i) for i in [p_list, q_list, s_list, t_list]]

# Plot peaks
```

```
    if plot:
        #plt.axhline(y=Th, color='r', linestyle='-')
        if peaks_np.size > 0:
            plt.scatter(p_list[:, 0], p_list[:, 1], marker='o', c="black", zorder=10, label="P")
            plt.scatter(q_list[:, 0], q_list[:, 1], marker='o', c="green", zorder=10, label="Q")
            plt.scatter(peaks_np[:, 0], peaks_np[:, 1], marker='o', c="orange", zorder=10,
label="R-Peaks")
            plt.scatter(s_list[:, 0], s_list[:, 1], marker='o', c="purple", zorder=10, label="S")
            plt.scatter(t_list[:, 0], t_list[:, 1], marker='o', c="red", zorder=10, label="T")
        plt.legend()

    if len(peaks) == 0:
        plt.plot(x)
        plt.title("No peaks could be found in the following ECG signal:")
        plt.show()
        print("'peaks' list is empty.\nIs the ECG signal fine? Is it upside down?")

    return p_list, q_list, peaks_np, s_list, t_list

def power_spectrum(y, sampling_rate, show_plot = False):
    '''
    Calculates a Single-Sided Amplitude Spectrum of y(t)
    From https://glowingpython.blogspot.com/2011/08/how-to-plot-frequency-spectrum-with.html

    Parameters
    ----------

    x: np. 1D ecg signal.
```

sample_rate: int. Sample rate.

secs: int. Seconds of the given time window.

show_plot: bool. Whether a plot should be shown.

Returns

----

np power spectrum.

'''

```
secs = len(y)/sampling_rate
t = np.arange(0, secs, 1.0/sampling_rate)
n = len(y) # length of the signal
k = np.arange(n)
T = n/sampling_rate
frq = k/T # two sides frequency range
frq = frq[range(int(n//2))] # one side frequency range

Y = fft(y)/n # fft computing and normalization
Y = Y[range(int(n//2))]

if show_plot:
        ax1 = plt.subplot(2,1,1)
        ax1.plot(t, y)
        ax1.set_xlabel('Time')
        ax1.set_ylabel('Amplitude')
        ax2 = plt.subplot(2,1,2)
        ax2.plot(frq, abs(Y),'r')
        ax2.set_xlabel('Freq (Hz)')
```

```
                ax2.set_ylabel('|Y(freq)|')
                plt.show()


        return abs(Y)

#-------------------------------------------------------------------------------------------------------
# Extraction methods for GSR, ECG, EDA
#-------------------------------------------------------------------------------------------------------

def gsr_feature_extraction(x, sampling_rate, name="", plot= False):
        '''
        Function to calculate features for an EDA signal.
        Different features are calculated for the given signal 'x'.

        Parameters
        ----------
        x: np. Sensor data.
        sampling_rate: Int. Sampling rate of the given sensor data.
        name: String. String to put a name in front of computed features. Default is ''.
        plot: Bool. Boolean that describes whether to plot the outcome of the analysis or not. Default is
False.

        Returns
        ----------
        result: df. Pandas Dataframe describing the computed features.
        '''

        df =  pd.DataFrame()
```

```
# Plot one example
if plot:
        example = x[randrange(x.shape[0])]
        example = preprocess_gsr(example, sampling_rate= sampling_rate, plot= True)
        plt.legend()
        plt.title("EDA preprocessing")
        plt.show()

        analyze_EDA(example, sampling_rate= sampling_rate, plot= True)
        plt.show()

# max
df[name + "_max"] = np.max(x, axis=1)
# min
df[name + "_min"] = np.min(x, axis=1)
# range
df[name + "_range"] = (np.max(x, axis=1)-np.min(x, axis=1))
# std
df[name + "_std"] = (np.std(x, axis=1))
# iqr
df[name + "_iqr"] = (iqr(x, axis=1))
# mean
df[name + "_mean"] = np.mean(x, axis=1)

print("Finished simple EDA metrics. Start computing more complex ones...")

# rms
df[name + "_rms"] = (np.sqrt(np.mean(x**2, axis=1)))
```

```
        # local maxima

        df[name + "_local_max"] = [np.mean(argrelextrema(i, np.greater)) if argrelextrema(i,
np.greater)[0].size>0 else 0 for i in x]

        # local minima

        df[name + "_local_min"] = [np.mean(argrelextrema(i, np.less)) if argrelextrema(i,
np.less)[0].size>0 else 0 for i in x]

        # mean absolut value

        df[name + "_mean_abs"] = (np.mean(np.abs(x), axis= 1))

        # mean absolut values of the first differences

        df[name + "_mean_abs_1_diff"] = np.mean(np.abs(x[:, 1:] - x[:, :-1]), axis= 1)

        # mean absolut values of the second differences

        df[name + "_mean_abs_2_diff"] = np.mean(np.abs(x[:, 2:] - x[:, :-2]), axis= 1)

        # zscore (relative to the sample mean and standard deviation) as standardization method

        standardized_gsr = zscore(x)

        # mean absolut values of the first differences -standardized - z score

        df[name + "_mean_abs_1_diff_std"] = np.mean(np.abs(standardized_gsr[:, 1:] -
standardized_gsr[:, :-1]), axis= 1)

        # mean absolut values of the second differences -standardized - z score

        df[name + "_mean_abs_2_diff_std"] = np.mean(np.abs(standardized_gsr[:, 2:] -
standardized_gsr[:, :-2]), axis= 1)

        # Variation of the first and second moment of the signal over time

        df[name + "_var_mom"] = np.apply_along_axis(lambda x: np.var([moment(x, 1), moment(x, 2)]),
arr=x, axis=1)


        # ------------ Selfmade

        # Location max (selfmade)

        df[name + "_argmax"] = np.argmax(x, axis= 1)

        # Location min (selfmade)

        df[name + "_argmin"] = np.argmin(x, axis= 1)

        # Difference start, end (selfmade)
```

```
df[name + "_diff_start_end"] = x[:, -1]-x[:, 0]


# Further analysis: For examlpe compute Tonic and phasic components
print("Start EDA decomposition...")
features = [analyze_EDA(i, sampling_rate= sampling_rate) for i in tqdm(x)]
# Unpack results
phasic = [i[0] for i in features]
tonic = [i[1] for i in features]
peaks = [i[2] for i in features]
onsets = [i[3] for i in features]
offsets = [i[4] for i in features]
rise_times = [i[5] for i in features]
amplitudes = [i[6] for i in features]
recovery = [i[7] for i in features]
half_recovery = [i[8] for i in features]


# Mean phasic - tonic
df[name + "_mean_phasic"] = [np.mean(x) for x in phasic]
df[name + "_mean_tonic"] = [np.mean(x) for x in tonic]
# STD phasic - tonic
df[name + "_std_phasic"] = [np.std(x) for x in phasic]
df[name + "_std_tonic"] = [np.std(x) for x in tonic]
# Range tonic
df[name + "_range_tonic"] = [x[-1]-x[0] for x in tonic]
# Number of GSR-peaks
df[name + "_#GSR"] = [len(x) for x in peaks]
# Mean + STD amplitude
df[name + "_mean_amplitudes"] = [np.mean(x) if len(x)>0 else 0 for x in amplitudes]
df[name + "_std_amplitudes"] = [np.std(x) if len(x)>0 else 0 for x in amplitudes]
```

```
        # Mean + STD rise_times
        df[name + "_mean_rise_times"] = [np.mean(x) if len(x)>0 else 0 for x in rise_times]
        df[name + "_std_rise_times"] = [np.std(x) if len(x)>0 else 0 for x in rise_times]
        # Mean + STD half_recovery
        df[name + "_mean_half_recovery"] = [np.mean(x) if len(x)>0 else 0 for x in half_recovery]
        df[name + "_std_half_recovery"] = [np.std(x) if len(x)>0 else 0 for x in half_recovery]
        # Mean + STD recovery
        df[name + "_mean_recovery"] = [np.mean(x) if len(x)>0 else 0 for x in recovery]
        df[name + "_std_recovery"] = [np.std(x) if len(x)>0 else 0 for x in recovery]
        # Mean + STD onsets
        df[name + "_mean_onsets"] = [np.mean(x) if len(x)>0 else 0 for x in onsets]
        df[name + "_std_onsets"] = [np.std(x) if len(x)>0 else 0 for x in onsets]
        # Mean + STD offsets
        df[name + "_mean_offsets"] = [np.mean(x) if len(x)>0 else 0 for x in offsets]
        df[name + "_std_offsets"] = [np.std(x) if len(x)>0 else 0 for x in offsets]

        #----------------
        # Features from: Data Descriptor: Emotional ratings and skin conductance response to visual, auditory and haptic stimuli
        # sum of amplitude
        df[name + "_sum_amp"] = [np.sum(x) if len(x)>0 else 0 for x in amplitudes]
        # first amp
        df[name + "_first_amp"] = [x[0] if len(x)>0 else 0 for x in amplitudes]
        #phasic max
        df[name + "_phasic_max"] = [np.max(x) for x in phasic]
        #----------------

        #----------------
        # Features on normalized data
```

```
        normalized_x = normalize(x)
        df[name + "_norm_mean"] = [np.mean(x) for x in normalized_x]
        df[name + "_norm_std"] = [np.std(x) for x in normalized_x]
        df[name + "_norm_var"] = [np.var(x) for x in normalized_x]
        #----------------

        # Features for review
        def get_cvxEDA(y, delta, tau0=2., tau1=0.7, delta_knot=10., alpha=8e-4, gamma=1e-2,
                       solver=None, options={'reltol':1e-9, 'show_progress': False, 'gp':
dict(msg_lev='GLP_MSG_OFF')}):
            """CVXEDA Convex optimization approach to electrodermal activity processing
            This function implements the cvxEDA algorithm described in "cvxEDA: a
            Convex Optimization Approach to Electrodermal Activity Processing"
            (http://dx.doi.org/10.1109/TBME.2015.2474131, also available from the
            authors' homepages).
            Arguments:
            y: observed EDA signal (we recommend normalizing it: y = zscore(y))
            delta: sampling interval (in seconds) of y
            tau0: slow time constant of the Bateman function
            tau1: fast time constant of the Bateman function
            delta_knot: time between knots of the tonic spline function
            alpha: penalization for the sparse SMNA driver
            gamma: penalization for the tonic spline coefficients
            solver: sparse QP solver to be used, see cvxopt.solvers.qp
            options: solver options, see:
                                    http://cvxopt.org/userguide/coneprog.html#algorithm-
parameters
            Returns (see paper for details):
            r: phasic component
```

```
p: sparse SMNA driver of phasic component

t: tonic component

l: coefficients of tonic spline

d: offset and slope of the linear drift term

e: model residuals

obj: value of objective function being minimized (eq 15 of paper)

"""

import cvxopt as cv


n = len(y)

y = cv.matrix(y)


# bateman ARMA model

a1 = 1./min(tau1, tau0) # a1 > a0

a0 = 1./max(tau1, tau0)

ar = np.array([(a1*delta + 2.) * (a0*delta + 2.), 2.*a1*a0*delta**2 - 8.,

        (a1*delta - 2.) * (a0*delta - 2.)]) / ((a1 - a0) * delta**2)

ma = np.array([1., 2., 1.])


# matrices for ARMA model

i = np.arange(2, n)

A = cv.spmatrix(np.tile(ar, (n-2,1)), np.c_[i,i,i], np.c_[i,i-1,i-2], (n,n))

M = cv.spmatrix(np.tile(ma, (n-2,1)), np.c_[i,i,i], np.c_[i,i-1,i-2], (n,n))


# spline

delta_knot_s = int(round(delta_knot / delta))

spl = np.r_[np.arange(1.,delta_knot_s), np.arange(delta_knot_s, 0., -1.)] # order 1

spl = np.convolve(spl, spl, 'full')

spl /= max(spl)
```

```
        # matrix of spline regressors
        i = np.c_[np.arange(-(len(spl)//2), (len(spl)+1)//2)] + np.r_[np.arange(0, n, delta_knot_s)]
        nB = i.shape[1]
        j = np.tile(np.arange(nB), (len(spl),1))
        p = np.tile(spl, (nB,1)).T
        valid = (i >= 0) & (i < n)
        B = cv.spmatrix(p[valid], i[valid], j[valid])

        # trend
        C = cv.matrix(np.c_[np.ones(n), np.arange(1., n+1.)/n])
        nC = C.size[1]

        # Solve the problem:
        # .5*(M*q + B*l + C*d - y)^2 + alpha*sum(A,1)*p + .5*gamma*l'*l
        # s.t. A*q >= 0

        old_options = cv.solvers.options.copy()
        cv.solvers.options.clear()
        cv.solvers.options.update(options)

        if solver == 'conelp':
                # Use conelp
                z = lambda m,n: cv.spmatrix([],[],[],(m,n))
                G = cv.sparse([[-A,z(2,n),M,z(nB+2,n)],[z(n+2,nC),C,z(nB+2,nC)],
                                        [z(n,1),-1,1,z(n+nB+2,1)],[z(2*n+2,1),-1,1,z(nB,1)],
                                        [z(n+2,nB),B,z(2,nB),cv.spmatrix(1.0, range(nB),
range(nB))]])
                h = cv.matrix([z(n,1),.5,.5,y,.5,.5,z(nB,1)])
                c = cv.matrix([(cv.matrix(alpha, (1,n)) * A).T,z(nC,1),1,gamma,z(nB,1)])
```

```
                res = cv.solvers.conelp(c, G, h, dims={'l':n,'q':[n+2,nB+2],'s':[]})
                obj = res['primal objective']
            else:
                # Use qp
                Mt, Ct, Bt = M.T, C.T, B.T
                H = cv.sparse([[Mt*M, Ct*M, Bt*M], [Mt*C, Ct*C, Bt*C],
                               [Mt*B, Ct*B, Bt*B+gamma*cv.spmatrix(1.0, range(nB),
range(nB))]])
                f = cv.matrix([(cv.matrix(alpha, (1,n)) * A).T - Mt*y,  -(Ct*y), -(Bt*y)])
                res = cv.solvers.qp(H, f, cv.spmatrix(-A.V, A.I, A.J, (n,len(f))),
                                    cv.matrix(0., (n,1)), solver=solver)
                obj = res['primal objective'] + .5 * (y.T * y)
            cv.solvers.options.clear()
            cv.solvers.options.update(old_options)

            l = res['x'][-nB:]
            d = res['x'][n:n+nC]
            t = B*l + C*d
            q = res['x'][:n]
            p = A * q
            r = M * q
            e = y - r - t

            return (np.array(a).ravel() for a in (r, p, t, l, d, e, obj))

        def get_dPhEDA(x):
            length = len(x)
            new = np.zeros(length)
            for i, value in enumerate(x):
```

```
                    if (i - 2) < 0 or (i + 2) > (length - 1):
                        new[i]= value
                        continue

                    new[i] = (x[i-2] - 8*x[i-1] + 8*x[i+1] - x[i+2]) / (12 * (1/2))

                return new

        # --- Preprocess

        # resample to 4 Hz
        sec = x.shape[1]/sampling_rate
        resampled_first = resample_axis(x, input_fs= sampling_rate, output_fs= 4)

        # median filter
        median_filtered = [ndimage.median_filter(i, size= 4) for i in resampled_first]

        # resample to 2 Hz
        resampled_second = resample_axis(median_filtered, input_fs= 4, output_fs= 2)

        # highpass filter #
        highpass_filtered = [butter_filter(data= i, cutoff_freq= 0.01, fs= sampling_rate, btype=
"highpass", order= 8) for i in resampled_second]

        # --- dPhEDA route
        # cvxEDA
        print("Extract cvxEDA EDA...")
        cvxEDA = [get_cvxEDA(i, 1./2) for i in tqdm(highpass_filtered)]
        # get the pasic part
```

```
        phasic = [list(i)[0] for i in cvxEDA]

        # five point stencil

        dPhEDA = [get_dPhEDA(i) for i in phasic]

        if plot:

                plt.plot(resample_axis(x[0], sample_size= 250, axis= 0), label = "raw")

                plt.plot(resample_axis(resampled_first[0], sample_size= 250, axis= 0), label =
"resampled_first")

                plt.plot(resample_axis(median_filtered[0], sample_size= 250, axis= 0), label =
"median_filtered")

                plt.plot(resample_axis(resampled_second[0], sample_size= 250, axis= 0), label =
"resampled_second")

                plt.plot(resample_axis(highpass_filtered[0], sample_size= 250, axis= 0), label =
"highpass_filtered")

                plt.plot(resample_axis(phasic[0], sample_size= 250, axis= 0), label = "phasic")

                plt.plot(resample_axis(dPhEDA[0], sample_size= 250, axis= 0), label = "dPhEDA")

                plt.legend()

                plt.show()


        # add dPhEDA to the feature list

        dPhEDA = np.array(dPhEDA)

        for i in range(len(dPhEDA[0])):

                df[name + "_dPhEDA_{}".format(i)] = dPhEDA[:, i]


        # --- Time-Varying Index of Sympathetic Activity (TVSymp) and Modified TVSymp (MTVSymp)

        # (time-frequency spectral analysis)


        def get_MTVSymp(x, k_sec = 5, fs = 2):

                MTVSymp = np.zeros(len(x))

                for i in range(len(x)):

                        start = max(i - (fs * k_sec), 0)
```

```
                end = max(i - 1, 0)

                if end == 0:
                        MTVSymp[i]= x[i]
                        continue

                ut = np.mean(x[start:end])

                if ut > x[i]:
                        MTVSymp[i]= 0
                else:
                        MTVSymp[i]= x[i] - ut
        return MTVSymp

# - VFCDM: variable frequency complex demodulation

# parameters
data_length = x[0].shape[0]
center_frequencies = [0.04, 0.12, 0.2, 0.28, 0.36, 0.44, 0.52, 0.6, 0.68, 0.76, 0.84, 0.92]

bandwidth = 0.04 # Fω
length = data_length//2# Nω is chosen to be approximately half the data length [16].

components = []
for center in center_frequencies[1:3]:
        # finite-impulse response (FIR)
        fir = signal.firwin(numtaps = length, cutoff= center, width= bandwidth)

        #adaptive lowpass filter (LPF)
```

```
            component = [signal.lfilter(fir, 1, i) for i in highpass_filtered]

            components.append(component)


        # We summed the second and third components to include the sympathetic dynamics,
        # which range between 0.045–0.25 Hz, followed by normalization to unit variance.
        # The summed value is denoted by X'.

        # sum of the second and third components
        sum = np.sum(np.array(components), axis= 0)
        # followed by normalization to unit variance
        normed = [(i - i.mean(axis=0)) / i.std(axis=0) if i.std(axis=0) != 0 else i for i in sum]

        # Hilbert transform
        analytic_signal = [hilbert(i) for i in normed]

        #TVSymp, a(t), is obtained by calculating the instantaneous amplitude of Z(t) ->
amplitude_envelope
        TVSymp = [np.abs(i) for i in analytic_signal]

        # --- Calculation of MTVSympt
        MTVSymp = [get_MTVSymp(i) for i in TVSymp]

        # add TVSymp to the feature list
        TVSymp = np.array(TVSymp)
        for i in range(len(TVSymp[0])):
            df[name + "_TVSymp_{}".format(i)] = TVSymp[:, i]
```

```
    # defragment the DataFrame
    df = df.copy()

    # add MTVSymp to the feature list
    MTVSymp = np.array(MTVSymp)
    for i in range(len(MTVSymp[0])):
        df[name + "_MTVSymp_{}".format(i)] = MTVSymp[:, i]

    if plot:
        plt.plot(resample_axis(x[0], sample_size= 250, axis= 0), label = "raw")
        plt.plot(resample_axis(normed[0], sample_size= 250, axis= 0), label = "normed")
        plt.plot(resample_axis(TVSymp[0], sample_size= 250, axis= 0), label = "TVSymp")
        plt.plot(resample_axis(MTVSymp[0], sample_size= 250, axis= 0), label = "MTVSymp")
        plt.legend()
        plt.show()

    return df


def analyze_EDA(x, sampling_rate, plot= False, onset_threshold = 0.01, offset_threshold = 0,
peak_amplification_threshold = 0.04, method = "highpass"):
    """
    Function to analyze a given EDA signal.
    Filters the given signal with a lowpass filter and decomposes its into tonic + phasic component.
    Furthermore, GSR are localized and returned.
    Decomposition information: "https://imotions.com/blog/skin-conductance-response/"
    + "https://www.biopac.com/knowledge-base/phasic-eda-issue/"

    Parameters
```

---

x: np. Eda signal to analyze.

show_plot: Bool. Whether to plot a graph with results or not. Default= False.

onset_threshold = 0.01, offset_threshold = 0, peak_amplification_threshold = 0.05 -> Default parameters to detect GSRs.

method: String. Method to decompose tonic/phasic signal. Can be either "highpass" or "median". Default: "highpass".

Returns

Lists: phasic, tonic, peaks, onsets, offsets, rise_times, amplitudes, recovery, half_recovery.

"""

```
df = pd.DataFrame()
df["Filtered"]= x

# phasic/tonic decomposition
if method == "highpass":
        import scipy
        sos = scipy.signal.butter(2, 0.05, btype="lowpass", output="sos", fs=sampling_rate)
        df["Tonic"] = scipy.signal.sosfiltfilt(sos, x)
        sos = scipy.signal.butter(2, 0.05, btype="highpass", output="sos", fs=sampling_rate)
        df["Phasic"] = scipy.signal.sosfiltfilt(sos, x)

elif method == "median":
```

```
        # Calculate Tonic
        df["Tonic"] = moving_average(df["Filtered"].values, 4 * sampling_rate)
        # Calculate Phasic
        df["Phasic"] = df["Filtered"].values - df["Tonic"].values
else:
        print("Method '{}' not known to decompose EDA.".fotrmat(method))
        raise NotImplementedError

# --- GSR detection
possible_peaks = []
# find possible peaks: onset > 0.01 || offset < 0
phasic = df["Phasic"].values
i = 0
while i < len(phasic):
        if phasic[i] > onset_threshold:
                for j in range(i + 1, len(phasic)):
                        if phasic[j] < offset_threshold:
                                possible_peaks.append([i, j])
                                i = j + 1
                                break
        i += 1

# features to be saved
peaks_x = [] # x-values of peak
onsets_x = [] # x-values of onset
offsets_x = [] # x-values of offset
rise_times = [] # Number of samples between onset and peak
amplitudes = [] # Y difference between peak and offsets
recovery_times = [] # Number of samples between peak and offset
```

```
    half_recovery_times = [] # Number of samples between peak and half recovery


    # remove from possible peaks where peak is not exceeding 0.05
    for p in possible_peaks:
        data = df["Filtered"].values[p[0]:p[1]]
        rise = np.max(data) - data[0]
        if rise >= peak_amplification_threshold:
            index = int(np.argmax(data) + p[0])
            peaks_x.append(index)
            onsets_x.append(p[0])
            offsets_x.append(p[1])
            rise_times.append(index - p[0])
            amplitudes.append(rise)
            recovery_times.append(p[1] - index)
            peak_offset_difference = abs(x[index] - x[p[1]])
            half_recovery_y = (peak_offset_difference/2) + min(x[index : p[1]])
            half_recovery_times.append(np.argmin(abs(x[index:p[1]] - half_recovery_y)))


    assert len(peaks_x) == len(onsets_x) == len(offsets_x) == len(rise_times) == len(amplitudes) == len(recovery_times) == len(half_recovery_times)


    if plot:
        fig = plt.figure()


        linewidth= 3
        fontsize= 18
        marker_size= 130


        # --- First plot
```

```
        ax = fig.add_subplot(311)
        ax.plot(df["Filtered"].values, color="royalblue", label= "Filtered", linewidth=linewidth)
        ax.plot(df["Tonic"].values, color="lightgreen", label= "Tonic", linewidth=linewidth)
        ax.legend(loc='upper left', fontsize= fontsize)
        ax.tick_params(left=False,
                       bottom=False,
                       labelleft=False,
                       labelbottom=False)


        # --- Second plot
        ax = fig.add_subplot(312)
        ax.plot(df["Phasic"].values, color="orange", label= "Phasic", linewidth=linewidth)
        ax.axhline(0, color="gainsboro", linestyle= "dashed", linewidth=linewidth)
        ax.axhline(onset_threshold, color="mediumseagreen", linestyle= "dashed", label=
"Onset threshold", linewidth=linewidth)
        possible_onsets = [x[0] for x in possible_peaks]
        possible_offsets = [x[1] for x in possible_peaks]
        ax.scatter(possible_onsets, [df["Phasic"][x] for x in possible_onsets], s= marker_size,
marker='>', c="dimgrey", zorder=10, label="Possible Onset", edgecolors='b')
        ax.scatter(possible_offsets, [df["Phasic"][x] for x in possible_offsets], s= marker_size,
marker='<', c="darkgrey", zorder=10, label="Possible Offset", edgecolors='b')
        ax.legend(loc='upper left', fontsize= fontsize)
        ax.tick_params(left=False,
                       bottom=False,
                       labelleft=False,
                       labelbottom=False)


        # --- Third plot
        ax = fig.add_subplot(313)
```

```
        half_recovs_x = [i + j for i, j in zip(half_recovery_times, peaks_x)]


        def get_y(x):
            return df["Filtered"][x]


        ax.plot(df["Filtered"].values, color= "#E91E63", label= "Raw", linewidth=linewidth)
        for peak_x, onset_x, half_recovery_x in zip(peaks_x, onsets_x, half_recovs_x):
            peak_y, onset_y, half_recovery_y = get_y(peak_x), get_y(onset_x),
get_y(half_recovery_x)


            # color peak "royalblue"
            # plot risetime
            plt.plot([peak_x, onset_x], [onset_y, onset_y], c="#FFA726", linestyle="dashed",
linewidth=linewidth)
            # plot amplitude
            plt.plot([peak_x, peak_x], [onset_y, peak_y], c="#1976D2", linestyle="solid",
linewidth=linewidth)
            # plot half recovery
            plt.plot([half_recovery_x, peak_x], [half_recovery_y, half_recovery_y],
c="#FDD835", linestyle="dashed", linewidth=linewidth)


        ax.scatter(peaks_x, [df["Filtered"][x] for x in peaks_x], s= marker_size, marker='v',
c="orange", zorder=10, label="Peaks", edgecolors='b')
        ax.scatter(onsets_x, [df["Filtered"][x] for x in onsets_x], s= marker_size, marker='>',
c="dimgrey", zorder=10, label="Onset", edgecolors='b')
        ax.scatter(offsets_x, [df["Filtered"][x] for x in offsets_x], s= marker_size, marker='<',
c="darkgrey", zorder=10, label="Offset", edgecolors='b')
        ax.scatter(half_recovs_x, [df["Filtered"][x] for x in half_recovs_x], s= marker_size,
marker='o', c="skyblue", zorder=10, label="Half recovery", edgecolors='b')
        ax.legend(loc= 'upper left', fontsize= fontsize)
        ax.tick_params(left=False,
                       bottom=False,
```

```
                                    labelleft=False,
                                    labelbottom=False)

    return df["Phasic"].values, df["Tonic"].values, peaks_x, onsets_x, offsets_x, rise_times, amplitudes, recovery_times, half_recovery_times


def ecg_feature_extraction(x, sampling_rate, name= "", plot= False):
    '''
    Function to calculate features for an ECG signal.
    Different features are calculated for the given signal 'x'.

    Parameters
    ----------
    x: np. Sensor data.
    sampling_rate: Int. Sampling rate of the given sensor data.
    name: String. String to put a name in front of computed features. Default is ''.
    plot: Bool. Boolean that describes whether to plot the outcome of the analysis or not. Default is False.

    Returns
    ----------
    result: df. Pandas Dataframe describing the computed features.
    '''
    df =  pd.DataFrame()

    # preprocess
    #x = np.apply_along_axis(preprocess_ecg, axis=1, arr= x, plot= False, sampling_rate= sampling_rate)

    # Plot one example
```

```
if plot:
        example = x[0]
        # Plot preprocessing
        preprocess_ecg(example, sampling_rate= sampling_rate, plot= True)
        plt.title("Ecg preprocessing")
        plt.show()

        # Plot the QRS complex points
        calc_hr_peaks(example, sampling_rate= sampling_rate, plot= True)
        plt.plot(example)
        plt.legend()
        plt.title("ECG analysis")
        plt.show()

        # Plot Biosppy analysis
        from biosppy.signals import ecg
        out = ecg.ecg(signal=example, sampling_rate=250, show=True)
        plt.show()

# Retrieve peaks - Format List with "DataSamples" entries of type Array(RPeak, XY-Axis)
print("Calculate HR peaks...")
features = [calc_hr_peaks(i, sampling_rate= sampling_rate, plot= False) for i in tqdm(x)]
# Unpack results
p_list = [i[0] for i in features]
q_list = [i[1] for i in features]
peaks = [i[2] for i in features]
s_list = [i[3] for i in features]
t_list = [i[4] for i in features]
```

```
# Retrieve RR interval - Difference of x component ("[:, 0]") of the peaks
rr = [np.diff(i[:, 0]) if len(i) > 0 else [] for i in peaks]

# remove empty slides
rr = [x if len(x)>0 else [0] for x in rr]

# mean RR
df[name + "_meanRR"] = [np.mean(x) for x in rr]
# RMSSD: take the square root of the mean square of the differences
df[name + "_RMSSD"] = [np.sqrt(np.mean(np.square(np.diff(x)))) if len(x)>1 else 0 for x in rr]
# mean of the std of the RR-intervals
df[name + "_stdRR"] = [np.std(x) if len(x)>1 else 0 for x in rr]
# slope of the linear regression of RR
df[name + "_slope"] = [best_fit_slope(xs=np.arange(len(x)), ys=x) for x in rr]
# Own: number of found R-peaks
df[name + "_#_r_peaks"] = [len(x) for x in peaks]

# --- additional features
# max
df[name + "_max"] = np.max(x, axis=1)
# min
df[name + "_min"] = np.min(x, axis=1)
# range
df[name + "_range"] = (np.max(x, axis=1)-np.min(x, axis=1))
# std
df[name + "_std"] = (np.std(x, axis=1))
# iqr
df[name + "_iqr"] = (iqr(x, axis=1))
# mean
```

```
        df[name + "_mean"] = np.mean(x, axis=1)

        # rms

        df[name + "_rms"] = (np.sqrt(np.mean(x**2, axis=1)))

        # local maxima

        df[name + "_local_max"] = np.apply_along_axis(lambda x: np.mean(argrelextrema(x,
np.greater)), arr=x, axis=1)

        # local minima

        df[name + "_local_min"] = np.apply_along_axis(lambda x: np.mean(argrelextrema(x, np.less)),
arr=x, axis=1)

        # mean absolut value

        df[name + "_mean_abs"] = (np.mean(np.abs(x), axis= 1))


        # --- PQRST features

        # Electrocardiogram Feature Extraction and Pattern Recognition Using a Novel Windowing
Algorithm

        names = [name + "_pr_interval", name + "_qt_interval", name + "_pt_interval", name +
"_qrs_duration"]

        zips = [[p_list, q_list],  [q_list, t_list], [p_list, t_list], [q_list, s_list]]

        for name, to_zip in zip(names, zips):

                df[name + "_mean"] = [np.mean(j[:,0] - i[:,0])  if len(i)>0 else 0 for i, j in zip(*to_zip)]

                df[name + "_std"] = [np.std(j[:,0] - i[:,0])  if len(i)>0 else 0 for i, j in zip(*to_zip)]

                df[name + "_range"] = [max(j[:,0] - i[:,0]) - min(j[:,0] - i[:,0]) if len(i)>0 else 0 for i, j in
zip(*to_zip)]

                df[name + "_diff"] = [(j[-1,0]-i[-1,0]) - (j[0,0]-i[0,0]) if len(i)>0 else 0 for i, j in zip(*to_zip)]


        return df


def emg_feature_extraction(x, sampling_rate, name= "", plot= False):

        '''

        Function to calculate features for an EMG signal.
```

```
    Different features are calculated for the given signal 'x'.

    Parameters
    ----------
    x: np. Sensor data.
    sampling_rate: Int. Sampling rate of the given sensor data.
    name: String. String to put a name in front of computed features. Default is ''.
    plot: Bool. Boolean that describes whether to plot the outcome of the analysis or not. Default is False.

    Returns
    ----------
    result: df. Pandas Dataframe describing the computed features.
    '''
    df =  pd.DataFrame()

    # Plot one example
    if plot:
        example = x[randrange(x.shape[0])]
        example = preprocess_emg(example, sampling_rate= sampling_rate, plot= True)
        analyze_emg(example, sampling_rate= sampling_rate, plot= True)
        plt.legend()
        plt.title("EMG analysis")
        plt.show()

    # preprocess
    #x = np.apply_along_axis(preprocess_emg, axis=1, arr= x, plot= False, sampling_rate= sampling_rate)
```

```
        # max
        df[name + "_max"] = np.max(x, axis=1)
        # min
        df[name + "_min"] = np.min(x, axis=1)
        # range
        df[name + "_range"] = (np.max(x, axis=1)-np.min(x, axis=1))
        # std
        df[name + "_std"] = (np.std(x, axis=1))
        # iqr
        df[name + "_iqr"] = (iqr(x, axis=1))
        # mean
        df[name + "_mean"] = np.mean(x, axis=1)
        # rms
        df[name + "_rms"] = (np.sqrt(np.mean(x**2, axis=1)))
        # mean absolut value
        df[name + "_mean_abs"] = (np.mean(np.abs(x), axis= 1))
        # mean absolut values of the first differences
        df[name + "_mean_abs_1_diff"] = np.mean(np.abs(x[:, 1:] - x[:, :-1]), axis= 1)
        # mean absolut values of the second differences
        df[name + "_mean_abs_2_diff"] = np.mean(np.abs(x[:, 2:] - x[:, :-2]), axis= 1)
        # zscore (relative to the sample mean and standard deviation) as standardization method
        standardized_emg = zscore(x)
        # mean absolut values of the first differences -standardized - z score
        df[name + "_mean_abs_1_diff_std"] = np.mean(np.abs(standardized_emg[:, 1:] -
standardized_emg[:, :-1]), axis= 1)
        # mean absolut values of the second differences -standardized - z score
        df[name + "_mean_abs_2_diff_std"] = np.mean(np.abs(standardized_emg[:, 2:] -
standardized_emg[:, :-2]), axis= 1)
        # Variation of the first and second moment of the signal over time
```

```
        df[name + "_var_mom"] = np.apply_along_axis(lambda i: np.var([moment(i, 1), moment(i, 2)]), arr=x, axis=1)


        # Note: Empirical Mode Decomposition and time windows of activity are not implemented by now.

        # Note: Could be implemented: Degree of stationarity in the spectrum domain


        # -- Additional EMG features

        # signals power spectrum

        print("Calc power spectrum for EMG...")

        spec = [power_spectrum(i, sampling_rate= sampling_rate) for i in tqdm(x)]

        # mode frequency of the signals power spectrum

        df[name + "_freq_ps"] = [mode(i, keepdims=False)[0] for i in spec]

        # mean frequency of the signals power spectrum

        df[name + "_mean_freq"] = [np.mean(i) for i in spec]

        # zero crossing of the time-domain signal

        df[name + "_zero_crossing"] = np.apply_along_axis(lambda i: ((i[:-1] * i[1:]) < 0).sum(), arr=x, axis=1)


        # --- Further power sprectrum analysis

        # Source: https://www.intechopen.com/books/computational-intelligence-in-electromyography-analysis-a-perspective-on-current-applications-and-future-challenges/the-usefulness-of-mean-and-median-frequencies-in-electromyography-analysis

        # Power sprectrum using periodogram

        print("Calc periodogram for EMG...")

        PSD = [signal.periodogram(i) for i in tqdm(x)]

        #PSD= f: Array of sample frequencies.; Pxx: Power spectral density or power spectrum of x.


        if plot:

                example = PSD[randrange(x.shape[0])]
```

```
            plt.plot(example[0], example[1])
            plt.title("Periodogram: PSD estimator on EMG signal")
            plt.show()


        # MNF is an average frequency
        df[name + "_MNF"] = [sum(i[0]*i[1])/sum(i[1]) if sum(i[1]) != 0 else 0 for i in PSD]
        # MDF - frequency at which the EMG power spectrum is divided into two regions with equal amplitude or TTP (dividing the total power area into two equal parts)
        def get_median_index(x):
            i = 1
            total_sum = sum(x)
            while (sum(x[:i]) < total_sum / 2):
                i += 1
            return i
        df[name + "_MDF"] = [get_median_index(i[1]) for i in PSD]
        # TTP - aggregate of EMG power spectrum
        df[name + "_TTP"] = [sum(i[1]) for i in PSD]
        # MNP - average power of EMG power spectrum
        df[name + "_MNP"] = [np.mean(i[1]) for i in PSD]
        # PKF is a frequency at which the maximum EMG power occurs
        df[name + "_PKF"] = [i[0][np.argmax(i[1])] for i in PSD]
        # SM1
        df[name + "_SM1"] = [sum(i[1] * i[0]) for i in PSD]
        # SM2
        df[name + "_SM2"] = [sum(i[1] * (i[0] ** 2)) for i in PSD]
        # SM3
        df[name + "_SM3"] = [sum(i[1] * (i[0] ** 3)) for i in PSD]
        # --- FR - FR is used to discriminate between relaxation and contraction of the muscle
        # cutoff frequency between low- and high-frequencies
```

```
        cutoffs = [np.argmin(abs(f - mnf)) for f, mnf in  zip([i[0] for i in PSD], df[name + "_MNF"].values)]

        df[name + "_FR"] = [sum(P[:c]) / sum(P[c:]) if sum(P[c:]) != 0 else 0 for P, c in zip([i[1] for i in PSD], cutoffs)]

        # PSR - extension versionof PKF and FR features - ratio between the energyP0which is nearby the maximum va-lue of the EMG power spectrum and the energy P which is thewhole energy of the EMG power spectrum

        def psr(x, n= 20):

                f0 = np.argmax(x)

                start= max(f0-n, 0)

                end= min(len(x), f0+n)

                return sum(x[start:end]) - sum(x)

        df[name + "_PKF"] = [psr(i[1]) for i in PSD]

        # VCF - Variance of central frequency

        df[name + "_VCF"] = [ (SM2/SM0) - (SM1/SM0) **2 if SM0 != 0 else 0 for SM0, SM1, SM2 in zip(df[name + "_TTP"].values, df[name + "_SM1"].values, df[name + "_SM2"].values)]


        # --- Selfmade

        # PKF is a frequency at which the maximum EMG power occurs

        df[name + "_max_power_spec"] = [np.max(i[1]) for i in PSD]


        # RMS envelope analysis

        print("Analyze EMG...")

        RMS = [analyze_emg(i, sampling_rate= sampling_rate) for i in tqdm(x)]

        df[name + "_mean_RMS"] = [np.mean(i) for i in RMS]

        df[name + "_max_RMS"] = [np.max(i) for i in RMS]

        df[name + "_std_RMS"] = [np.std(i) for i in RMS]


        return df


def analyze_emg(x, sampling_rate, plot= False):
```

```
    '''
    Function to analyze an EMG signal.
    Source: https://www.mdpi.com/1424-8220/20/17/4892/htm

    Parameters
    ----------
    x: np. Sensor data.
    sampling_rate: Int. Sampling rate of the given sensor data.
    plot: Bool. Boolean that describes whether to plot the outcome of the analysis or not.

    Returns
    ----------
    result: DF. Filtered envelope of the EMG signal.
    '''

    # RMS is calculated over 100 ms windows
    RMS = RMS_envelope(x, w= int(sampling_rate//10))
    # 1st order Butterworth lowpass filter with cut-off frequency at 1 Hz is applied to the RMS
    filtered = butter_filter(RMS, cutoff_freq= 2, fs= sampling_rate, order= 1, btype= "lowpass")

    if plot:
        plt.plot(RMS, color="mediumseagreen", label= "RMS")
        plt.plot(filtered, color="orange", label= "Final filter")

    return filtered

def generic_features(x, name= ""):
    '''
    Function to calculate generich features for any signal.
```

Different features are calculated for the given signal 'x'.

Parameters

---

x: np. Sensor data.

name: String. String to put a name in front of computed features. Default is ''.

Returns

---

result: df. Pandas Dataframe describing the computed features.

'''

```
df =  pd.DataFrame()

# max
df[name + "_" + "max"] = np.max(x, axis=1)
# min
df[name + "_" + "min"] = np.min(x, axis=1)
# range
df[name + "_" + "range"] = (np.max(x, axis=1)-np.min(x, axis=1))
# std
df[name + "_" + "std"] = (np.std(x, axis=1))
# iqr
df[name + "_" + "iqr"] = (iqr(x, axis=1))
# mean
df[name + "_" + "mean"] = np.mean(x, axis=1)
# diff mean first/second half
half = len(x)//2
df[name + "_" + "diff_1.2.half_mean"] = np.mean(x[:,:half], axis=1)-np.mean(x[:, half:], axis=1)
```

```
    return df


def respiration_feature_extraction(x, sampling_rate, name= "", plot= False):
    '''
    Function to calculate features for a respiration signal.
    Different features are calculated for the given signal 'x'.

    Parameters
    ----------
    x: np. Sensor data.
    sampling_rate: Int. Sampling rate of the given sensor data.
    name: String. String to put a name in front of computed features. Default is ''.
    plot: Bool. Boolean that describes whether to plot the outcome of the analysis or not. Default is
False.

    Returns
    ----------
    result: df. Pandas Dataframe describing the computed features.
    '''

    df =  pd.DataFrame()

    # Plot one example
    if plot:
        example = x[randrange(x.shape[0])]
        example = preprocess_resp(example, sampling_rate= sampling_rate, plot= plot)
        plt.title("Respiration preprocessing")
        plt.legend()
```

```
            plt.show()


            analyze_resp(example, plot= plot)
            plt.legend()
            plt.title("Respiration analysis")
            plt.show()


        # preprocess
        #x = np.apply_along_axis(preprocess_resp, axis=1, arr= x, plot= False, sampling_rate= sampling_rate)


        # --- Generic statistical approaches
        # max
        df[name + "_max"] = np.max(x, axis=1)
        # min
        df[name + "_min"] = np.min(x, axis=1)
        # range
        df[name + "_range"] = (np.max(x, axis=1)-np.min(x, axis=1))
        # std
        df[name + "_std"] = (np.std(x, axis=1))
        # iqr
        df[name + "_iqr"] = (iqr(x, axis=1))
        # mean
        df[name + "_mean"] = np.mean(x, axis=1)


        # --- Paper: Respiratory Feature Extraction for Radar-Based Continuous Identity Authentication
        # breathing rate - FFT (standard: 12 to 20 beats per minute (0.2-0.33 Hz)
        # Analyse the respiration signal
        print("Analyze respitation...")
```

```
        features = [analyze_resp(i) for i in tqdm(x)]
        # Unpack results
        peaks_x = [i[0] for i in features]
        peaks_y = [i[1] for i in features]
        lows_x = [i[2] for i in features]
        lows_y = [i[3] for i in features]
        inhale_areas = [i[4] for i in features]
        exhale_areas = [i[5] for i in features]

        # Number of peaks
        df[name + "_#_peaks"] = [len(x) for x in peaks_x]
        # Number of peaks
        df[name + "_#_lows"] = [len(x) for x in lows_x]
        # Mean Amplitude
        df[name + "_mean_ampl_x"] = [np.mean([abs(x-y) for x,y in zip(i, j)]) for i, j in zip(peaks_x,
lows_x)]
        df[name + "_mean_ampl_y"] = [np.mean([abs(x-y) for x,y in zip(i, j)]) for i, j in zip(peaks_y,
lows_y)]
        # Std Amplitude
        df[name + "_mean_ampl_x"] = [np.std([abs(x-y) for x,y in zip(i, j)]) for i, j in zip(peaks_x, lows_x)]
        df[name + "_mean_ampl_y"] = [np.std([abs(x-y) for x,y in zip(i, j)]) for i, j in zip(peaks_y, lows_y)]
        # Mean volume of in/exhalation
        df[name + "_mean_in"] = [np.mean(i) if len(i)>0 else 0 for i in inhale_areas]
        df[name + "_mean_ex"] = [np.mean(i) if len(i)>0 else 0 for i in exhale_areas]
        # STD volume of in/exhalation
        df[name + "_std_in"] = [np.std(i) if len(i)>0 else 0 for i in inhale_areas]
        df[name + "_std_ex"] = [np.std(i) if len(i)>0 else 0 for i in exhale_areas]

        return df
```

```
def analyze_resp(x, plot= False):
    '''
    Function to analyze a respiration signal.
    Trapezius detection from paper: "Respiratory Feature Extraction for Radar-Based Continuous Identity Authentication"

    Parameters
    ----------
    x: np. Sensor data.
    plot: Bool. Boolean that describes whether to plot the outcome of the analysis or not.

    Returns
    ----------
    result: peaks, df["Filtered"][peaks], lows, df["Filtered"][lows], inhale_area, exhale_area
    '''

    df = pd.DataFrame()
    df["Filtered"]= x

    # --- Peak detection
    # Difference of two consecutive y-values
    diff = np.diff(df["Filtered"])
    # Sign difference of "diff"
    sdiff = np.diff(np.sign(diff))
    # Rising edge
    zero_crossings_rising = (sdiff == 2)
    # Falling edge
    zero_crossings_falling = (sdiff == -2)
```

```
        # X values of the lows
        lows = np.where(zero_crossings_rising)[0]
        # X values of the peaks
        peaks = np.where(zero_crossings_falling)[0]
        # All x values
        indices_both = np.where(zero_crossings_rising | zero_crossings_falling)[0]
        assert len(lows) + len(peaks) == len(indices_both)

        # --- Breath ratio -> trapezium detection
        upper_trap_points = []
        lower_trap_points = []
        for first, second in zip(indices_both[:-1], indices_both[1:]):
                amplitude = abs(df["Filtered"][first] - df["Filtered"][second])
                lowest_point = min(df["Filtered"][first], df["Filtered"][second])
                lower_border = amplitude * 0.3 + lowest_point
                higher_border = amplitude * 0.7 + lowest_point

                # Find closest point to "higher_border"
                diff_upper = np.abs(df["Filtered"].values[first:second] - higher_border)
                upper_trap_points.append(first + np.argmin(diff_upper))
                # Find closest point to "lower_border"
                diff_lower = np.abs(df["Filtered"].values[first:second] - lower_border)
                lower_trap_points.append(first + np.argmin(diff_lower))

        # --- Form trapezius
        inhale_x = []
        exhale_x = []
        for a, b, c, d in zip(upper_trap_points[:-1], lower_trap_points[:-1], upper_trap_points[1:],
lower_trap_points[1:]):
```

```
            # chech inhalte or exhale
            if a > b:
                inhale_x.append((a,b,d,c))
            else:
                exhale_x.append((a,b,d,c))


    if plot:
            import matplotlib.patches as patches
            ax = plt.gca()


            ax.plot(df["Filtered"].values, color="royalblue", label= "Post filter")
            ax.scatter(peaks, [df["Filtered"][x] for x in peaks], marker='o', c="orange", zorder=10, label="Positive Peaks")
            ax.scatter(lows, [df["Filtered"][x] for x in lows], marker='o', c="pink", zorder=10, label="Negative Peaks")
            ax.scatter(upper_trap_points, [df["Filtered"][x] for x in upper_trap_points], marker='o', color="green", zorder=10, label= "70% Border")
            ax.scatter(lower_trap_points, [df["Filtered"][x] for x in lower_trap_points], marker='o', color="red", zorder=10, label= "30% Border")
            for x, y in zip(inhale_x, [[df["Filtered"][x] for x in b] for b in inhale_x]):
                ax.add_patch(patches.Polygon(xy=list(zip(x, y)), fill=True, label= "Inhale", alpha=0.3))
            for x, y in zip(exhale_x, [[df["Filtered"][x] for x in b] for b in exhale_x]):
                ax.add_patch(patches.Polygon(xy=list(zip(x, y)), fill=True, label= "Exhale", alpha=0.3, color="orange"))
            ax.plot(lower_trap_points, [df["Filtered"][x] for x in lower_trap_points], color="red", linestyle='dashed', alpha=0.5)
            ax.plot(upper_trap_points, [df["Filtered"][x] for x in upper_trap_points], color="green", linestyle='dashed', alpha=0.5)


    # --- Calculate the are of in/exhale
    inhale_area, exhale_area = [], []
```

```
        for x in exhale_x:
                y = [df["Filtered"][i] for i in x]
                exhale_area.append(0.5*np.abs(np.dot(x,np.roll(y,1))-np.dot(y,np.roll(x,1))))
        for x in inhale_x:
                y = [df["Filtered"][i] for i in x]
                inhale_area.append(0.5*np.abs(np.dot(x,np.roll(y,1))-np.dot(y,np.roll(x,1))))

        return peaks, df["Filtered"][peaks], lows, df["Filtered"][lows], inhale_area, exhale_area


def feature_extraction(X, sensor_list, sampling_rate):
        '''
        Function to calculate hand-crafted-features for a given dataframe 'X'.
        Dependen on the given 'sensor_list' different features are computed for each sensor channel.

        Parameters
        ----------
        X: np. Data to extract features from. Expect dataframe to be in shape [samples, time, sensors,
0].
        sensor_list: list. List of names of the sensors in 'X'. Should be in the same order as sensors in 'X'.
        sampling_rate: int. Integer defining the sampling rate of the given data.

        Returns
        ----------
        result: DF. Pandas Dataframe in shape (samples, features).
        '''

        result = pd.DataFrame()
        for i, sensor in enumerate(sensor_list):
                print("\n\n_____")
```

```
                print(f"Start extracting features for sensor '{sensor}'...")

                if sensor in["time"]:

                        continue

                if sensor in ["gsr", "Eda_RB", "Eda_E4"]:

                        result = pd.concat([result, gsr_feature_extraction(X[:, :, i, 0], name= sensor,
sampling_rate= sampling_rate)], axis=1, sort=False)

                        continue

                if sensor == "Resp":

                        result = pd.concat([result, respiration_feature_extraction(X[:, :, i, 0], name=
sensor, sampling_rate= sampling_rate)], axis=1, sort=False)

                        continue

                if sensor in ["Ecg", "ecg"]:

                        result = pd.concat([result, ecg_feature_extraction(X[:, :, i, 0], name= sensor,
sampling_rate= sampling_rate)], axis=1, sort=False)

                        continue

                if sensor in ["emg_trapezius", "Emg"]:

                        result = pd.concat([result, emg_feature_extraction(X[:, :, i, 0], name= sensor,
sampling_rate= sampling_rate)], axis=1, sort=False)

                        continue

                result = pd.concat([result, generic_features(X[:, :, i, 0], name= sensor)], axis=1,
sort=False)
        result = result.fillna(0)


        return result


def get_hcf(dataset):
        '''
        Function to return hand-crafted-features for a given dataset.


        Parameters
```

```
    ----------
    dataset: String. String describing the used dataset. Should be either "painmonit" or "biovid".

    Returns
    ----------
    features: DF. Pandas Dataframe in shape (samples, features).
    '''

    np_dir = Path("datasets", dataset, "hcf")
    feature_file = Path(np_dir, "features.csv")

    if not feature_file.exists():
        raise FileExistsError(f"File '{feature_file.resolve()} does not exists. Did you create hand-crafted features (run `hcf.py')?")

    return pd.read_csv(feature_file, sep= ";", decimal= ",")

def create_hcf(X, sensor_list, dataset, sampling_rate, overwrite= False):
    '''
    Features are just recomputed when there is no existing "features.csv" file containing already computed metrics.

    Parameters
    ----------
    X: np. Data to extract features from. Expect dataframe to be in shape [samples, time, sensors, 0].
    sensor_list: list. List of names of the sensors in 'X'. Should be in the same order as sensors in 'X'.
    dataset: String. String describing the used dataset. Should be either "painmonit" or "biovid".
    sampling_rate: int. Integer defining the sampling rate of the given data.
    '''
```

```
    np_dir = Path("datasets", dataset, "hcf")
    feature_file = str(Path(np_dir, "features.csv"))

    if not overwrite and Path(feature_file).exists():
        print("HCF have been created before and will not be overwritten.")
        return

    # create directory
    if not Path(np_dir).exists():
        os.makedirs(np_dir)

    # create numpy hcf
    features = feature_extraction(X, sensor_list= sensor_list, sampling_rate= sampling_rate)

    if features.isnull().values.any():
        print("Created features contain NaN. Try to remove columns containing NaN...")

        columns_with_nan = features.columns[features.isna().any()].tolist()
        print("Columns with NaN: {}.".format(columns_with_nan))

        features = features.drop(columns_with_nan, axis=1)

    # save the hcf
    features.to_csv(feature_file, sep= ";", decimal= ",", index=False)
    print(f"Features have been saved under '{feature_file}'.")

if __name__ == "__main__":
    """Main function.
```

```
    """

    from config import painmonit_sensors, biovid_sensors, sampling_rate_painmonit, sampling_rate_biovid

    from scripts.data_handling import read_biovid_np, read_painmonit_np

    print("Create HCF for Biovid...")

    X_biovid, y_biovid, subjects_biovid = read_biovid_np()

    create_hcf(X_biovid, sensor_list= biovid_sensors, dataset= "biovid", sampling_rate= sampling_rate_biovid, overwrite= False)

    print("HCF for Biovid created.")


    print("Create HCF for UzL...")

    X_uzl, y_uzl, subjects_uzl = read_painmonit_np(label= "heater")

    create_hcf(X_uzl, sensor_list= painmonit_sensors, dataset= "painmonit", sampling_rate= sampling_rate_painmonit, overwrite= False)

    print("HCF for UzL created.")
```