Brewing data analyzer

Information about the dataset

The sensor data, which was recorded during several brewing processes, was equipped with relevant metadata and stored in an HDF5 file. In this file, an HDF5 group is created for each brewing process. Each group of a brewing process in turn contains subgroups to the different vats. The metadata of the vat groups contain data on the physical properties of the vats. The vat groups each contain three data sets, one each for the variables Ti and hi, where i stands for the sensor number, and also one for the time t. The metadata of the variable groups contains information describing the variable in question.

Filter Widths: (9, 29, 48, 191)

Notes on how to proceed with the project

All the necessary files are already included. In your main script main.py import all the packages you need, including the functions module.

The functions module (file functions.py) in the project subfolder already contains all the necessary functions, including transfer parameters and their types. Do not add any other transfer parameters, do not change the variable type or the order. Your job is to implement the functionality and return values of the existing functions. If you write code outside of the given functions, it will only be evaluated if it contributes meaningfully to the functionality of the project. The functions are called in the main.py script to manipulate the data. Implement the main() function in the main.py script.

Make sure there is a clear structure within the scripts. Comment on the code sensibly and equip all functions with Docstrings in Google format. Comments and editions in the programme are to be formulated in English. Make sure you use consistent and precise language. Version your code by first initializing the folder containing the project structure as a git folder, and then making meaningful commits as you edit the task. Think about when a commit is necessary. Add a commit-message to each commit in the usual structure. Again, make sure to use consistent and precise language.

Tasks

Task 1: Read metadata

Place the HDF5 file in the project/data subfolder. After initializing the local git repository, you should add the default (given) files to version control. However, you should not version the data record.

a) First, in the main() function in the script main.py, declare the variables file_path, brewing, and tank_id. The variable file_path contains the path to the HDF5 file. The brewing variable and the

tank_id variable contain the brewing process and tank labels, you can choose whichever one you want.

b) Now declare a tuple named measured_quantities. This tuple contains the names of the HDF5 data records in which the measurement data is located. The exact names can be found in the HDF5 file.

c) Switch to the functions.py file. Now implement the read_metadata() function there. The signature of this function is as follows:

```
1 def read_metadata(file: str, path: str, attr_key: str) -> Any | None:
2    . . .
3
```

The passing parameters of this function are the path to an HDF5 file, the path to an HDF5 group or record within the HDF5 file, and the name of the meta data to be read. This function is intended to read the metadata of a group or a record of any HDF5 file. Catch the KeyError that is thrown when the metadate, HDF5 group, or record does not exist. In this case, have a warning issued in the terminal. This prevents your program from terminating prematurely if you specify an incorrect commit parameter.

d) Switch back to the main.py file. Now call the read_metadata() function in the main() function that you have just implemented to read the metadata T_env, specific_heat_capacity_beer and density_ beer of the brewing process group. This metadata specifies the ambient temperature Tenv during the measurement, the specific heat capacity of the beer c and the density of the beer ρ, respectively.

e) Now read the metadata of the tank group. To do this, call the read_metadata() function as well. The metadata mass_tank, surface_area_tank, footprint_tank, heat_transfer_ coeff_tank, specific_heat_capacity_tank, power_heater and efficiency_heater are relevant. These metadata indicate respectively the mass of the tank m, the outer surface of the tank As, the base area of the tank Ab, the heat transfer coefficient of the tank material k, the specific heat capacity of the tank material c, the electrical power of the heating plate P and the efficiency of the heating plate η.

(Note: The lauter tun (B002) does not have a heating plate. Accordingly, the power in the metadata is P = 0W. If you chose tub B002, functions that take into account the performance of the heating plate must still be implemented! Note: Since the same HDF5 group path is required multiple times when calling the function read_metadata, it makes sense to declare a variable for this path.)

f) In order to be able to archive the evaluated data efficiently, it should be structured sensibly already when it is read out of the HDF5 file. To do this, declare an initially empty dictionary with the name df_data in the main.py function. In task 4 you will fill this dictionary with the evaluated measurement data.

Task 2: Reading data

For the evaluation of the measurement data, it is first important to read out the measurement data assigned to them and check them for plausibility. To store the read data, a dictionary is used as the data structure.

a) In the main() function in the main.py script, declare an empty dictionary named raw_data. Then implement a loop that iterates through the tuple measured_quantities.

b) Switch to the functions.py file. Implement the read_data() function there. The signature of this function is as follows:

```
1 from numpy.typing import NDArray
2 def read_data(file: str, path: str)-> NDArray | None: ...
3    . . .
```

The passing parameters of this function are the path to an HDF5 file and the path to a record within that file. This function is supposed to read a data record from an HDF5 file and save it as numpy.ndarray2 (In the signature, the alias NDArray was used as the type hint). Accordingly, the return value of the function is an array. If the path within the HDF5 file to the record does not exist or is a path to a group, a warning should be issued in the terminal and None should be returned by the function.

c) Implement the check_equal_length() function. The signature of this function is as follows:

```
1 def check_equal_length(*arrays: NDArray)-> bool:
2   . . .
3
```

The pass parameters are any number of arrays that are to be checked for the same dimension. The asterisk at the arrays pass parameter ensures that all passed parameters that cannot be explicitly assigned to a function argument are combined in a tuple. For example, if you pass three arrays to the function, arrays is a tuple with these three arrays. The return value is True if all arrays have the same dimension. Otherwise, it returns False.

d) Switch back to the main.py file. Now call the function read_data() in the loop implemented in part 2a) to read out the measured data. Pass the path to the dataset within the HDF5 file, which you can generate from the labels of the brewing process, the tank, and the current measured variables in the loop. Save this to the raw_data Dictionary, where the key is the current measured variable.

e) Use the check_equal_length() function to check whether the read arrays of the measurement data have the same dimension. If this is not the case, a ValueError should be thrown.

Task 3: Check and process data

Ultrasonic sensors attached to the lid of the tanks are used to measure the fill levels of the tanks. When the liquids in the tanks are heated, steam is produced that settles on the ultrasonic sensors and causes them to fog up. As a result, the measurement of the level is significantly distorted. In the data you have received, this is particularly easy to see where the level measurement becomes negative. In order to enable further processing of the data, these incorrect measuring points must be eliminated. To do this, you will implement several functions below.

a) The recorded time data has been stored in the form of Timestamps with millisecond precision. To convert these to seconds from the start of the measurement, implement the process_time_data() function in the functions.py file. The signature of this function is as follows:

```
1 def process_time_data(data: NDArray)-> NDArray :
2   . . .
```

The pass parameter is an array of Timestamps. The return value is an array of the time data since the beginning of the measurement. Subtract the first value of the passed array from all the other values and convert them in seconds.

b) Implement the remove_negatives() function. The signature of this function is as follows:

```
1 def remove_negatives(array: NDArray)-> NDArray:
2    . . .
```

The pass parameter is an array. The function is supposed to replace all values below 0 with the value np.nan in the passed array. The return value is accordingly the processed array

c) Implement the linear_interpolation() function. The signature of this function is as follows:

```
1 def linear_interpolation(
2 time: NDArray, start_time: float, end_time: float, start_y: float, end_y: float
3 )-> NDArray:
```

The pass parameters are an array of time values to interpolate over, as well as the start and end values of the interpolation. The return value is an array containing the interpolated values that is the same length as the originally passed array. Within this function, a linear interpolation is to be performed. Mathematically, a linear interpolation can be described as follows:

$$y = y_0 + (y_1 - y_0)\,\frac{x - x_0}{x_1 - x_0}$$

d) Implement the interpolate_nan_data() function. The signature of this function is as follows:

```
1 def interpolate_nan_data(time: NDArray, y_data: NDArray)-> NDArray:
2   ...
```

The passing parameters of the function are an array with the time values of the measurement, and an array of measurement data that contains np.nan values that are to be replaced by interpolation. The function should go through the following steps: (Note: Use the copy() function, or the ndarray.copy() class method of the Numpy arrays.)

The steps:

1. First, it is necessary to check whether the first and last values in the array y_data contain a value. If one of these values is np.nan, a ValueError should be thrown.
2. Create a variable active_gap that is initially False. Create a variable interpolated_data that contains a copy of the passed array y_data. (Note: Use the copy() function, or the ndarray.copy() class method of the Numpy arrays.)
3. Now iterate over the entries of the y_data array. As soon as the currently considered value is np.nan for the first time, the index of the currently considered value is stored in the variable start_index. This index marks the beginning of an interval of unknown length in which there are no values (respectively np.nan) in the array. Also, set the variable active_gap to True
4. As soon as a value occurs during the iteration that is not equal to np.nan, the index is stored in the variable end_index
5. Now call the linear_interpolation() function you implemented earlier. The transfer rates are the time data in the interval [start_index:end_index] and the interpolation limits. (Note: Think carefully about which indexes you need to use for the interpolation boundaries.)
6. Reset the active_gap variable to False.
7. After the loop is finished, return the array interpolated_data.

e) Finally, a function is to be developed with the help of which the noisy data can be filtered. The filter_data() function is intended to filter the data using the simple moving average (SMA). The signature of the function is as follows:

```
1 def filter_data(data: NDArray, window_size: int)-> NDArray:
2   ...
```

The pass parameters are an array of data to be filtered, and also the number of values over which the filter should average. This value is also known as the window width. The function has already been partially implemented. Just add to the calculation of the SMA. This is mathematically defined at the

point j as follows, where SMA is the mean (middle value), k is the window width, n = k//2 is the result of an integer division of k and 2, and pi is the value of the matrix at index i:

$$\mathrm{SMA}_j = \frac{1}{k} \sum_{i=j-n}^{j+n+1} p_i$$

f) Switch back to the main.py file. In the main() function, declare an empty dictionary named processed_data and an empty dictionary named df_data. Also, declare a tuple with the name filter_sizes. This tuple consists of four filter sizes provided to you (specifically: the window width of the filter). Now fill in the Dictionary processed_data like this:

1. Convert the Timestamp-data using the process_time_ data function you have implemented. Save the generated array in the df_data Dictionary with the key "time".

2. Declare a loop that iterates over the filter_sizes tuple.

3. Within the loop, call the filter_data() function. Pass the temperature data and the current filter size to the function. Store the resulting array in the processed_data Dictionary, where the key is "temperature_k_FILTER_SIZE". Replace "FILTER_SIZE" with the current filter size.

4. Use the remove_negatives() function to remove the negative values from the fill level data. Then interpolate the resulting gaps using the interpolate_nan_data() function. Finally, filter the now complete data using the filter_data() function. The transfer values are the interpolated data and the current filter size. Save the result in the processed_data Dictionary. Use "level_k_FILTER_SIZE" as the key. Again, replace the word "FILTER_ SIZE" with the current filter size.

Task 4: Analyze data

By this point, you should have implemented the main() function to the point where you can read out the cleaned and filtered measurement data for the brewing process and tank chosen by you. In the next step, you will implement several functions to be able to calculate the internal energy of the tank during the process. The equation required for this is derived from the first law of thermodynamics:

$$\frac{\mathrm{D}K}{\mathrm{D}t} + \frac{\mathrm{D}E}{\mathrm{D}t} = P + \dot{Q}.$$

The first law states that the material change in time D/Dt of the kinetic energy K and the internal energy E of a material body is equal to the work P performed on it per unit of time and the heat energy supplied to it per unit of time Q. The potential energy is part of the work performed by the body. For the process under consideration,the temperature of the container is assumed to correspond to the liquid. The change in the kinetic energy and the amount of energy performed on

the liquid Work is neglected. Apart from the heating capacity Qzu and the convective heat loss Qab, no other outputs are supplied to the system. This results in the simplified formulation:

$$\frac{\mathrm{D}E}{\mathrm{D}t} = \dot{Q}_{zu} - \dot{Q}_{ab}.$$

The total change of the internal energy DE/Dt per unit of time can be divided into the local temporal change of the internal energy ∂E/∂t and the flux of internal energy H due to the inflow or outflow of matter. In this project task, you will only consider sub-processes in which an influx of mass into the tank occurs. Thus, Hab = 0 W, and the left side of the first law is simplified to:

$$\frac{\mathrm{D}E}{\mathrm{D}t} = \frac{\partial E}{\partial t} - \dot{H}_{zu}.$$

The flow of internal energy Hzu is defined according to this next equation, where mzu is the supplied mass flow, c is the specific heat capacity of the beer and Tzu is the temperature of the beer:

$$\dot{H}_{zu} = \dot{m}_{zu} c T_{zu}$$

The mass flow rate depends on the change in the level in the tank h, the density of the beer ρ and the base area of the tank Ab. The following applies:

$$\dot{m} = \varrho \dot{V} = \varrho A_b \dot{h}.$$

By substituting equations 5, 6 and 7 in equation 4 and sequential integration, one obtains

$$E\left(t\right) = \int_0^{t^*} \left(\dot{Q}_{zu} - \dot{Q}_{ab} + \dot{m}_{zu} c T_{zu}\right) \mathrm{d}t + E_0.$$

Your task is to implement equation 8 piece by piece in your program.

a) Implement the calc_heater_heat_flux() function. The signature of the function is as follows:

```
1 def calc_heater_heat_flux(P_heater: float, eta_heater: float)-> float:
2    ...
```

The transfer parameters are the electrical heating power of the heating plate P and its efficiency η. The power supplied by the heating plate is calculated according to the regulation:

$$\dot{Q}_{zu} = P\eta$$

b) Implement the calc_convective_heat_flow() function. The signature of the function is as follows:

```
1 def calc_convective_heat_flow(
2 k_tank: float, area_tank: float, t_total: float, t_env: float
3 )-> float:
4 ...
```

The transfer parameters are the heat transfer coefficient k of the tank, the outer surface area of the tank As, the current temperature of the tank T, and the ambient temperature Tenv. The capacitive heat loss $\dot{}$ Qab is calculated according to the following equation:

$$\dot{Q}_{ab} = kA_s\left(T - T_{env}\right)$$

The next step is to calculate the change in internal energy. This can also be referred to as an enthalpy change. Integration of the change in internal energy according to Equation 6 results in:

$$\int_0^{t^*} \dot{H}_{zu} = \int_0^{t^*} \dot{m}_{zu} c T_{zu} = \dot{m}_{zu} T_{zu} t^* + m_0,$$

where m0 = 0 kg is the amount of beer present in the tank at time t0 = 0s. Since $\dot{}$ mint∗ = m(t∗), the enthalpy at time t = t∗ can be determined from the mass of beer present in the tank at that time. This eliminates the numerical differentiation of the tank level.

c) Implement the calc_mass() function. The signature of this function is as follows:

```
1 def calc_mass(level_data: NDArray, tank_footprint: float, density: float)-> NDArray:
2 ...
```

The transfer parameters are the data on the filling level (Fill height) of the tank h, the base area of the tank Ab and the density of the beer ρ.

d) Implement the function calc_enthalpy(), which calculates the enthalpy of the amount of beer m in the tank at a time t∗. The signature of this function is:

```
1 def calc_enthalpy(
2 mass: float, specific_heat_capacity: float, temperature: float
3 )-> float:
4  . . .
```

The transfer parameters of the function are the mass of the beer at time t∗, the specific heat capacity of the beer and the temperature of the beer. Within the function, implement the following equation:

$$H_{zu}\left(t^{*}\right)=m\left(t^{*}\right)cT_{zu}$$

e) Go to the main.py file. Now use the functions you have just implemented to calculate the internal energy of the tank over time.

1) Within the Filter-width loop, declare an empty list named inner_energy.

2) Implement a loop where you iterate over the time data.

3) Use the implemented functions to calculate the power supplied by the heating plate and the convective heat loss. Also, calculate the enthalpy at the time of the current iteration. Use the calculated values to calculate the internal energy according to Equation 13 and add the result to the list inner_energy.

$$E\left(t^{*}\right)=\dot{Q}_{zu}t^{*}-\dot{Q}_{ab}t^{*}+H_{zu}\left(t^{*}\right)+E_{0}$$

Transfer the metadata read out in task 1 at the appropriate places. To calculate the initial internal energy E0, use the function calc_enthalpy(), where you substitute the specific heat capacity of the beer c with the specific heat capacity of the tank ct instead. The dimensions of the tank can be found in the metadata. The temperature of the tank corresponds to that of the ambient air at the beginning of the process

4) After calculating the internal energy for all time steps, convert the list inner_energy to an array and save it in the df_data Dictionary. The key is "inner_ energy_k_FILTER_SIZE". Replace the word "FILTER_SIZE" with the current filter size.

Task 5: Archive the data

In order to ensure the reproducibility of research results and to facilitate the reuse of research data, the data and the associated metadata must be archived. For this purpose, the previously processed data is stored in an HDF5 file. All relevant metadata must be added to this file so that the file contains all the information for automated recovery and graphical representation of the data.

a) In the main() function, declare the variable h5_path. Assign to this variable the path to the HDF5 file in which you will archive your processed data as a string. Name the HDF5 file data_ GdD_plot_WS2425.h5. Then introduce the variable group_path and assign it the path to the group within the HDF5 file where the data will be stored. Finally declare a dictionary with the name metadata. A minimal example of the metadata that you need to use alongside others is shown below. In the minimal example, replace the values of the dictionary metadata with meaningful values.

```
1 metadata = {
2 "legend_title": "your_title",
3 "x_label": "your_x_label",
4 "x_unit": "your_x_unit",
5 "y_label": "your_y_label",
6 "y_unit": "your_y_unit",
7 }
8  . . .
```

(Note: Make sure that there are no conflicts in the naming of the variables.)

b) Implement the store_plot_data() function. The signature of this function is as follows, whereby the import of the class Any from the typing module is only for understanding:

```
1 from typing import Any
2 def store_plot_data(
3 data: dict[str, NDArray], file_path: str, group_path: str, metadata: dict[str, Any]
4 )-> None:
5  . . .
```

The transfer parameters are the data in the form of a dictionary, the path to the HDF5 file to be created, the path within the HDF5 file to the group in which the data is stored, and a dictionary that contains the metadata about the data. The purpose of this function is to convert data from a dictionary into a pandas.DataFrame and save it, together with metadata, in an HDF5 file. Save the metadata as attributes of the group. The function has no return value.

(Note: Use the functionalities of the class pandas.HDFStore.)

c) Implement the read_plot_data() function. The signature of this function is as follows:

```
1 def read_plot_data(
2 file: str, group_path: str
3 )-> tuple[pd. DataFrame, dict[str, Any]]:
4  . . .
```

The transfer parameters are the path to the HDF5 file from which data is to be read and the path to the group within the HDF5 file from which the data is to be read. The purpose of this function is to read the data and metadata from the specified group of the HDF5 file, and to automatically generate the legend, the title of the legend, and the axis labels for a plot from the metadata. The return value of the function is a tuple with the following components: (i) a pandas.DataFrame with the archived data, and (ii) a dictionary for the axis labels and the title of the legend. (Note: You can assume that the users of the function adhere to the naming convention of the metadata dictionary. This means that at least the keys defined in the minimal example in the metadata dictionary exist, see task 5a).

d) Switch back to the main.py script. Call the store_plot_data() function there. Pass the Dictionary df_data, the path h5_path, the HDF5 group path group_path, and the Metadata Dictionary metadata.

Task 6: Visualize data

The visualization of data is necessary in order to make it possible to quickly grasp correlations that can be recognized from the analyzed data. Now plot your archived data. To do this, use the read_plot_data() function, among others.

a) First, implement the plot_data() function in the functions.py file. The signature of this function is as follows, whereby the import of the class Figure from the matplotlib.figure module is used for understanding:

```
1 from matplotlib.figure import Figure
2 def plot_data(
3 data: pd. DataFrame, formats: dict[str, str])-> Figure:
4  . . .
```

The transfer parameters are a Pandas DataFrame and a dictionary with the information relevant to the formatting of the graph (diagram), such as the axis labels. The return value is a Matplotlib Figure. Generate a graph for the internal energy of a vat in Gigajoules over time in hours within the function

and stylize the graph in an appealing way. You must represent the four internal energy curves you calculated that depend on the filter size used.

• Choose appropriate x- and y-axis intercepts for your diagram.

• Add axis labels and a legend to your diagram to explain the data and explain the operating conditions. To do this, use the return values of the function read_ plot_data()

b) Implement the publish_plot() function. The signature of the function is as follows:

```
1 def publish_plot(
2 fig: Figure, source_paths: str | list[str], destination_path: str
3 )-> None:
4  . . .
```

The transfer parameters are a Matplotlib Figure that contains the graph, one or more paths to the files needed to create the graph, and the path to the folder where the graph and data are to be stored. Save your diagram using the plotid module. In the tagplot() function, use "time" as id_method and "GdD_WS_2425_" as a prefix. Use the publish() function to publish the chart and the relevant data.

c) Go back to the main.py file. In the main() function, use the read_plot_data() function to read the processed data. Generate your diagram using the plot_data() function you just implemented and save the diagram, the HDF5 file you created, the functions.py module you implemented, and the main.py file using the publish_plot function. Use "./plotid" as a destination_path.

Information on the submission of the project

1) Version the file main.py as well as the project/functions.py module. The HDF5 file must not be versioned. Do not upload the original HDF5 file.

3) Upload the files in a zip archive. Name the zip file according to the following scheme: GdD_WS2425_ETPJ.zip

4) The zip archive contains the following files:

a) the implemented module project/functions.py

b) the implemented script main.py

c) the hidden folder .git

d) the folder of the target of plotid, respectively the target folder of the function publish_plot(): plotid/GdD_WS_2425_/. This contains the following files:

- the dataset data_GdD_plot_WS2425.h5 you archived
- the plot with the ID generated by plotid
- copy of the project/functions.py file
- copy of the file main.py
- the automatically generated environment required_imports.txt