Příloha 4

Kódové provedení batch_tools

copanda – hlavní detekční algoritmus STUDNA, který shromažďuje zbytek kódy dohromady
batch_preprocessing – kód zodpovědný za zpracování dat a trénování prediktivního modelu
batch_suspint – kód zodpovědný za získání příznaků learning entropy
batch_suspfreq – kód zodpovědný za získání příznaků FFT
batch_suspstat – kód zodpovědný za získání statistických příznaků
batch_susprough – kód zodpovědný za získání příznaků drsnosti

---

```
from STUDNA.mtools import suspicious_intervals, suspicious_frequencies
from STUDNA.ptools import clear_noise_rough, learning_entropy_2
from STUDNA.ptools import return_roughness, calculate_Ra, calculate_Rq,
calculate_Rdq, calculate_Skew, calculate_Kurt
from STUDNA import mlearning
import numpy as np
import warnings
import pickle


def copada(bench_time, bench_input, bench_output, actual_time, actual_input,
actual_output, settings=None, **kwargs):
    """
    Complete pack of anomaly detection and analysis. Uses all batch_tools unique
    functions to
    create a single pack of important anomaly detection parameters. Does not tell
    or predict
    if there is an anomaly in signal! Only returns calculated values needed for
    such prediction.

    Parameters
    -------------
    bench_time : array-lie
        Time domain of benchmark singal.

    bench_input : array-lie
        Benchmark inputs.

    bench_output : array-lie
        Benchmark outputs.

    actual_time : array-lie
        Time domain of compared signal.

    actual_input : array-lie
        Compared inputs.

    actual_output : array-lie
        Compared outputs.

    settings : dict
        Dictionary that contains all neeeded precalculations
        to speed up the algorithm


    Keyword Arguments
    ----------------------
    BATCH PREPROCESSING
    epochs : int
        Number of epochs for which is neuron trained.

    update_epochs : int
        Number of epochs for which is neuron update. Lesser epochs = higher
```

```
        sensitivity.

    error_limit : float
        If error limit is reached in the pretraining phase,
        then algorithm shuts down with ValueError before upcoming
        learning. Correct wanted error limit saves time and ensures
        that pretraining of the data was done well.

    lr : float
        Learning rate.

    biased : bool
        Adds bias to the learning process.

    learning_technique : str
        In batch preprocessing the data are subjected to the machine learning
        process. Multiple
        techniques can be used for learning process. Input the shortcut of the
        technique.
            Available techniques:
                LNU : linear neuron unit

    prelearning_process : str
        Preprocess input into predicor.
        'delayed_inputs' - Uses function delayed_inputs()
        'delayed_inout' - Uses funciton delayed_inout() to create delayed outputs
and inputs
        None - Uses function untouched_inputs() that does nothing

    delay : int
        Number of delay steps used in prelearning_process.


    BATCH SUSPINT
    bw_mag : float
        Bandwidth parameter estimator. Helps to estimate parameter that
        directly corresponds with suspicion sensitivity. Tests show that
        values from 0 to 1 works the best.

        The lower the value the higher is sensitivity, but less accurate.
        The higher the value the lower is sensitivity, but more accurate.

        Warning: Lower (i.e., negative) and higher values than 0 and 1 are
        allowed, but their performance is drastically worse for 1 and 2 weights
        systems.

    bw_par_calc : str
        Way of calculating sensitivity parameters. Provides different approaches
        to adjust sensitivity value:
        'complete'
            Is less sensitive to small errors, when in signal appears significant
            error, but provides possibility to spot error in the original signal.

            Calculates bandwidth = std*bw_mag + mean from whole learning entropy.

        'partial'
            Higher sensitivity for all the errors. Significant errors cannot cover
            less significant errors. But it is worse at detecting errors in
            healthy signal.

            Calculates bandwidth = std*bw_mag + mean from original ('healthy')
learning entropy.

    FILTER
    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

    FFT
    nfrq : int
        Max number of new recognized frequencies.

    EXTRA
    return_extra : bool
        Returns extra information about the signal.


    Returns
    ---------
    returns : dict
        All needed values for anomaly detection.
        See returns.keys()

    settings : dict
        Dictionary of all constant calculations (benchmark)
        that will speed up calculations for repetitive jobs.

    """
    # Initialization
    returns = dict()

    extra = False if isinstance(kwargs.get('return_extra'), type(None)) else
bool(kwargs.get('return_extra'))

    preprocess_pars = [
        100 if isinstance(kwargs.get('epochs'), type(None)) else
int(kwargs.get('epochs')),
        3 if isinstance(kwargs.get('update_epochs'), type(None)) else
int(kwargs.get('update_epochs')),
        1e-4 if isinstance(kwargs.get('lr'), type(None)) else
float(kwargs.get('lr')),
        0.1 if isinstance(kwargs.get('error_limit'), type(None)) else
float(kwargs.get('error_limit')),
        True if isinstance(kwargs.get('biased'), type(None)) else
bool(kwargs.get('biased')),
        None if isinstance(kwargs.get('learning_technique'), type(None)) else
str(kwargs.get('learning_technique')),
        None if isinstance(kwargs.get('prelearning_process'), type(None)) else
str(kwargs.get('prelearning_process')),
        1 if isinstance(kwargs.get('delay'), type(None)) else
int(kwargs.get('delay')),
        0.05 if isinstance(kwargs.get('buff_par'), type(None)) else
float(kwargs.get('buff_par')),
        0.1 if isinstance(kwargs.get('pca_par'), type(None)) else
float(kwargs.get('pca_par'))
    ]

    suspint_pars = [
        1.47 if isinstance(kwargs.get('bw_mag'), type(None)) else
float(kwargs.get('bw_mag')),
        'complete' if isinstance(kwargs.get('bw_par_calc'), type(None)) else
str(kwargs.get('bw_par_calc')),
        4 if isinstance(kwargs.get('cl_par'), type(None)) else
int(kwargs.get('cl_par'))
    ]

    suspfreq_pars = [
        2 if isinstance(kwargs.get('nfrq'), type(None)) else
int(kwargs.get('nfrq'))
    ]
```

```
    # Some steps to fasten the time - load settings
    if isinstance(settings, type(None)):
        settings = dict()
        settings['batch_preprocessing'] = None
        settings['batch_suspstat'] = None
        settings['batch_susprough'] = None

    else:
        settings = settings.copy()

    # Some uneccecary step because batch_processing works that way
    out_packed = np.array((bench_output, actual_output))
    in_packed = np.array((bench_input, actual_input))
    t_packed = np.array((bench_time, actual_time))


    # Prepare pre-training
    processed_pack, settings['batch_preprocessing'] =
batch_preprocessing(in_packed, out_packed, t_packed,

epochs=preprocess_pars[0],

update_epochs=preprocess_pars[1],

lr=preprocess_pars[2],

error_limit=preprocess_pars[3],

biased=preprocess_pars[4],

preloaded_pack=settings['batch_preprocessing'],

learning_technique=preprocess_pars[5],

prelearning_process=preprocess_pars[6],

delay=preprocess_pars[7],

buff_par=preprocess_pars[8],

pca_par=preprocess_pars[9])


    # Calculate suspicious intervals
    suspint_pack = batch_suspint(processed_pack, bw_mag=suspint_pars[0],
                                 bw_par_calc=suspint_pars[1],
                                 cl_par=suspint_pars[2], provide_details=True)

    returns['bench_peaks'] = suspint_pack['partial_num_of_peaks'][0]
    returns['actual_peaks'] = suspint_pack['partial_num_of_peaks'][1]
    returns['bench_peak_power'] = suspint_pack['partial_power_of_peaks'][0]
    returns['actual_peak_power'] = suspint_pack['partial_power_of_peaks'][1]
    returns['bench_entropy'] = suspint_pack['partial_peaks_entropy'][0]
    returns['actual_entropy'] = suspint_pack['partial_peaks_entropy'][1]

    # Calculate basic stats
    suspstat_pack, settings['batch_suspstat'] = batch_suspstat(processed_pack,
settings['batch_suspstat'])

    for key in suspstat_pack:
        returns[key] = suspstat_pack[key]

    # Calculate roughness coefficients
    susprough_pack, settings['batch_susprough'] = batch_susprough(processed_pack,
settings['batch_susprough'])

    for key in susprough_pack:
        returns[key] = susprough_pack[key]
```

```
    # Calculate new frequencies
    suspfreq_pack = batch_suspfreq(processed_pack, suspfreq_pars[0])

    for key in suspfreq_pack:
        returns[key] = suspfreq_pack[key]

    if extra:
        returns['nfrq'] = suspfreq_pars[0]
        returns['nw'] = processed_pack.get('weights').shape[1]
        returns['noise_std'] = np.std(processed_pack.get('noise'))
        returns['weight_convergence'] = processed_pack.get('weight_convergence')

    return returns, settings


def batch_preprocessing(inputs_matrix, outputs_matrix, time_matrix,
learning_technique: str = "LNU",
                        prelearning_process: str = None, preloaded_pack=None,
delay: int = 1, **kwargs):
    """
    Preprocess data so they are ready for batch algorithms.
    Combines multiple signal samples, teaches them with selected machine learning
    technique and joins the learning entropy into single signal and single time
domain.
    Thees outputs can be used for every batch-tool in this library.

    Parameters
    ---------------
    inputs_matrix : array-like
        Array n x m, where 'n' is length of the inputted signals (number of data-
points)
        and 'm' is number of samples.

    outputs_matrix : array-like
        Array n x m, where 'n' is length of the outputted signals (number of data-
points)
        and 'm' is number of samples.

    time_matrix : array-like
        Array nxm, where n is length of the time (number of data)
        and m is number of time domains for each sample. Each i-th time domain
corresponds
        to the i-th sample. For i = 0, ..., m

    learning_technique : str
        In batch preprocessing the data are subjected to the machine learning
process. Multiple
        techniques can be used for learning process. Input the shortcut of the
technique.
            Available techniques:
                LNU : linear neuron unit

    prelearning_process : str
        Preprocess input into predicor.
        'delayed_inputs' - Uses function delayed_inputs()
        'delayed_inout' - Uses funciton delayed_inout() to create delayed outputs
and inputs
        None - Uses function untouched_inputs() that does nothing

    delay : int
        Number of delay steps used in prelearning_process.

    preloaded_pack : bytes
        In order to save time while pretraining, predictor can be loaded into the
system.
```

```
    Keyword Arguments
    -----------------
    Parameters for the machine learning adjustment.
    lr : float
        Learning rate.

    epochs : int
        Number of epochs for which is neuron trained.

    update_epochs : int
        Number of epochs for which is neuron update. Lesser epochs = higher
sensitivity.

    biased : bool
        Adds bias to the learning process.

    error_limit : float
        If error limit is reached in the pretraining phase, then algorithm shuts
down with ValueError before upcoming
        learning. Correct wanted error limit saves time and ensures that
pretraining of the data was done well.

    Parameters for noise filtering adjustment.
     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


    Returns
    -------------------
    returns : dict
        Dictionary that contains all the important data for further batch
processes.
        Keys:
            noise : Noise in the outputted signal
            learning_effort : Main data array that is used to calculate learning
entropy.
                              Corresponds to the absolute value of dw summed up
during every epoch of learning
            weights : Weights of learning process during last learning epoch.
            delta_weights : Weight updates during last learning epoch.
            inputs : Flatten inputs. Warning: Might be shorten due to learning
technique.
            outputs : Flatten outputs. Warning: Might be shorten due to learning
technique.
            time : Flatten time domain. Warning: Might be shorten due to learning
technique.
            separator : Tuple of initial signal length and number of signals

    preloaded_pack : bytes
        Pretrained predictor.
        Note: 'None' if preloaded_pack was inserted.

    """

    # Learning parameters
    returns = dict()
    learning_params = dict()
    learning_params['lr'] = 5e-5 if isinstance(kwargs.get('lr'), type(None)) else
float(kwargs.get('lr'))
    learning_params['epochs'] = 100 if isinstance(kwargs.get('epochs'), type(None))
else int(kwargs.get('epochs'))
    learning_params['biased'] = True if isinstance(kwargs.get('biased'),
```

```
type(None)) else bool(kwargs.get('biased'))
    learning_params['error_limit'] = 0.01 if isinstance(kwargs.get('error_limit'),
type(None)) else float(kwargs.get('error_limit'))
    learning_params['update_epochs'] = 2 if isinstance(kwargs.get('update_epochs'),
type(None)) else int(kwargs.get('update_epochs'))
    learning_technique = 'LNU' if isinstance(learning_technique, type(None)) else
str(learning_technique)

    buff_par = 0.05 if isinstance(kwargs.get('buff_par'), type(None)) else
float(kwargs.get('buff_par'))
    pca_par = 0.1 if isinstance(kwargs.get('pca_par'), type(None)) else
float(kwargs.get('pca_par'))

    # Check if axis is correct
    if time_matrix.shape[1] < time_matrix.shape[2]:
        warnings.warn('Warning. Axis 1 of time domain is smaller than axis 2. Make
sure, that axis 0 is sample axis, asix 1 is time axis and axis 2 is number of
signals.')
    elif (time_matrix.shape[1] != outputs_matrix.shape[1]) or
(time_matrix.shape[1] != inputs_matrix.shape[1]):
        raise TypeError(f'Inputted matrices must have same axis 1 shape.')

    # Choose learning preprocess function
    if prelearning_process == 'delayed_inputs':
        learning_preprocess_function = mlearning.delayed_inputs
    elif prelearning_process == 'delayed_inout':
        learning_preprocess_function = mlearning.delayed_inout
    else:
        learning_preprocess_function = mlearning.untouched_inputs

    # Load learning model or create new one
    if not isinstance(preloaded_pack, type(None)):
        predictor = pickle.loads(preloaded_pack)

    else:
        # Choose learning technique
        if learning_technique == 'LNU':
            predictor = mlearning.LNU(lr=learning_params.get('lr'),
epochs=learning_params.get('epochs'),
                                      biased=learning_params.get('biased'))

        else:
            raise SyntaxError(f'Unknown technique {learning_technique}')

        # Learn every signal by chosen technique
        lr_inputs, lr_outputs = learning_preprocess_function(inputs_matrix[0],
outputs_matrix[0], min_delay=delay)
        predictor.fit(lr_inputs, lr_outputs)
        returns['weight_convergence'] = predictor.w_by_epoch()

    # Check if error isn't too big
    # Method of estimation - SOFT - mean abs value of error must not be larger that
error_limit (percent)
    error_from_mean =
abs(predictor.err_by_window())/np.mean(abs(outputs_matrix[0]))
    error_from_max = abs(predictor.err_by_window())/np.max(abs(outputs_matrix[0]))
    error = np.mean((error_from_mean, error_from_max))
    if error > learning_params.get('error_limit'):
        raise ValueError(f'Prediction error is higher than setup limit. Change
learning parameters. '
                         f'\nError: {error}'
                         f'\nLimit: {learning_params.get("error_limit")}')

    # Signal noise detection
    filter_input = outputs_matrix[0]

    returns['noise'] = clear_noise_rough(filter_input, pca_par=pca_par,
                                         buff_par=buff_par) - filter_input
```

```
    # Generate weight updates without learning, for detection purposes. = Generate
output
    first = True
    for x, y in zip(inputs_matrix, outputs_matrix):
        spec_x, spec_y = learning_preprocess_function(x, y, min_delay=delay)
        predictor.fake_update(spec_x, spec_y, learning_params.get('update_epochs'))

        if first:
            returns['learning_effort'] = predictor.learning_effort().copy()
            returns['weights'] = predictor.w_by_window().copy()
            returns['delta_weights'] = predictor.dw_by_window().copy()
            first = False
        else:
            returns['learning_effort'] =
np.concatenate((returns.get('learning_effort'),
predictor.learning_effort().copy()))
            returns['weights'] = np.concatenate((returns.get('weights'),
predictor.w_by_window().copy()))
            returns['delta_weights'] =
np.concatenate((returns.get('delta_weights'), predictor.dw_by_window().copy()))

    # Join signals and time-domains from time matrix to create continuous time
domain
    # Outputs are sliced, because of possible shortening of data in learning method
    reducted_len = len(predictor.learning_effort())

    returns['inputs'] = inputs_matrix[:, -
reducted_len:].reshape(reducted_len*inputs_matrix.shape[0], inputs_matrix.shape[2])
    returns['outputs'] = outputs_matrix[:, -
reducted_len:].reshape(reducted_len*outputs_matrix.shape[0],
outputs_matrix.shape[2])
    returns['time'] = time_matrix[:, -
reducted_len:].reshape(reducted_len*time_matrix.shape[0], time_matrix.shape[2])
    returns['separator'] = tuple([time_matrix.shape[1], time_matrix.shape[0]])

    if not isinstance(preloaded_pack, type(None)):
        preloaded_pack = None
    else:
        preloaded_pack = pickle.dumps(predictor)

    # Return learning entropy, learning weights, time domain
    return returns, preloaded_pack


def batch_suspint(batch_pack, bw_mag: float = 1, bw_par_calc: str = 'mean-noise',
**kwargs):
    """
    Algorithm based on suspicious_intervals, but adapted for maintenance
prediction.
    Takes learning effort of two or more signals joins them,
    processes with suspicious_intervals() and provides enhanced description.

    Parameters
    -------------------
    batch_pack : batch_preprocessing dict
        Dictionary that was generated by batch_preprocessing function.
        Must contain noise, learning_effort, time, separator

    bw_mag : float
        Bandwidth parameter estimator. Helps to estimate parameter that
        directly corresponds with suspicion sensitivity. Tests show that
        values from 0 to 1 works the best.

        The lower the value the higher is sensitivity, but less accurate.
        The higher the value the lower is sensitivity, but more accurate.

        Warning: Lower (i.e., negative) and higher values than 0 and 1 are
```

```
        allowed, but their performance is drastically worse for 1 and 2 weights
        systems.

    bw_par_calc : str
        Way of calculating sensitivity parameters. Provides different approaches
        to adjust sensitivity value:
        'complete'
            Is less sensitive to small errors, when in signal appears significant
            error, but provides possibility to spot error in the original signal.

            Calculates bandwidth = std*bw_mag + mean from whole learning entropy.

        'partial'
            Higher sensitivity for all the errors. Significant errors cannot cover
            less significant errors. But it is worse at detecting errors in
            healthy signal.

            Calculates bandwidth = std*bw_mag + mean from original ('healthy')
learning entropy.

    Keyword Arguments
    -------------------
    cl_par : float
        Cluster parameter in time_domain units. Determines what should algorithm
        recognize as one cluster.

        Example: There are 4 suspicious places in signal with time difference
        (.3, .5, 1, 2) seconds. If cl_par = 1 (s), then three suspicious places
        with differences (.3,.5,1) are recognized as one cluster.

    provide_details : bool
        If True additional key will be added into outputted dictionary -
'tech_details'.
        Provides additional dictionary that contains: learning_entropy, suspects,
suspect_intervals.

        For visualisation purposes.

        Usage example:
        magnifier = numpy.max(batch_pack.get('delta_weights'))
        plt.plot(batch_pack.get('delta_weights'), alpha=0.5)
        plt.plot(returned.get('tech_details').get('suspects')*magnifier,
color='red')


    Returns
    -------------------
    returns : dict
        Dictionary with main statistics is returned.
        Keywords:
            'num_of_peaks' - Total number of suspected values
            'partial_num_of_peaks' - Number of suspected values in each sample
packed into list
            'power_of_peaks' - Mean of learning entropy in suspected intervals
            'partial_power_of_peaks' - Power_of_peaks for each sample packed into
list
            'intervals' - List of suspicious intervals in tuple (start, end)
            'duration' - Total duration of unusual intervals

    """

    cl_par = 1 if isinstance(kwargs.get('cl_par'), type(None)) else
float(kwargs.get('cl_par'))
    provide_details = False if isinstance(kwargs.get('provide_details'),
type(None)) else bool(kwargs.get('provide_details'))

    # Special bw_parameter calculation, that provides more reliability
    if bw_par_calc == 'complete':
```

```
        bw_par = None
    elif bw_par_calc == 'partial':
        e = learning_entropy_2(batch_pack.get('learning_effort'), order=2)
        e_clean = clear_noise_rough(e)
        start = batch_pack.get('separator')[0] *
range(batch_pack.get('separator')[1])[0]
        end = batch_pack.get('separator')[0] *
(range(batch_pack.get('separator')[1])[-1])
        e_mean_unwanted = np.mean(e_clean[start:end]**2)
        e_std_unwanted = np.std(e_clean[start:end]**2)
        bw_par = e_mean_unwanted + e_std_unwanted*bw_mag
    else:
        raise ValueError(f'Unknown method: {bw_par_calc}')

    # suspects - crit_pos - 0, time - 1, crit_pos_b - 2, suspect_b - 3, G - 4,
pos_diff - 5, E_clean - 6, E - 7
    suspects = suspicious_intervals(batch_pack.get('learning_effort'),
batch_pack.get('time'),
                                    bw_mag=bw_mag,
                                    bw_par=bw_par,
                                    noise_std=np.std(batch_pack.get('noise')),
                                    cl_par=cl_par, adjust_mode=True,
normalize=True)

    # Generate additional information about signal (partial number of peaks,
partial power of peaks)
    partials = {'num_of_peaks': list(), 'power_of_peaks': list(),
                'std_entropy': list()}
    for k in range(batch_pack.get('separator')[1]):
        start = batch_pack.get('separator')[0] * k
        end = batch_pack.get('separator')[0] * (k + 1)

        # Partial number of peaks
        partials['num_of_peaks'].append(sum(suspects[4][start:end])[0])

        # Partial power of peaks
        partials['power_of_peaks'].append(
            np.mean(suspects[6][start:end][suspects[4][start:end] == 1]))

        # Standard deviation of the entropy
        partials['std_entropy'].append(np.std(suspects[6][start:end]))

    returns = {
        'num_of_peaks': suspects[0].get('sum'),
        'partial_num_of_peaks': partials['num_of_peaks'],
        'power_of_peaks': suspects[0].get('mean')/np.std(batch_pack.get('noise')),
        'partial_power_of_peaks': partials['power_of_peaks'],
        'partial_peaks_entropy': partials['std_entropy'],
        'intervals': suspects[0].get('intervals'),
        'duration': suspects[0].get('duration')
    }

    # If user wants additional information for visualisation. Add that to the
returns.
    if provide_details:
        # If we recognized some suspects
        if sum(suspects[4]) > 0:
            # Create signal that shows beginning and end of the interval
            suspect_intervals = np.zeros((len(suspects[1]), 1))
            for suspected in suspects[1].reshape(-1, 1)[suspects[4] ==
1][suspects[2]]:
                suspect_intervals[suspects[1] == suspected] = 1

            # Magnify the intervals
            suspect_intervals *= max(batch_pack.get('delta_weights')[:, 0])
        else:
            suspect_intervals = np.zeros(len(suspects[1]))
```

```
        tech_details = {
            'learning_entropy': suspects[6]**2,
            'suspects': suspects[4],
            'suspect_intervals': suspect_intervals,

        }

        returns['tech_details'] = tech_details
    return returns


def batch_suspfreq(batch_pack, spaces: int):
    """
    Algorithm based on suspicious_frequencies, but adopted for maintanance
prediction.
    Searches for unusual frequencies in every weight.

    Parameters
    -------------
    batch_pack : batch_preprocessing dict
        Dictionary that was generated by batch_preprocessing function.
        Must contain noise, learning_effort, time, separator

    spaces : int
        In order to create table of constant size, there is a parameter
        spaces. Returns fixed number of frequencies.

    Returns
    ------------
    returns : dict
        Dictionary with psd and frq for every weight and fixed amount of
        frequencies is returned.
        Keywords:
            'wX_psd_n' - n-th PSD of X-th weight
            'wX_frq_n' - n-th frequency of X-th weight

    """
    returns = dict()
    helper = dict()

    # Calculate some constants
    sep = batch_pack.get('separator')[0]
    timestep = batch_pack.get('time')[1] - batch_pack.get('time')[0]

    # Repeat process for every weight
    for nw in range(batch_pack.get('delta_weights').shape[1]):
        w_name = 'w' + str(nw)

        # Split signals to healthy (bench) and aged (compared)
        bench = batch_pack.get('delta_weights')[0:sep, nw]
        compared = batch_pack.get('delta_weights')[sep:sep * 2, nw]

        suspfreq = suspicious_frequencies(bench, compared, timestep)

        helper[f"{w_name}_psd"] = suspfreq['psd']
        helper[f"{w_name}_frq"] = suspfreq['frq']

    # Transform values
    for key in helper:
        for space in range(spaces):
            if space < len(helper[key]):
                returns[f'{key}_{space}'] = helper[key][space]
            else:
                returns[f'{key}_{space}'] = 0

    return returns
```

```
def batch_suspstat(batch_pack, preloaded_pack=None):
    """
    Algorithm for maintance prediction. Searches for unusualities in the
    statistics of the signal: mean, std, q75, q50, q25.

    Parameters
    ----------------
    batch_pack : batch_preprocessing dict
        Dictionary that was generated by batch_preprocessing function.
        Must contain noise, learning_effort, time, separator

    preloaded_pack : batch_suspstat dict
        Dictionary that contains all needed calculations of bench data
        so that calculations might be skipped.


    Returns
    ----------------
    returns : dict
        Dictionary of statistics for every weight.
        See returns.keys()

    preloaded_pack_return : dict, None
        If preloaded_pack was not loaded, then this will return dictinoary
        of all the needed calculations in order to skip those calculations next
time.

    """
    returns = dict()
    sep = batch_pack.get('separator')[0]

    # Or load it
    if not isinstance(preloaded_pack, type(None)):
        preloaded_pack_return = None

    # Create preloaded pack
    else:
        preloaded_pack = dict()
        stats = dict()
        for nw in range(batch_pack.get('weights').shape[1]):
            w_name = 'w' + str(nw)
            bench = batch_pack.get('weights')[0:sep, nw]
            stats['mean'] = np.mean(bench)
            stats['std'] = np.std(bench)
            stats['75'] = np.quantile(bench, 0.75)
            stats['50'] = np.quantile(bench, 0.50)
            stats['25'] = np.quantile(bench, 0.25)
            preloaded_pack[w_name] = stats.copy()
        preloaded_pack_return = preloaded_pack

    # Calculate the main point
    for nw in range(batch_pack.get('weights').shape[1]):
        w_name = 'w' + str(nw)
        compared = batch_pack.get('weights')[sep:sep * 2, nw]

        returns[f'bench_{w_name}_mean'] = preloaded_pack[w_name]['mean']
        returns[f'actual_{w_name}_mean'] = np.mean(compared)
        returns[f'bench_{w_name}_std'] = preloaded_pack[w_name]['std']
        returns[f'actual_{w_name}_std'] = np.std(compared)
        returns[f'bench_{w_name}_75'] = preloaded_pack[w_name]['75']
        returns[f'actual_{w_name}_75'] = np.quantile(compared, 0.75)
        returns[f'bench_{w_name}_50'] = preloaded_pack[w_name]['50']
        returns[f'actual_{w_name}_50'] = np.quantile(compared, 0.5)
        returns[f'bench_{w_name}_25'] = preloaded_pack[w_name]['25']
        returns[f'actual_{w_name}_25'] = np.quantile(compared, 0.25)

    return returns, preloaded_pack_return
```

```
def batch_susprough(batch_pack, preloaded_pack=None, **kwargs):
    """
    Algorithm for maintanance prediction. Searches for unusualities in
    roughness coefficients Ra, Rq, Rdq, Skewness and Kurtosis.

    Parameters
    ----------------
    batch_pack : batch_preprocessing dict
        Dictionary that was generated by batch_preprocessing function.
        Must contain noise, learning_effort, time, separator

    preloaded_pack : batch_suspstat dict
        Dictionary that contains all needed calculations of bench data
        so that calculations might be skipped.


    Keyword arguments
    --------------------
    filt_order : int
        Order of low pass filter - coresponds to waviness

    crti_freq : float
        Frequency of waviness. 0 to 1


    Returns
    ----------------
    returns : dict
        Dictionary of coefficients for every weight.
        See returns.keys()

    preloaded_pack_return : dict, None
        If preloaded_pack was not loaded, then this will return dictinoary
        of all the needed calculations in order to skip those calculations next
time.

    """

    filt_order = 3 if isinstance(kwargs.get('filt_order'), type(
        None)) else int(kwargs.get('filt_order'))
    crit_freq = 1e-2 if isinstance(kwargs.get('crit_freq'),
                                   type(None)) else float(kwargs.get('crit_freq'))

    sep = batch_pack.get('separator')[0]
    returns = dict()

    # Or load it
    if not isinstance(preloaded_pack, type(None)):
        preloaded_pack_return = None

    # Create preloaded pack
    else:
        preloaded_pack = dict()
        stats = dict()
        for nw in range(batch_pack.get('weights').shape[1]):
            w_name = 'w' + str(nw)

            bench = batch_pack.get('weights')[0:sep, nw]
            bench_rough, bench_wave = return_roughness(
                bench.reshape(-1, 1), filt_order, crit_freq)

            stats['rstd'] = np.std(bench_rough)
            stats['Ra'] = calculate_Ra(bench_rough)
            stats['Rq'] = calculate_Rq(bench_rough)
            stats['Rdq'] = calculate_Rdq(bench_rough)
            stats['Skew'] = calculate_Skew(bench_rough)
            stats['Kurt'] = calculate_Kurt(bench_rough)
```

```
            preloaded_pack[w_name] = stats.copy()
        preloaded_pack_return = preloaded_pack

    for nw in range(batch_pack.get('weights').shape[1]):
        w_name = 'w' + str(nw)

        compared = batch_pack.get('weights')[sep:sep * 2, nw]
        compared_rough, compared_wave = return_roughness(
            compared.reshape(-1, 1), filt_order, crit_freq)

        returns[f'bench_{w_name}_rstd'] = preloaded_pack[w_name]['rstd']
        returns[f'actual_{w_name}_rstd'] = np.std(compared_rough)
        returns[f'bench_{w_name}_Ra'] = preloaded_pack[w_name]['Ra']
        returns[f'actual_{w_name}_Ra'] = calculate_Ra(compared_rough)
        returns[f'bench_{w_name}_Rq'] = preloaded_pack[w_name]['Rq']
        returns[f'actual_{w_name}_Rq'] = calculate_Rq(compared_rough)
        returns[f'bench_{w_name}_Rdq'] = preloaded_pack[w_name]['Rdq']
        returns[f'actual_{w_name}_Rdq'] = calculate_Rdq(compared_rough)
        returns[f'bench_{w_name}_Skew'] = preloaded_pack[w_name]['Skew']
        returns[f'actual_{w_name}_Skew'] = calculate_Skew(compared_rough)
        returns[f'bench_{w_name}_Kurt'] = preloaded_pack[w_name]['Kurt']
        returns[f'actual_{w_name}_Kurt'] = calculate_Kurt(compared_rough)

    return returns, preloaded_pack_return
```