# AUTONOMOUS OPEN DATA PREDICTION FRAMEWORK

## Documentation

Riga Technical University

Faculty of Computer Science and Information Technology

Information Technology Institute

Department of Management Information Technology

**Jānis Pekša**

Research Assistant

http://iti.rtu.lv/vitk/lv/katedra/darbinieki/janis-peksa

**INDEX**

# 1. INTRODUCTION

This is the detailed documentation of the program, which is part of the Autonomous Open Data Prediction Framework. Here are described methods to implement all essential functionality. Some of the methods aren't described because they are built-in in Python or taken from Python modules. It will be announced if some of Python files described below use other modules for Python. All these functions, methods, and modules written not by me have their references, description, and examples of usage on the Internet. All of them can be used freely, as provided and without almost any limitations.

# 2. WEBSITE CONNECTION

The first source of data is the website www.lvceli.lv/cms. The **Request.py** to implement the needed functionality. At the core of it are such modules as **requests** and **BeautifoulSoup.** First, one is needed to make GET/POST requests to the website and the second one is needed to process HTML properly which is returned by requests.o

Examples of using and explanation:

```
table = Request.get_table()
```

Firstly, to get a table from the website. This method does everything it needs to return the correct table with observations that it contains. No need to pass the URL of the website because it's given as a default parameter, as shown in the definition below. Now to have a table with data, but before need to separate values from HTML because this method only returns part of the webpage.

```
data_lists = Request.get_data_lists_from_table(table)
```

Now using another method that will extract all data from HTML, save it to lists and return a list of records like [[], [], []].

This data program can process correctly, export to the database and make estimations but before this operations, some more functionality will be described to understand what is going on inside fully.

Because working with databases to store vast amounts of data, this module is crucial for gathering data in real-time and has some simple methods to work with. Still, it is much more convenient to import/export data using clients of databases that are described next.

*Definitions:*

*Definition of get_table()*

```
def get_table(url='http://www.lvceli.lv/cms/'):
    with requests.session() as s:
        r = s.post(url, data=login_data, headers=headers)

    soup = BeautifulSoup(r.content, 'html5lib')
    return soup.find("table", attrs={"class": "norm", "id": "table-1"})
```

In this method, making a POST request to a website with data that contains login and password to get access to the table with observations. Because every successful application returns a whole webpage to separate our table from the whole piece of HTML to get by using unique attributes of the table that are needed. In details:

soup – an instance of **class BeautifulSoup**, where HTML will be stored.

Using method *find(),* and in arguments, passing the tag that looks for, and it's unique attributes to be sure that a correct table in the result is taken.

*Definition of get_data_lists_from_table()*

```
def get_data_lists_from_table(table) -> list:
    tmp_list = []
    data_list = []

    for row in table.find_all('tr'):
        for cell in row.find_all('td'):
            tmp_list.append(cell.text)

        while len(tmp_list) < 19:
            tmp_list.append('-')

        data_list.append(tmp_list)
        tmp_list = []

    data_list.pop(0)

    return data_list
```

In this method, iterating through every row in the table and through every cell of row (loop in loop, internal circuit) and taking text from HTML tags and adding it to the temporal list, which appends to the main index of data. At the end of the method, the first list of data is being deleted because it contains only titles of columns that is not needed in our main list.

# 3. MYSQL CONNECTION

To be able to get data from the MySQL database, **MySQLClient.py,** which has few methods to download data from a table in the database, and it uses **mysql.connector** module for Python to get a possibility to connect to the database.

A simple instruction to start:

1. Firstly, it is needed to create a client:

```
sql_client = MySQLClient('xx.xx.xx.xx', 'xxx', 'xxx', 'xxx')
```

   In parameters, we pass the IP address of database, login, and password to get access and name of the table to get data from. In this case, it is a database with observations for one month.
2. Now it is connected to the database and have two ways: get all of the data stored in the database or data from exact station/s. It can be either use *get_all_info_from_database*() or *get_info_by_station()/get_info_by_stations()*. The difference is in the amount of returned data and which data is needed. Example:

```
records = sql_client.get_info_by_station('LV01')
```

   Now in variable *records* is stored all of observations from station with code "LV01".

```
records = sql_client.get_all_info_from_database()
```

   But using method to get all of stored observations, it will get all of them and store in variable *records*.

```
records = sql_client.get_info_by_stations(['LV01', 'LV02', 'LV03'])
```

   By using this method it can pass list of station codes and return lists of data for every station that is asked.

*Definitions*

*Definition of MySQLClient __init__() also known as class constructor*

```
def __init__(self, host, username, password, database):
    self.host = host
    self.username = username
    self.password = password
    self.database = database
    self.database_connection = self.connect_to_database()
    self.cursor = self.database_connection.cursor()
```

Here is just create an instance of the class with all the needed variables. *self.cursor* is a variable that is responsible for interaction with the database: making queries, executing commands, and returning results.

*Definition of MySQLClient connect_to_database()*

```
def connect_to_database(self):
    database_connection = mysql.connector.connect(
        host=self.host,
        user=self.username,
        passwd=self.password,
        database=self.database
    )
    return database_connection
```

In this method, an establishing the connection to the database using data provided to *__init__()* and method of **mysql.connector** and returning it to variable in *__init__()*.

*Definition of get_all_info_from_database()*

```
def get_all_info_from_database(self):
    command = 'SELECT * FROM data_records'
    self.cursor.execute(command)
    records = self.cursor.fetchall()
    return records
```

In this method, an executing simple query written and stored in variable *command* and returning all observations from the database.

*Definition of get_info_by_station()*

```
def get_info_by_station(self, station_code: str = 'LV01'):
    command = 'SELECT * FROM data_records WHERE stacijas_kods = %s'
    self.cursor.execute(command, (station_code,))
    records = self.cursor.fetchall()
    return records
```

In this method, an executing the query to the database and returning all rows in the database with data from the station passed in parameters or defined as the default parameter. In variable *command, the* SQL command is constructed to make a proper query.

*Definition of get_info_by_stations()*

```
def get_info_by_stations(self, station_codes: list) -> list:
    info = []
    for station_code in station_codes:
        command = 'SELECT * FROM data_records WHERE stacijas_kods = %s'
        self.cursor.execute(command, (station_code,))
        records = self.cursor.fetchall()
        info.append(records)
    else:
        return info
```

This method does almost everything the same as the previous one; however, for many stations and returns, a list of records from many stations.

# 4. MONGODB CONNECTION

## *I. Difference between database clients*

Comparing with the MySQL module, the program has methods both for importing and exporting data if a user with a login/password that is passed to *MongoDBClient* has permission to do so.

This program is capable of working with MongoDB as well. Methods are located in **MongoDBClient.py** and **AutoDBFiller.py**. They use **pymongo** module to connect to the MongoDB cluster, where databases with collections of data are stored.

A simple instruction to start:

1. In the case with MySQL, first of all, is creating a client that will connect to the cluster:

```
mongo_db_client = MongoDBClient('login', 'password', 'MeteoInfoTable',
'LastInsertTable')
```

In arguments, passing login and password to connect to the cluster, name of the database where observations are stored, and name of the database where the time of the last insert is stored.

*Definitions*

*Definition of MongoDBClient __init__() also known as class constructor*

```
def __init__(self, login: str, password: str, main_database_name: str,
time_database_name: str):
    self.my_client = self.get_connection(login, password)
    self.main_database_name = main_database_name
    self.__main_database = self.my_client[self.main_database_name]
    self.time_database_name = time_database_name
    self.__time_database = self.my_client[self.time_database_name]
    self.is_it_first_filling = self.is_not_first_filling_made()
```

In this method, creating our client by establishing a connection and defining essential variables.

*Definition of get_connection()*

```
def get_connection(login, password):
    return pymongo.MongoClient(
        "mongodb+srv://xxxxx".format(login, password))
```

In this method an establishing connection using login and password provided to *__init__()* and returning it to class constructor.

## *II. MongoDB Importing*

```
1. records = mongo_db_client.get_all_info_from_main_database()
```

Now it is stored all information from main database in list of lists with dictionaries inside. Consider this as two-dimension array of dictionaries like [[{}, {}, {}], [{}, {}, {}]]

```
2. records = mongo_db_client.get_data_from_collection('A1 km 12 Ādaži')
```

If needed to get info from one station, it is possible to use this method and pass the name of the station.

```
3. records = mongo_db_client.get_data_from_collections(['A1 km 12
   Ādaži', 'A1 km 21 Lilaste'])
```

In case needed observations from many stations it is possible to use this method and pass list of station names.

4. Please, be attentive that in the case of MySQL database, station codes were used but with MongoDB names of stations!
5. How to process this data correctly and use it in estimations is described later.

*Definitions*

*Definition of get_all_info_from_main_database()*

```
def get_all_info_from_main_database(self) -> list:
    collection_names = self.__main_database.list_collection_names()
    list_of_info = []
    temp_list = []
    for name in collection_names:
        for search_result in self.__main_database[name].find():
            search_result.pop('_id')
            search_result['Station'] = name
            temp_list.append(search_result)
        list_of_info.append(temp_list)
        temp_list = []

    return list_of_info
```

In this method, calling for all collections present in the main database, and depending on names returned in list requests are being made in the loop. In *for* loop, we are deleting *_id* keyword with the value from every returned result because it's unnecessary and adding *Station* keyword with a value of the name of collection getting at this moment, so every result has the name of the station it relates to. As a result, getting a list of lists of dictionaries as it was described earlier.

*Definition of get_data_from_collection()*

```
def get_data_from_collection(self, station_name: str) -> list:
    list_of_info = []
    for search_result in self.__main_database[station_name].find():
        search_result.pop('_id')
        search_result['Station'] = station_name
        list_of_info.append(search_result)
    else:
        return list_of_info
```

In this method, passing the name of one station as a parameter and making a request to database, processing results of search the same as in previous method and returning list of info.

*Definition of get_data_from_collections()*

```
def get_data_from_collections(self, station_names: list) -> list:
    list_of_info = []

    for station_name in station_names:
        for search_result in self.__main_database[station_name].find():
            search_result.pop('_id')
            search_result['Station'] = station_name
            list_of_info.append(search_result)
    else:
        return list_of_info
```

In this method, passing the list of station names, but the logic is the same as in two previous methods. The only difference is that it iterates through, the given list of names, that contain those names which the user-provided.

## *III. MongoDB Exporting*

1. To use all of functionality it should recreate client and use AutoDBFiller()

   ```
   mongo_db_client = AutoDBFiller('login', 'password',
   'MeteoInfoTable', 'LastInsertTable')
   ```

   Parameters are the same as in case with MongoDBClient() because ***class AutoDBFiller*** is child-class of ***class MongoDBClient*** and has all of it methods and some new.
2. As described earlier, it is possible to use requests to take data from the website, but this data has to be processed before it can be exported to the database, so be sure to check the Data Handling chapter and methods presented there.

# 5. DATA HANDLING

Created **DataHandler.py** with methods to process data in different ways to prepare it for different usage. This chapter will be seprated in many small to systemize data and increase the readability of the document.

## *I. For exporting to MongoDB*

After getting the list of lists of values from the website, need to turn it into a JSON-like document to be able to export it to MongoDB because this database contains all records in JSON. Python is a data type called the *dictionary,* so a set of methods that will help to prepare all the data to export.

```
data_dicts = DataHandler.get_data_dicts(data_lists)
```

Here useing variable *data_lists* from the Website Connection chapter, which contains a list of lists of values from the website. After calling this method, turning a list of lists into a list of dictionaries like [{}, {}, {}]

Now data can be exported to the database. This process will be described in the MongoDB Exporting chapter.

*Definitions*

*Definition of get_data_dicts()*

```
def get_data_dicts(data_lists) -> list:
    data_dicts = []

    for data in data_lists:
        data_dict = {'Station': data[0],
                     'Time': data[1],
                     'Date': DataHandler.get_date(),
                     'Air Temperature': data[2],
                     'Air Temperature(-1 h)': data[3],
                     'Humidity': data[4],
                     'Dew Point': data[5],
                     'Precipitation': data[6],
                     'Intensity': data[7],
                     'Visibility': data[8],
                     'Road Temperature': data[9],
                     'Road Temperature(-1 h)': data[10],
                     'Road Condition': data[11],
                     'Road Warning': data[12],
                     'Freezing Point': data[13],
                     'Road Temperature 2': data[14],
                     'Road Temperature 2(-1 h)': data[15],
                     'Road Condition 2': data[16],
                     'Road Warning 2': data[17],
                     'Freezing Point 2': data[18]}
        data_dicts.append(data_dict)

    return data_dicts
```

This method iterates through a list of data, takes values from the list by indexes they are stored, and append created dictionary to list which will be returned in the result.

*Definition of get_date()*

```
def get_date() -> str:
    from datetime import datetime

    year = datetime.now().year
```

```
    month = datetime.now().month
    day = datetime.now().day

    return 'Year: {} Month: {} Day: {}'.format(year, month, day)
```

This method takes current date-time, adds its parts to string (year, month, and day), and returns this string. It uses **datetime** module to obtain the current date and time.

## *II. Process data from MongoDB*

*Data from MongoDB database*

```
lists_of_measurements, list_of_station_names =
DataHandler.get_prepared_lists_for_estimation(main_data, 'Dew Point')
```

Called method returns two lists: with measurements (observations) and with station names whose data returned. Because of the chance that sometimes stations haven't been gathering data program checks every list and, if it is possible to fill missing data, it fills, if not – deletes the list of data. Station names aren't needed for estimation but are essential for building plots. Plots are described later.

*Definitions*

*Definition of get_prepared_lists_for_estimation()*

```
def get_prepared_lists_for_estimation(main_data, value: str = 'Dew Point') -> tuple:
    lists_of_chosen_values = DataHandler.get_lists_of_chosen_values_with_station_names(main_data, value)

    lists_of_measurements_with_station_names = DataHandler.get_lists_of_measurements_with_station_names(
        lists_of_chosen_values)

    list_of_station_names = DataHandler.get_station_names(lists_of_measurements_with_station_names)

    lists_of_measurements_without_station_names = DataHandler.get_lists_of_measurements_without_station_names(
        lists_of_measurements_with_station_names)

    return lists_of_measurements_without_station_names, list_of_station_names
```

This method does a lot, but basically, it does what it has to. Here are four steps:

1. Getting a list of lists of chosen values or values defined as a default parameter, and at the end of every list, appending the name of the station to which it is related. For this goal is used *get_lists_of_chosen_values_with_station_names()*.
2. Because all values are stored as text(string type), to convert it to numbers to be able to work with data. For this goal is used *get_lists_of_measurements_with_station_names()* In this method are being checked lists of data is it possible to fill missing data and if it is possible, it is being filled. As a result, suitable lists returned. Those ones who weren't fillable were deleted.
3. Now the program starts dividing the list of data to two lists: list of station names and list_of_measurements. For first goal is used *get_station_names()*.
4. *For* last step is used *get_list_of_measurements_without_station_names()*.

*Definition of get_lists_of_chosen_values_with_station_names()*

```
def get_lists_of_chosen_values_with_station_names(lists_of_dicts: list, value_name: str) ->
list:
    list_of_values = []
    temp_list = []

    for list_of_dicts in lists_of_dicts:
        for data_dict in list_of_dicts:
            temp_list.append(data_dict[value_name])
        else:
            temp_list.append(data_dict['Station'])
        list_of_values.append(temp_list)
```

```
        temp_list = []

    return list_of_values
```

Here is a simple method that fills the list with lists of values that are being passed as argument and station names values belong to.

*Definition of get_lists_of_measurements_with_station_names()*

```
def get_lists_of_measurements_with_station_names(lists_of_values: list) -> list:
    lists_of_measurements = []

    for list_of_values in lists_of_values:
        if '-' in list_of_values or '' in list_of_values:
            list_of_values.pop()
            if DataHandler.is_possible_to_fill_missing_data(list_of_values):
                indexes = DataHandler.get_indexes_for_filling(list_of_values)
                list_of_values = DataHandler.fill_missing_data(list_of_values, indexes)
            else:
                continue
        lists_of_measurements.append(DataHandler.get_list_of_float_numbers_and_station(list_of_values))

    return lists_of_measurements
```

This method iterates through lists of values, and if there are '-' or empty spots, these lists are sent to the special method to be checked if it is possible to fill missing data, and if it is, missing data will be filled. If it is not possible, these lists will not be added to the final list of data and then deleted from memory. Before appending list of values to main lists all values are turned to *float* type in *get_list_of_float_numbers_and_station()*.

*Definition of is_possible_to_fill_missing_data()*

```
def is_possible_to_fill_missing_data(list_of_values: list) -> bool:
    index = 0
    adder = 1

    while adder < 5 and index < len(list_of_values) - 1 and index + adder < len(list_of_values):
        if list_of_values[index] == '-' and list_of_values[index + adder] == '-':
            adder += 1
        else:
            index += adder
            adder = 1
    else:
        if '' in list_of_values:
            return False
        if adder > 4:
            return False
        elif 1 < adder < 5 and adder == len(list_of_values):
            return False
        else:
            return True
```

This method simply counts the quantity of missing data in a row, and if there is somewhere in the list, five and more missing data samples method returns *False*. It will return *False* also if there is just empty string because that means that all list is empty(a feature of the website).

*Definition of get_indexes_for_filling()*

```
def get_indexes_for_filling(list_of_values: list) -> tuple:

    index = 0
    adder = 1

    indexes = []

    while adder < 5 and index < len(list_of_values) - 1 and index + adder < len(list_of_values):
        if adder == 1 and list_of_values[index] == '-' and list_of_values[index + 1] != '-':
            indexes.append([index, 'one'])

        if list_of_values[index] == '-' and list_of_values[index + adder] == '-':
            adder += 1
        elif adder != 1:
            indexes.append([index - 1, index + adder, 'normal'])
            index += adder
            adder = 1
        else:
            index += adder
            adder = 1
    else:
        if adder != 1:
            indexes.append([index - 1, index + adder, 'in the end'])
```

This method searches for all spots in the list and creates a list of coordinates for another method to fill missing data later correctly. There are three types of cords: usual series of missing spots, when only one spot missing data(in real life there is very little chance that this situation will occur, but I left this variant just in case) and when series of missing spots are located at the end of the list. How these scenarios are processed is shown in the next method definition.

*Definition of fill_missing_data_in_list()*

```
def fill_missing_data_in_list(list_of_values: list, indexes: tuple) -> list:
    for index_list in indexes:
        if index_list[-1] == 'normal':
            average = (list_of_values[index_list[0]] + list_of_values[index_list[1]]) / 2

            for i in range(index_list[0] + 1, index_list[1]):
                list_of_values[i] = average
        elif index_list[-1] == 'one':
            average = (list_of_values[index_list[0] - 1] + list_of_values[index_list[0] + 1]) / 2
            list_of_values[index_list[0]] = average
        elif index_list[-1] == 'in the end':
            for i in range(index_list[0], index_list[1]):
                list_of_values[i] = list_of_values[index_list[0]]

    return list_of_values
```

This method returns the list with filled missing spots. It iterates through a list of coordinates for filling and depending on coordinates and type of variant of the positioning of missing spots filling occurs.

*Definition of get_list_of_float_numbers_and_station()*

```
def get_list_of_float_numbers_and_station(list_of_values: list) -> list:
    list_of_data = []

    for value in list_of_values:
        try:
            list_of_data.append(float(value))
        except ValueError:
            list_of_data.append(value)

    return list_of_data
```

The simple method to turn every value in the list from *str* to *float* except station name. The method returns a new list of data.

*Definition of get_station_names()*

```
def get_station_names(list_of_lists_of_data: list) -> list:
    list_of_station_names = []

    for list_of_data in list_of_lists_of_data:
        list_of_station_names.append(list_of_data[-1])

    return list_of_station_names
```

A simple method that just takes the last element from every list which is station name and appends it to list of station names.

*Definition of get_lists_of_measurements_without_station_names()*

```
def get_lists_of_measurements_without_station_names(
        list_of_lists_of_measurements_with_station_names: list) -> list:
    list_of_lists_of_measurements_without_station_names = []

    for list_of_measurements in list_of_lists_of_measurements_with_station_names:
        list_of_measurements.pop(len(list_of_measurements) - 1)
        list_of_lists_of_measurements_without_station_names.append(list_of_measurements)

    return list_of_lists_of_measurements_without_station_names
```

A simple method that iterates through every list deletes the last element of every list, which is station name and append these lists to the final list that will be returned.

## *III. Process data from MySQL*

```
index = DataHandler.get_index_of_value('Dew Point')
```

If needed to estimate data from MySQL database first needs to be done is get index of value that are needed to make estimation with.

```
list_of_measurements = DataHandler.get_exact_value_from_my_sql_records(records, index)
```

Now after getting exact value from records returned from MySQL database.

IMPORTANT NOTE!

```
records = sql_client.get_info_by_stations(['LV01', 'LV02', 'LV03'])
index = DataHandler.get_index_of_value('Dew Point')
lists_of_measurements = DataHandler.get_exact_value_from_many_my_sql_records(records,
index)
```

If needed to get values from some exact stations than should use other method to extract value because previous method is only for processing one station.

```
lists_of_measurements = DataHandler.get_lists_of_floats(lists_of_measurements)
list_of_measurements = DataHandler.get_list_of_floats(list_of_measurements)
```

Depending on situation it should be using method for process info from one station or from many stations.

At this moment, it should have a list of values or lists of values. However, in this list/lists may be missing values of type *None.* A simple method to replace them with a dash of type *str,* so these lists could be suitable for usage with methods for filling missing data I described earlier.

```
lists_of_measurements =
DataHandler.replace_none_with_dash(lists_of_measurements)
```

This method accepts a list of lists and returns it after processing.

Now passing lists for filling missing data if it is possible.

```
lists_of_measurements = DataHandler.fill_missing_data_in_lists(lists_of_measurements)
```

This method accepts a list of lists as well and returns it. Some details of functionality will be described in the next Definition chapter.

However, in lists that returned are present elements that can't be used in estimation due to not every list, missing data were filled.

Now the only option is to check which lists contain *str* elements, dashes put instead of *None,* or empty at all and remove them from the main list of measurements. Besides, station codes should be updated because our data needs to be related to the correct station.

```
station_codes = DataHandler.get_station_codes()
```

The first step – just create a list of station codes using this simple method.

```
station_codes, lists_of_measurements =
DataHandler.zip_codes_and_measurements(station_codes, lists_of_measurements)
```

Now passing our lists in the following method, and it will return corrected lists that can be used in estimation and plot building.

After these manipulations, data is ready for estimation. The process of evaluation is described later.

*Definitions*

*Definition of get_index_of_value()*

```
def get_index_of_value(value: str = 'Dew Point') -> int:
    if value == 'id':
        return 0
    elif value == 'Station code':
        return 1
    elif value == 'Datetime':
        return 2
    elif value == 'Air Temperature':
        return 3
    elif value == 'Air Temperature(-1 h)':
        return 4
    elif value == 'Humidity':
        return 5
    elif value == 'Dew Point':
        return 6
    elif value == 'Precipitation':
        return 7
    elif value == 'Intensity':
        return 8
    elif value == 'Visibility':
        return 9
    elif value == 'Road Temperature':
        return 10
    elif value == 'Road Temperature(-1 h)':
        return 11
    elif value == 'Road Condition':
        return 12
    elif value == 'Road Warning':
        return 13
    elif value == 'Freezing Point':
        return 14
    elif value == 'Road Temperature 2':
        return 15
    elif value == 'Road Temperature 2(-1 h)':
        return 16
    elif value == 'Road Condition 2':
        return 17
    elif value == 'Road Warning 2':
```

```
        return 18
    elif value == 'Freezing Point 2':
        return 19
```

A simple method that returns *int* as an index of value passed in argument or which is defined as a default parameter.

*Definition of get_exact_value_from_my_sql_records()*

```
def get_exact_value_from_my_sql_records(records: tuple, index: int) -> list:
    values = []

    for record in records:
        values.append(record[index])

    return values
```

A simple method that just iterates through records and takes elements by index and adds them in a list that will be returned in the end.

*Definition of get_exact_value_from_many_my_sql_records()*

```
def get_exact_value_from_many_my_sql_records(list_of_records: list, index: int) -> list:
    values = []

    for records in list_of_records:
        values.append(DataHandler.get_exact_value_from_my_sql_records(records, index))

    return values
```

This method does the same as the previous one but for records from many stations. Returns list of lists.

*Definition of replace_none_with_dash()*

```
def replace_none_with_dash(lists_of_measurements: list) -> list:
    i = 0

    for list_of_measurements in lists_of_measurements:
        for _ in list_of_measurements:
            if list_of_measurements[i] is None:
                list_of_measurements[i] = '-'
            i += 1
        i = 0

    return lists_of_measurements
```

This simple method just iterates through every element of every row and, if it is *None* type, replaces it on ‘-‘ character.

*Definition of fill_missing_data_in_lists()*

```
def fill_missing_data_in_lists(lists_of_measurements: list) -> list:
    i = 0
    for list_of_measurement in lists_of_measurements:
        if DataHandler.is_possible_to_fill_missing_data(list_of_measurement):
            indexes = DataHandler.get_indexes_for_filling(list_of_measurement)
            list_of_measurement = DataHandler.fill_missing_data_in_list(list_of_measurement, indexes)
            lists_of_measurements[i] = list_of_measurement
            i += 1
        else:
            i += 1
    return lists_of_measurements
```

This method iterates through every list and checks if it is possible to fill missing data or not and if it is, it makes filling using already described methods.

IMPORTANT NOTE!

This method works with the whole list at a time, so if even there are only one series of missing data that couldn’t be filled, the entire list will be ignored.

*Definition of get_station_codes()*

```
def get_station_codes() -> list:
    station_codes = []

    for i in range(1, 65):
        station_code = 'LV{:02d}'.format(i)
        station_codes.append(station_code)

    return station_codes
```

This method simply creates a list, fills it with station codes in *str* type, and returns it in the result.

*Definition of zip_codes_and_measurements()*

```
def zip_codes_and_measurements(station_codes: list, lists_of_measurements: list) -> tuple:
    stop = len(lists_of_measurements)
    i = 0

    while i < stop:
        if '-' in lists_of_measurements[i] or not bool(lists_of_measurements[i]):
            lists_of_measurements.pop(i)
            station_codes.pop(i)
            stop -= 1
            i = 0
        else:
            i += 1

    return station_codes, lists_of_measurements
```

This method checks every list is it containing dashes or is empty, and if it is, then it deletes this list from the main list and corresponding station code from the list of station codes. As a result, modified lists are returned.

## *IV. Accuracy*

This chapter will be useful after getting familiar with the Estimation chapter because skill is counted using measurements and estimates. Anyway, the calculation of efficiency is handled by **DataHandler.py,** so this process will be described here.

Few steps should be made to calculate accuracy:

1. Cast all data values to *int* type
   ```
   lists_of_estimates      =      DataHandler.get_lists_of_ints(lists_of_estimates)
   lists_of_measurements = DataHandler.get_lists_of_ints(lists_of_measurements)
   ```
2. Use method, pass lists with data as argument and will get list of accuracies
   ```
   accuracies = DataHandler.get_accuracies(lists_of_measurements, lists_of_estimates)
   ```
   Now having list of accuracies of stations that were used in estimation processs.

*Definitions*

*Definition of get_lists_of_ints()*

```
def get_lists_of_ints(lists_of_values) -> list:
    for values in lists_of_values:
        i = 0
        while i < len(values):
            values[i] = int(values[i])
            i += 1

    return lists_of_values
```

This method iterates through every element of every list and casts it to *int*.

*Definition of get_accuracies()*

```
def get_accuracies(lists_of_measurements: list, lists_of_estimates: list) -> list:
    accuracies = []
    for measurements, estimates in zip(lists_of_measurements, lists_of_estimates):
        accuracies.append(DataHandler.get_accuracy(measurements, estimates))

    return accuracies
```

This method iterates through every list of measurements, estimates, and appends the result of *get_accuracy()* method, which is described below.

*Definition of get_accuracy()*

```
def get_accuracy(measurements: list, estimates: list) -> float:
    from sklearn.metrics import accuracy_score

    return accuracy_score(measurements, estimates)
```

This method uses a method from **sklearn** module to calculate accuracy and returns the result.

# 6. ESTIMATION

For estimation purposes, **KalmanFilter.py** is created that has a basic implementation of the Kalman Filter for a one-dimension model.

```
kalman_filter = KalmanFilter(error_in_estimate, initial_estimate,
error_in_estimate, measurements)
```

The first step is to create Kalman filter object and pass needed arguments: error in the estimate, an initial estimate, error in the estimate, and list of measurements that will be used in the estimation process.

```
estimates = kalman_filter.get_estimates()
```

Now it can get a list of estimates by using only one method.

However, if working with a list of lists, then can use the static method from **KalmanFilter.py** that will handle essential processes and return us a list of lists of estimates.

```
lists_of_estimates = KalmanFilter.get_lists_of_estimates(lists_of_measurements,
error_in_estimate, error_in_measurement)
```

In arguments, passing lists of measurements, errors in the estimate, and error in measurement. As a result, getting our estimations that can be used for building plots. Plots are described later.

*Definitions*

*Definition of KalmanFilter __init__() also known as class constructor*

```
def __init__(self, initial_error_in_estimate, __initial_estimate,
__error_in_measurement, __measurements):
    self.__measurements = __measurements
    self.__estimate = __initial_estimate
    self.__error_in_estimate = initial_error_in_estimate
    self.__error_in_measurement = __error_in_measurement
    self.__measurement = None
    self.__kalman_gain = None
```

In this method, is just declare essential variables for the estimation process.

*Definition of get_estimates()*

```
def get_estimates(self) -> list:
    list_of_estimates = []

    for measurement in self.__measurements:
        self.__measurement = measurement
        self.make_basic_calculations()
        list_of_estimates.append(self.__estimate)

    return list_of_estimates
```

This method iterates through all measurements in the list of measurements that are passed when having been creating KalmanFilter() object. *make_basic_calculations()* does exactly what it stands for; its definition is provided below. After every calculation new estimate is appended in the list of estimates that will be returned in the result.

*Definition of make_basic_calculations()*

```
def make_basic_calculations(self):
    self.__kalman_gain = self.__calculate_kalman_gain()
    self.__estimate = self.__calculate_estimate()
    self.__error_in_estimate = self.__calculate_error_in_estimate()
```

This method uses other simple methods to produce Kalman Filter calculations for a one-dimension model and make an estimation. The description of every method is given below.

*Definition of __calculate_kalman_gain()*

```
def __calculate_kalman_gain(self) -> float:
    return self.__error_in_estimate / (self.__error_in_estimate +
self.__error_in_measurement)
```

*Definition of __calculate_estimate()*

```
def __calculate_estimate(self) -> float:
    return self.__estimate + self.__kalman_gain * (self.__measurement -
self.__estimate)
```

*Definition of __calculate_error_in_estimate()*

```
def __calculate_error_in_estimate(self) -> float:
    return (1 - self.__kalman_gain) * self.__error_in_estimate
```

*Definition of get_lists_of_estimates()*

```
def get_lists_of_estimates(lists_of_measurements: list, error_in_estimate: float,
                           error_in_measurement: float) -> list:
    lists_of_estimates = []

    for measurements in lists_of_measurements:
        initial_estimate = KalmanFilter.__get_initial_estimate(measurements)
        kf = KalmanFilter(error_in_estimate, initial_estimate,
error_in_measurement, measurements)
        lists_of_estimates.append(kf.get_estimates())

    return lists_of_estimates
```

This method iterates through lists of measurements, automatically creates initial estimate, KalmanFilter object, and append list of estimates to final list that will be returned in the result.

*Definition of __get_initial_estimate()*

```
def __get_initial_estimate(measurements: list) -> float:
    return measurements[-1] + (
            measurements[-1] - measurements[-2])
```

This method returns an average of last two elements.

# 7. PLOTS

This program has **GraphEditor.py** that uses **matplotlib** module to create plots.

To create one plot or plots for a list of estimates, it is needed to create an object of ***class GraphEditor*** and use correct proper methods:

```
graph = GraphEditor(estimates, measurements, value, station_code, period)
graph.create_est_and_meas_plot()
```

Also, there are variants of plots that can be created:

```
graph.create_est_plot()
graph.create_meas_plot()
```

Creating separate plots for estimates and measurements.

To show the created plot, the following method should be used:

```
graph.show_plot()
```

It will display a plot in the program that are used, but if need to save the created plot, this method should be used for saving:

```
graph.save_plot()
```

IMPORTANT NOTE!

Don't show plot and then try to save it, it will not work. Separately these methods work just fine.

*Definitions*

*Definition of GraphEditor __init__ also known as class constructor*

```
def __init__(self, estimates: list, measurements: list, value: str, station_name:
str, period: str, accuracy=None):
    self.estimates = estimates
    self.measurements = measurements
    self.value = value
    self.station_name = station_name
    self.period = period
    self.accuracy = accuracy
    self.plt = None
```

Initialization of class object and defining variables.

*Definition of create_est_and_meas_plot()*

```
def create_est_and_meas_plot(self):
    plt.plot(self.estimates, label='estimates', color='blue')
    plt.plot(self.measurements, label='measurements', color='orange')
    plt.title(self.station_name + ', Period: ' + self.period)
    plt.ylabel(self.value)
    plt.xlabel('Observations')
    plt.legend()
    self.plt = plt
```

This method setups settings for plot such as title, legend, labels for x-y axis, and saves plot for further usage like saving or showing in the program.

*Definition of create_est_plot()*

```
def create_est_plot(self):
    plt.plot(self.estimates, color='blue')
    plt.title('Estimates: ' + self.station_name + ', Period: ' + self.period)
    plt.ylabel(self.value)
    plt.xlabel('Observations')
    self.plt = plt
```

This method does the same things but only for estimations.

*Definition of create_meas_plot()*

```
def create_meas_plot(self):
    plt.plot(self.measurements, color='orange')
    plt.title('Measurements: ' + self.station_name + ', Period: ' + self.period)
    plt.ylabel(self.value)
    plt.xlabel('Observations')
    self.plt = plt
```

This method creates a plot for measurement in the way, like two previous methods.

*Definition of show_plot()*

```
def show_plot(self):
    self.plt.show()
```

Simple method that shows plot in program.

*Definition of save_plot()*

```
def save_plot(self):
    self.plt.savefig('Plot{}.pdf'.format(GraphEditor.index), dpi=300)
    GraphEditor.index += 1
```

A method that saves plot to project folder with settings that were passed as arguments for *savefig()* method.

# 8. ARIMA

## *I. ARIMA main*

For making forecast with ARIMA model is used **Arima_Main.py** with modules such as **pandas** for working with data frames and data series, **numpy** and **ARIMA** from **statsmodels.tsa.arima_model**. For making forecast and receive all information that was used for forecasting, it is needed to create class ***Arima_Main*** and use correct method as shown here:

```
data_points_for_forecast, arima_model_order, steps, forecast =
Arima_Main(series_of_measurements, steps, optimize).get_arima_forecast()
```

IMPORTANT NOTE!

*series_of_measurements* must contain only one data column and one index column, because for making forecast are used *series_of_measurements* values, it means it can be used only DataFrame with one data column of just Series.

*Definition of get_arima_forecast()*

```
def get_arima_forecast(self):

    arima_models_order_and_data_points_dict =
Arima_Main.get_all_possible_arima_models_with_data_points(self)

    # dataframe of data points, steps and RMSE
    rmse_df = pd.DataFrame()

    rmse_dict = dict()
    last_point = len(self.data_series) - 1 - self.steps

    for data_points, order in arima_models_order_and_data_points_dict.items():

        first_point = last_point - data_points
        data_series_copy = self.data_series[first_point:last_point].copy()

        # ----- Make copy of real data for comparing with forecast ----- #
        station_data_copy_with_steps = self.data_series[last_point - 1:last_point
- 1 + self.steps].copy()

        forecast_list = Arima_Main.make_forecast(order, data_series_copy,
self.steps)

        rmse = Arima_Main.get_forecast_accuracy_with_real_data(forecast_list,
station_data_copy_with_steps.values)
        rmse_dict[data_points] = rmse
```

```
    rmse_series = pd.Series(rmse_dict)
    rmse_df[str(self.steps)] = rmse_series

    rmse_df.reset_index(inplace=True)
    rmse_df['order'] = arima_models_order_and_data_points_dict.values()
    rmse_df.rename(columns={'index': 'points'}, inplace=True)
    rmse_df.set_index(['points', 'order'], inplace=True)

    results = Arima_Main.get_steps_and_points_of_min_rmse(rmse_df)

    data_points_for_forecast = results[0][0]

    steps = int(results[1])

    data_series_copy = self.data_series[- data_points_for_forecast:].copy()

    arima_model_order = Arima_Order(data_series_copy).get_arima_best_order()

    forecast = Arima_Main.make_forecast(arima_model_order, data_series_copy,
steps)

    return [data_points_for_forecast, arima_model_order, steps, forecast]
```

As a result, we get a list, that contains:

- How much data points must be taken for making forecast.
- ARIMA best model's order (p, d, q).
- How much steps in the future must be taken for making forecast.
- List of forecast values.

*Definition of get_all_possible_arima_models_with_data_points()*

```
def get_all_possible_arima_models_with_data_points(self):
    if self.optimize:
        if len(self.data_series) <= 1000 + self.steps:
            optimize_point = len(self.data_series)
        else:
            optimize_point = 400
    else:
        optimize_point = len(self.data_series)
    arima_models_order_and_data_points_dict = dict()
    last_point = len(self.data_series) - 1 - self.steps
    process = 0
    data_point_range = range(100, optimize_point, 10)
    data_point_range_len = len(range(100, optimize_point, 10))
    for data_points in data_point_range:
        process += 1
        first_point = last_point - data_points
        if first_point < 0:
            break
        data_series_copy = self.data_series[first_point:last_point].copy()
        order = Arima_Order(data_series_copy).get_arima_best_order()
```

```
        if order == 0:
            # print('ARIMA model was NOT DEFINED for', len(data_series_copy),
'observation points')
            print('{:.0f}%'.format(process / data_point_range_len * 100))
            continue
        else:
            # print('ARIMA model with order', order, 'for', len(data_series_copy),
'observation points')
            print('{:.0f}%'.format(process / data_point_range_len * 100))
            arima_models_order_and_data_points_dict[data_points] = order

    return arima_models_order_and_data_points_dict
```

This method returns a dictionary with data points and with the best ARIMA model order for this data set. It looks like this:

{ data_points1 : model_order1, data_points2 : model_order2, … }

or

{ 100 : [1, 2, 1], 110 : [2, 1, 4] … }

*Definition of get_forecast_accuracy_with_real_data()*

```
def get_forecast_accuracy_with_real_data(forecast, actual):
    rmse = np.mean((forecast - actual) ** 2) ** .5
    return rmse
```

Returns RMSE as a float value, that characterizes the forecast accuracy with actual data

*Definition of make_forecast()*

```
def make_forecast(arima_model_order, data_series, steps):
    arima_model_station_data = ARIMA(data_series,
order=arima_model_order).fit(disp=0)
    forecast_list = arima_model_station_data.forecast(steps=steps)[0].tolist()

    return forecast_list
```

Method uses the best ARIMA model order, that was find before, and returns list of forecast value. The number of received values in the list is equal with steps value.

## *II. ARIMA order*

To determine ARIMA order is used **ArimaOrder.py** with modules ARIMA from **statsmodels.tsa.arima_model**, **adfuller** from **statsmodels.tsa.stattools** for finding p-value, that indicates if model is stationary or not, **plot_pacf**, **plot_acf** from **statsmodels.graphics.tsaplots** and **pyplot** for showing autocorrelation and partial autocorrelation graphs.

*Definition of get_d_value_and_ADF_test()*

```
def get_d_value_and_ADF_test(self):

    d = 0

    try:
        adf_test_results = adfuller(self.data_series.values)
    except:
        return d

    data_series_diff = self.data_series
    while adf_test_results[1] > 0.05 or d == 0:
        if d > 2:
            return 0
        d += 1
        # ----- make data stationary and drop NA values ----- #
        data_series_diff = data_series_diff.diff().dropna()
        try:
            data_series_diff_values = data_series_diff.values
            adf_result = adfuller(data_series_diff_values)
        except:
            d -= 1
            return d

    # print('ADF p-value:', adf_result[1])
    # print('d:', d)

    # ----- autocorrelation ----- #
    #plot_acf(data_diff)
    #plt.gcf().autofmt_xdate()
    #plt.show()
    # ----- partial autocorrelation ----- #
    #plot_pacf(data_diff)
    #plt.gcf().autofmt_xdate()
    #plt.show()

    return d
```

This method is used to define d value for ARIMA model (p, d , q) order. d represents the number of times that the data have to be "differenced" to produce a stationary signal (i.e., a signal that has a constant mean over time). This captures the "integrated" nature of ARIMA. If d=0, this means that our data does not tend to go up/down in the long term (i.e., the model is already "stationary"). In this case, then technically you are performing just ARMA, not AR-I-MA. If p is 1, then it means that the data is going up/down linearly. If p is 2, then it means that the data is going up/down exponentially. To define d value is used  Augmented Dickey-Fuller test with p-value:

- p-value > 0.05: the data has a unit root and is non-stationary.
- p-value <= 0.05: the data does not have a unit root and is stationary.
- The more negative is ADF Statistic, the more likely we have a stationary dataset.

*Definition of get_arima_best_order()*

```
def get_arima_best_order(self):

    d = Arima_Order.get_d_value_and_ADF_test(self)

    p_values = range(0, 5)
    q_values = range(0, 5)

    aic_dict = dict()

    for p in p_values:
        for d in range(d, 2):
            for q in q_values:
                order = (p, d, q)

                try:
                    arima_station_model = ARIMA(self.data_series,
order).fit(disp=0)
                    aic = arima_station_model.aic
                    # print('ARIMA%s aic = %.5f' % (order, aic))

                    if [p, d, q] == [0, 0, 0] or [p, d, q] == [0, 1, 0] or [p, d,
q]
                            == [0, 1, 1]:
                        continue

                    if aic not in aic_dict:
                        aic_dict[aic] = order

                except:
                    # print('ARIMA%s aic not defined' % (order,))
                    continue

    # if aic_dict is empty
    # it is impossible to create arima model for this data set
    if len(aic_dict) == 0:
        return 0

    min_val = min(aic_dict.keys())
    # print('min AIC value', min_val)
    # print('ARIMA model was created with order', aic_dict[min_val])

    # ----- ARIMA (p, d, q) ----- #
    p = aic_dict[min_val][0]
    q = aic_dict[min_val][2]

    return [p, d, q]
```

This method returns best ARIMA model order using Akaike information criterion for characterizing and d value from *d_value_and_ADF_test* method. AIC estimates the relative amount of information lost by a given model: the less information a model loses, the higher the quality of that model.

# 9. API

The fundamental representation of API that can be used as Web Service of this framework.

Five routes are used:

```
1. /get/estimates/all

2. /get/estimates

3. /get/accuracies

4. /get/forecast/arima

5. /get/forecast/arima/time_period
```

All these requests have essential arguments and secondary ones. Every request has an essential argument *value*.

## *I. /get/estimates/all*

The first route returns JSON with estimates for all stations and chosen value.

Essential argument: *value*

Secondary argument: *measurements*

For example:

/get/estimates/all?value=Dew%20Point

The server will return JSON with estimates for all datasets of Dew Point that are suitable for estimation.

/get/estimates/all?value=Dew%20Point&measurements=true

If add *measurements* argument and set it to true, then JSON will contain measurements for every station as well.

## *II. /get/estimates*

The second route can return JSON with estimates for the exact station or stations.

Essential argument: *value* and *stations*

Secondary argument: *measurements*

For example:

/get/estimates?value=Dew%20Point&stations=LV01

This request will return JSON with estimates of Dew Point for station LV01. To get estimates for many stations adding their code to *stations* argument separated by *comma,* like below:

/get/estimates?value=Dew%20Point&stations=LV01,LV02,LV03

If needed to get measurements too, just add *measurements* argument and set it to true to URL:

/get/estimates?value=Dew%20Point&stations=LV01,LV02,LV03&measurements=true

### *III. /get/accuracies*

Third route is for getting accuracy of estimation.

Essential argument: *value, stations*

Secondary argument: None

For example:

/get/accuracies?value=Dew%20Point&stations=LV01

/get/accuracies?value=Dew%20Point&stations=LV01,LV02,LV03

These requests will return JSON with accuracies for stations and value that are set up in request arguments.

### *IV. /get/forecast/arima*

Is used for getting forecast values using station data.

Essential argument: value, station, steps and optimize.

For *value* argumest can be used “Dew Point”, “Air Temperature” and etc.

*Steps* value means how many points ahead will be made forecast.

*Optimize* option is used to determine the best model using only last 1000 data point. In otherwise the program uses full data and it may take much more time and not make sense.

For example:

/get/forecast/arima?value=Dew%20Point&station=LV01&steps=5&optimize=true

### *V. /get/forecast/arima/time_period*

Is used for getting forecast values using station data and using time period with date from and date till.

Essential argument: value, station, steps, optimize, date_from and date_till.

Date value can be inputed in two formats:

YYYY-MM-DD and time value will be automatically set to 00:00

YYYY-MM-DD_HH:MM

For example:

/get/forecast/arima/time_period?value=Dew%20Point&station=LV01&steps=5&optimize=true&date_from=2020-02-10_12:00&date_till=2020-04-15_14:00

/get/forecast/arima/time_period?value=Dew%20Point&station=LV01&steps=5&optimize=true&date_from=2020-02-10&date_till=2020-04-15