Modules for healthy environment

Release 0.4

Klára Nečasová and Peter Tisovčík

Nov 05, 2019

CONTENTS

CHAPTER

ONE

INSTALLATION

This page describes an installation of the package `dm` and how to integrate it into the project.

1.1 Getting the source code

```
Contact project manager (project no. TH02010845:  IotCloud - Intelligence for
IoT systems) to obtain source code.
```

1.2 Requirements

1.2.1 Ubuntu packages

Use the command below to install the required Ubuntu packages.

```
apt install python3-tk python3-venv
```

1.2.2 A Python virtual environment (optional)

It is suitable to create a Python virtual environment.

```
python3 -m venv env
```

After successful creation, it is necessary to activate the environment.

```
source env/bin/activate # for activate env
```

1.2.3 Python dependencies

List of used packages

```
matplotlib==3.0.2
mysql-connector-python==8.0.15
numpy==1.16.1
pandas==0.24.2
pickle-mixin==1.0.2
pytz==2018.9
requests==2.21.0
```

(continues on next page)

(continued from previous page)

```
scipy==1.2.1
sklearn==0.0
sphinx==2.2.0
sphinx_rtd_theme==0.4.3
sympy==1.3
```

Install Python dependencies using the command below. The installation should be done in the virtual environment.

```
pip install -r requirements.txt
```

1.2.4 Documentation dependencies (optional)

```
apt-get install latexmk texlive-latex-recommended \
                texlive-latex-extra texlive-fonts-recommended
```

To generate the documentation, it is required to go to the `docs` folder. The documentation can be generated either in HTML form or in PDF form. If the documentation is generated in PDF form, it is open in the Opera browser after a successful generation.

```
cd docs
make latex && \
cd build/latex/ && \
make && \
opera modulesforhealthyenvironment.pdf && \
cd - make html \
&& opera build/html/index.html
```

1.3 Configuration file

The configuration file `config.ini` has to be placed in the `/etc/dp/` folder. The example of a configuration file follows.

```
[ant-work]
api.key.dm =

[rehivetech]
api.key.dm =
api.key.acontroller =

[db]
host =localhost
user =
passwd =
database =statistiky

[rapidminer]
launcher = ./rapidminer-studio/RapidMiner-Studio.sh
repository.processes.path =

[OpenWeatherOrg]
api.key =
```

(continues on next page)

(continued from previous page)

```
[WundergroundCom]
api.key =

[package]
abs.path = /home/travis/build/iotcloud

abs.graph.path = ${abs.path}/src/graph

; shift last value in PreProcessing, this value is last measured from server
shift.last_value = 300

co2.event_file.name = ${abs.path}/examples/events_peto.json
t_h.event_file.name = ${abs.path}/examples/events_klarka.json
shower.event_file.name = ${abs.path}/examples/events_klarka_shower.json

; time shift that is subtracted or added to start or end of an interval respectively
interval_extension = 300

[open-detector]
generic.directory = ${package:abs.path}/examples2/0300_open_detector/generic_data
generic.devices.path = ${package:abs.path}/examples2/0001_create_update_db/devices.
↪json

generic.co2.data_file.name = ${generic.directory}/training_co2.csv
generic.t_h.data_file.name = ${generic.directory}/training_t_h.csv

generic.co2.model.name = ${generic.directory}/model_co2.bin
generic.t_h.model.name = ${generic.directory}/model_t_h.bin

adapted.directory = ${package:abs.path}/examples2/0300_open_detector/adapted_data
adapted.combined_with_generic_data = No

; time shift to select an event when window was closed
attrs.no_event.time_shift = -1800

; time shift that is subtracted or added to start or end of an interval respectively
attrs.interval_extension = 300

selector.cache.before = 1800
selector.cache.after = 600

co2.detection.delay = 180
t_h.detection.delay = 180

[estimate]
; window size for linear regression
co2.windows.size = 1200
t_h.windows.size = 2700

; time shift that is subtracted or added to start or end of an interval respectively
attrs.interval_extension = 300

estimate.time = 3600

[predictor]
; time shift that is subtracted or added to start or end of an interval respectively
attrs.interval_extension = 300
```

(continues on next page)

(continued from previous page)

```
selector.cache.before = 3600
selector.cache.after = 7200

generic.directory = ${package:abs.path}/examples2/0302_ventilation_predictor/generic_
↪data
adapted.directory = ${package:abs.path}/examples2/0302_ventilation_predictor/adapted_
↪data

generic.co2.data_file.name = ${generic.directory}/training_co2.csv
generic.t_h.data_file.name = ${generic.directory}/training_t_h.csv

generic.co2.model.name = ${generic.directory}/model_co2.bin
generic.t_h.model.name = ${generic.directory}/model_t_h.bin

generic.t_h.raw_csv.training_data = ${generic.directory}/raw_training.csv

; time shift to select an event when window was closed
attrs.no_event.time_shift = -1800
```

1.4 Other tools

The following command can be used for API key generation on the server. It is only required to enter an arbitrary email in the right form.

```
echo `pwgen 80 1` "= namea@example.org" >> /etc/beeeon/server/apikeys.properties
```

CHAPTER

TWO

NOTIFICATIONS

The example of a notification follows:

```
{
    "data": {
        "estimate_open": false,
        "type": "co2_open",
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": false,
    "raise_catch": true,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

- **data** - the key `type` is mandatory, other keys (key `estimate_open` in this case) are optional according to a value of the key `type`,
  - **type** - a notification type,
- **device_id** - a unique identification of a device,
- **event** - a name of an end point to which a notification is sent,
- **gateway_id** - a unique identification of a gateway,
- **raise** - if a notification contains valid data,
- **raise_catch** - if an exception was thrown during the processing,
- **readable** - date and time when a notification was created,
- **timestamp** - timestamp when a notification was created.

CHAPTER

THREE

WINDOW OPENING DETECTOR

The aim is to design a system detecting window opening events on the basis of a carbon dioxide concentration or humidity change. The window opening detector according to a carbon dioxide concentration is based only on the course of indoor carbon dioxide concentration because indoor and outside relative humidity and temperature did not influence the success rate of a detector. The detector based on humidity considers also the course of temperature.

The configuration parameters in the examples below can be changed in the configuration file `config.ini` that is described in section *Configuration file*.

3.1 CO2

Today, people spend a considerable part of their life in office and living spaces. Therefore it is necessary to maintain healthy indoor environment. The concentration of carbon dioxide is an indicator of the quality of the indoor environment. Excessive concentration can cause fatigue, headache or sore throat. The long-term effect of high concentration of carbon dioxide on the human body can lead to asthma and other respiratory diseases. The concentration of carbon dioxide can be influenced by ventilation. If the ventilation is performed right, no health problems with the respiratory tract and with the concentration of people occur. Moreover, ventilation only for the necessary time contributes to saving energy.

3.1.1 Initialize general models from local database

A local database was created for debugging purposes. The database includes pre-processed data that enables fast computing of required attributes. The example of a model created using data stored in the database follows.

```
"""Initialize general models from local database.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

import os

from dm.models.ModelsUtil import ModelsUtil
from dm.models.open_detector.generic_training_file_from_local_db import *
from dm.ConnectionUtil import ConnectionUtil as cu


if __name__ == '__main__':
    cu.setup_logging()

    table_name = 'measured_filtered_peto'
```

(continues on next page)

(continued from previous page)

```
    columns = ColumnMapper.OPEN_CO2
    no_event_shift = int(cu.open_detector('attrs.no_event.time_shift'))

    directory = cu.open_detector('generic.directory')
    if not os.path.isdir(directory):
        os.mkdir(directory)

    co2_csv = cu.open_detector('generic.co2.data_file.name') + '_from_local_db.bin'
    co2_model = cu.open_detector('generic.co2.model.name') + '_from_local_db.bin'
    data_co2 = training_set_co2(cu.package('co2.event_file.name'), no_event_shift,␣
↪table_name,
                                co2_csv, columns)
    ModelsUtil.write_model(data_co2, co2_model, ModelsUtil.replace_nothing_open)
```

The dataset in the example is created using columns of the table `measured_filtered_peto`, the columns are stored in the variable `OPEN_CO2`. It is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. The variable `directory` that can be set using the parameter `generic.directory` denotes directory where a created model will be stored. A model is created using the function `training_set_co2` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data stored in the database is below.

```
python3 examples2/0300_open_detector/co2_init_general_models_from_local_db.py
```

3.1.2 Initialize general models from a remote server

The example of a model created using data saved on the server follows.

```
"""Initialize general models from a remote server.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

import json

from dm.models.open_detector.generic_training_file import generic_training_file
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.WundergroundCom import WundergroundCom
from dm.models.open_detector.create_attrs import ColumnMapper
from dm.models.ModelsUtil import ModelsUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    no_event_shift = int(cu.open_detector('attrs.no_event.time_shift'))

    with open(cu.open_detector('generic.devices.path'), 'r') as f:
        devs = json.load(f)

    w = WundergroundCom()
```

(continues on next page)

(continued from previous page)

```
    w.api_key = cu.wunderground_api_key()

    data_co2 = generic_training_file(cu.package('co2.event_file.name'),
                                     no_event_shift, 'co2', ColumnMapper.OPEN_CO2,␣
↪cls,
                                     devs, w)
    ModelsUtil.write_model(data_co2, cu.open_detector('generic.co2.model.name'),
                           ModelsUtil.replace_nothing_open)
```

At first, it is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Then it is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. Devices that were used to gather data are stored in the variable `devs`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. A model is created using the function `generic_training_file` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data saved on the remote server is below.

```
python examples2/0300_open_detector/co2_init_general_models.py
```

3.1.3 Initialize adapted model

The example of a model adapted to a given room follows.

```
"""Initialize adapted model.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

import time
import requests
import json

from dm.WundergroundCom import WundergroundCom
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.open_detector.adapted_models import prepare_adapted_data_co2


def prepare_adapted_co2(cls, actual_time):
    url_req = 'https://api.aurorahub.io/watchdogs-refresh'
    json_data = requests.get(url_req).text
    python_obj = json.loads(json_data)['co2_open_window']

    cl = cls['rehivetech']

    for item in python_obj:
        first_seen = None
        out_dev = []

        for dev in item['input']:
            res = cl.device_info(dev['gateway_id'], dev['device_id'])

            if first_seen is None:
```

(continues on next page)

(continued from previous page)

```
                first_seen = res['first_seen']

            if first_seen is not None:
                first_seen = max(first_seen, res['first_seen'])

            if 'open_close' in dev['modules']:
                out_dev.append(
                    {
                        'db_column_name': 'open_close',
                        'name': 'BeeeOn sensor',
                        'gateway_id': str(dev['gateway_id']),
                        'device_id': str(dev['device_id']),
                        'module_id': dev['modules'].index('open_close'),
                        'server_name': 'rehivetech',
                    }
                )

            if 'co2' in dev['modules']:
                out_dev.append(
                    {
                        'db_column_name': 'co2_in_ppm',
                        'name': 'BeeeOn sensor',
                        'gateway_id': str(dev['gateway_id']),
                        'device_id': str(dev['device_id']),
                        'module_id': dev['modules'].index('co2'),
                        'server_name': 'rehivetech',
                    }
                )

        coord = item['location']['coord']
        prepare_adapted_data_co2(out_dev, cls, first_seen, actual_time,
                                 coord['lat'], coord['lon'], w)


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(time.time())

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    prepare_adapted_co2(cls, actual_time)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. The current time is stored in the variable `actual_time`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The function `prepare_adapted_co2` creates a model using data gathered by devices up to the current time. The devices are stored in the variable `out_dev`. To create the model data from the general model can be also used. This option can be enabled or disabled using the parameter `adapted.combined_with_generic_data` in the configuration file `config.ini`. If the parameter is set to `Yes`, data from the general model will be used, if it is set to `No`, data will not be used. The resulting model is stored in the directory `adapted_data` by default.

The example of a command which can be used to adapt a model to a certain room is below.

```
python examples2/0300_open_detector/co2_init_adapted_model.py
```

3.1.4 Detector

The example of the use of a model for window opening detection follows.

```
"""Detector.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'co2_in_ppm',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 2,
            'server_name': 'ant-work',
        }
    ]

    notification = ModelsUtil.estimate_open_co2(devs, cls, lat, lon, actual_time, w)
    ModelsUtil.json_to_file(notification, 'co2_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Detection is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensor used to gather data is stored in the variable `devs`. Estimation if a window was open or not based on the concentration of carbon dioxide at the given time is performed by the function `estimate_open_co2` and the result is stored in the variable `notification`.

An output of a detection is a notification that contains basic information described in the section *Notifications* and information if a window was open or not which is stored in the key `estimate_open`.

```
{
    "data": {
        "estimate_open": false,
        "type": "co2_open"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": true,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

The example of a command which can be used to the use of a model for window opening detection is below.

```
python examples2/0300_open_detector/co2_detector.py
```

3.2 Temperature and humidity

People spend about 90 % of their life indoors. Therefore it is necessary to maintain a healthy indoor environment that is significantly affected by humidity. It is possible to determine the common symptoms of excessive humidity in buildings. For example, people can suffer from allergy or asthma thanks to airborne dust mites and mould spores that are spread in the excessively humid air. A big difference between indoor and outdoor temperature can cause condensation when water drops or fog can occur on the window glass. Mould spots and musty odours indicate the presence of mould and mildew that can result in health problems. A humid environment encourages mildew, mould and bacterial growth that negatively influence indoor air quality. Ventilation is able to remove pollutants and humidity forming indoors or reduce their concentrations to admissible levels for the occupant health and comfort. It should be energy efficient, preserve indoor air quality and it should not harm the occupants or the building.

3.2.1 Initialize general models from local database

A local database was created for debugging purposes. The database includes pre-processed data that enables fast computing of required attributes. The example of a model created using data stored in the database follows.

```
"""Initialize general models from local database.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

import os

from dm.models.ModelsUtil import ModelsUtil
from dm.models.open_detector.generic_training_file_from_local_db import *
from dm.ConnectionUtil import ConnectionUtil as cu


if __name__ == '__main__':
    cu.setup_logging()

    table_name = 'measured_klarka'
    columns = ColumnMapper.OPEN_T_H
```

(continues on next page)

(continued from previous page)

```
    no_event_shift = int(cu.open_detector('attrs.no_event.time_shift'))

    directory = cu.open_detector('generic.directory')
    if not os.path.isdir(directory):
        os.mkdir(directory)

    t_h_csv = cu.open_detector('generic.t_h.data_file.name') + '_from_local_db.bin'
    t_h_model = cu.open_detector('generic.t_h.model.name') + '_from_local_db.bin'
    data_t_h = training_set_t_h(cu.package('t_h.event_file.name'), no_event_shift,␣
↪table_name,
                                t_h_csv, columns)
    ModelsUtil.write_model(data_t_h, t_h_model, ModelsUtil.replace_nothing_open)
```

The dataset in the example is created using columns of the table `measured_filtered_klarka`, the columns are stored in the variable `OPEN_T_H`. It is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. The variable `directory` that can be set using the parameter `generic.directory` denotes directory where a created model will be stored. A model is created using the function `training_set_t_h` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data stored in the database is below.

```
python3 examples2/0300_open_detector/t_h_init_general_models_from_local_db.py
```

3.2.2 Initialize general models from a remote server

The example of a model created using data saved on the server follows.

```
"""Initialize general models from a remote server.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

import json
from dm.models.open_detector.generic_training_file import generic_training_file
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.WundergroundCom import WundergroundCom
from dm.models.open_detector.create_attrs import ColumnMapper
from dm.models.ModelsUtil import ModelsUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    no_event_shift = int(cu.open_detector('attrs.no_event.time_shift'))

    with open(cu.open_detector('generic.devices.path'), 'r') as f:
        devs = json.load(f)

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()
```

(continues on next page)

(continued from previous page)

```
    data_t_h = generic_training_file(cu.package('t_h.event_file.name'),
                                     no_event_shift, 't_h', ColumnMapper.OPEN_T_H,␣
↪cls,
                                     devs, w)
    ModelsUtil.write_model(data_t_h, cu.open_detector('generic.t_h.model.name'),
                           ModelsUtil.replace_nothing_open)
```

At first, it is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Then it is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. Devices that were used to gather data are stored in the variable `devs`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. A model is created using the function `generic_training_file` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function. The example of a command which can be used to create a model using data saved on the remote server is below.

```
python examples2/0300_open_detector/t_h_init_general_models.py
```

3.2.3 Initialize adapted model

The example of a model adapted to a given room follows.

```
"""Initialize adapted model.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.open_detector.adapted_models import prepare_adapted_data_t_h
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'open_close',
            'gateway_id': '1908402624139667',
            'device_id': '0x900000000197053',
            'module_id': 0,
            'server_name': 'ant-work',
        },
```

(continues on next page)

(continued from previous page)

```
        {
            'db_column_name': 'temperature_in2_celsius',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        },
        {
            'db_column_name': 'rh_in2_percentage',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 1,
            'server_name': 'ant-work',
        }
    ]

    prepare_adapted_data_t_h(devs, cls, actual_time - 24 * 60 * 60 * 15, actual_time,
                             lat, lon, w)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Adaptation is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. The function `prepare_adapted_data_t_h` creates a model using data gathered by devices up to the current time. To create the model data from the general model can also be used. This option can be enabled or disabled using the parameter `enable` or `disable`. The resulting model is stored in the directory `adapted_data` by default.

The example of a command which can be used to adapt a model to a certain room is below.

```
python examples2/0300_open_detector/t_h_init_adapted_model.py
```

3.2.4 Detector

The example of the use of a model for window opening detection follows.

```
"""Detector.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
```

(continues on next page)

(continued from previous page)

```
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'temperature_in2_celsius',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        },
        {
            'db_column_name': 'rh_in2_percentage',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 1,
            'server_name': 'ant-work',
        }
    ]

    notification = ModelsUtil.estimate_open_t_h(devs, cls, lat, lon, actual_time, w)
    ModelsUtil.json_to_file(notification, 't_h_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Detection is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Estimation if a window was open or not based on the humidity and the temperature at the given time is performed by the function `estimate_open_t_h` and the result is stored in the variable `notification`.

An output of a detection is a notification that contains basic information described in the section *Notifications* and information if a window was open or not which is stored in the key `estimate_open`.

```
{
    "data": {
        "estimate_open": false,
        "type": "t_h_open"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": true,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

The example of a command which can be used to the use of a model for window opening detection is below.

```
python examples2/0300_open_detector/t_h_detector.py
```

CHAPTER

FOUR

ESTIMATION OF A VALUE

A model is able to estimate a concentration of a carbon dioxide or humidity in certain time.

The configuration parameters in the examples below can be changed in the configuration file `config.ini` that is described in section *Configuration file*.

4.1 CO2

The estimation of a concentration of a carbon dioxide is based on a calculation of a line direction using several values that have been recently measured.

The example of an estimation of a carbon dioxide concentration in certain time follows.

```
"""Estimation of a value - CO2.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())

    devs = [
        {
            'db_column_name': 'co2_in_ppm',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 2,
            'server_name': 'ant-work',
        }
    ]

    notification = ModelsUtil.estimate_co2(devs, cls, actual_time)
    ModelsUtil.json_to_file(notification, 'co2_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Estimation is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. The sensor used to gather data is stored in the variable `devs`. Estimation of a carbon dioxide concentration in certain time is performed by the function `estimate_co2` and the result is stored in the variable `notification`. All configuration parameters are saved in the configuration file `config.ini` that is described in the section *Configuration file*.

An output of the estimation is a notification that contains basic information described in the section *Notifications* and the following information:

- **current_value** - current value of CO2 [ppm],
- **es_level** - a value of a carbon dioxide concentration that will be probably reached in `es_time` [ppm],
- **es_time** - a time when an estimation of a carbon dioxide concentration will be performed [s],
- **estimate_time** - a time when an upper limit of a given level of the concentration will be reached [min],
- **level** - a level of the concentration whose upper limit will be used to compute the `es_time`,
  - **0** – 0 - 499 - decreased concentration,
  - **1** – 500 - 1499 - optimal concentration,
  - **2** – 1500 - 2499 - increased concentration,
  - **3** – 2500 - 3999 - high concentration,
  - **4** – 4000 - too high concentration,
- **level_value** - a value of `level` [ppm],
- **type** - a notification type.

The example of a notification follows.

```
{
    "data": {
        "current_value": 2000,
        "es_level": 2000,
        "es_time": 60,
        "estimate_time": 0,
        "level": 2,
        "level_value": 2500,
        "type": "co2_estimate"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": false,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

The example of a command that can be used to estimate a carbon dioxide concentration in certain time is below.

```
python examples2/0301_estimate/co2_estimate.py
```

4.2 Temperature and humidity

The estimation of humidity is based on a calculation of a line direction using several values that have been recently measured.

The example of an estimation of humidity in certain time follows.

```
"""Estimation of a value - temperature and humidity.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())

    devs = [
        {
            'db_column_name': 'temperature_in2_celsius',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        },
        {
            'db_column_name': 'rh_in2_percentage',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 1,
            'server_name': 'ant-work',
        }
    ]

    notification = ModelsUtil.estimate_t_h(devs, cls, actual_time)
    ModelsUtil.json_to_file(notification, 't_h_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Estimation is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. The sensors used to gather data are stored in the variable `devs`. Estimation of humidity in certain time is performed by the function `estimate_t_h` and the result is stored in the variable `notification`. All configuration parameters are saved in the configuration file `config.ini` that is described in the section *Configuration file*.

An output of the estimation is a notification that contains basic information described in the section *Notifications* and the following information:

- **current_value** - current value of humidity [%],
- **es_level** - a value of humidity that will be probably reached in `es_time` [%],

- **es_time** - a time when an estimation of humidity will be performed [s],
- **estimate_time** - a time when an upper limit of a given level of the humidity will be reached [min],
- **level** - a level of the humidity whose upper limit will be used to compute the `es_time`,
  - **0** – 0% - 29% - too low humidity,
  - **1** – 30% - 39% - decreased humidity,
  - **2** – 40% - 59% - optimal humidity,
  - **3** – 60% - 69% - increased humidity,
  - **4** – 70% - 100% - too high humidity,
- **level_value** - a value of `level` [%],
- **type** - a notification type.

The example of a notification follows.

```
{
    "data": {
        "current_value": 41,
        "es_level": 41,
        "es_time": 60,
        "estimate_time": 796,
        "level": 2,
        "level_value": 60,
        "type": "co2_estimate"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": false,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

The example of a command that can be used to estimate humidity in certain time is below.

```
python examples2/0301_estimate/t_h_estimate.py
```

CHAPTER

FIVE

VENTILATION LENGTH PREDICTOR

The aim is to define a proper way of air change rate regulation. The regulation is ensured by natural ventilation that is energy efficient, requires little maintenance, has low initial costs, and is environmentally friendly. Natural ventilation should be used wherever and whenever is possible apart from areas where the quality of the air outside the house is worse than indoor air quality. A predictor enables to predict how long ventilation should be performed to decrease a given quantity to the required value.

The configuration parameters in the examples below can be changed in the configuration file `config.ini` that is described in section *Configuration file*.

5.1 CO2

The predictor is based on the equation which can be used to calculate a decrease of a carbon dioxide concentration during a time interval.

5.1.1 Initialize general models from local database

A local database was created for debugging purposes. The database includes pre-processed data that enables fast computing of required attributes. The example of a model created using data stored in the database follows.

```
"""Initialize general models from local database.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.models.ModelsUtil import ModelsUtil
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.predictor.generic_training_file_from_local_db import training_file_co2


if __name__ == '__main__':
    cu.setup_logging()

    no_event_shift = int(cu.predictor('attrs.no_event.time_shift'))

    data_co2 = training_file_co2(cu.package('co2.event_file.name'), no_event_shift)
    co2_filename = cu.predictor('generic.co2.model.name') + '_from_local_db.bin'
    ModelsUtil.write_model(data_co2, co2_filename, replace=ModelsUtil.replace_co2_
↪ventilation_len)
```

It is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. A model is created using the function `training_file_co2` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data stored in the database is below.

```
python examples2/0302_ventilation_predictor/co2_init_general_models_from_local_db.py
```

5.1.2 Initialize general models from a remote server

The example of a model created using data saved on the server follows.

```
"""Initialize general models from a remote server.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.models.ModelsUtil import ModelsUtil
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.predictor.generic_training_file import training_file_co2
from dm.WundergroundCom import WundergroundCom
import json


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    no_event_shift = int(cu.open_detector('attrs.no_event.time_shift'))

    # file with devices for training
    with open(cu.open_detector('generic.devices.path'), 'r') as f:
        devs = json.load(f)

    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    data_co2 = training_file_co2(cu.package('co2.event_file.name'), no_event_shift,␣
↪cls, devs,
                                 lat, lon, w)
    co2_filename = cu.predictor('generic.co2.model.name')
    ModelsUtil.write_model(data_co2, co2_filename, replace=ModelsUtil.replace_co2_
↪ventilation_len)
```

At first, it is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Then it is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. Devices that were used to gather data are stored in the variable `devs`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the

function `wunderground_api_key`. A model is created using the function `training_file_co2` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data saved on the remote server is below.

```
python examples2/0302_ventilation_predictor/co2_init_general_models.py
```

5.1.3 Predictor

The example of the model usage for predicting how long ventilation should be performed follows.

```
"""Predictor.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'co2_in_ppm',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 2,
            'server_name': 'ant-work',
        },
        {
            "db_column_name": "temperature_in_celsius",
            "gateway_id": "1816820318180747",
            "device_id": "0xa900811026800001",
            "module_id": 0,
            "server_name": "ant-work"
        },
        {
            "db_column_name": "rh_in_percentage",
            "gateway_id": "1816820318180747",
            "device_id": "0xa900811026800001",
            "module_id": 1,
            "server_name": "ant-work"
        }
```

(continues on next page)

(continued from previous page)

```
    ]

    notification = ModelsUtil.predictor(devs, cls, lat, lon, actual_time, 'co2', w)
    ModelsUtil.json_to_file(notification, 'co2_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Prediction is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Prediction how long ventilation should be performed to decrease the concentration of carbon dioxide to a given value is performed by the function `predictor` and the result is stored in the variable `notification`.

An output of the prediction is a notification that contains basic information described in the section *Notifications* and the following information:

- **current_value** - current value of CO2 [ppm],
- **estimate_time** - a time when a carbon dioxide concentration will reach a given value [min],
- **final_value** - value of CO2 that should be reached after ventilation [ppm],
- **type** - what quantity is used for prediction (carbon dioxide).

The example of a notification follows.


```
{
    "data": {
        "current_value": 2000,
        "estimate_time": 9,
        "final_value": 400,
        "type": "co2_predictor"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": true,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```


The example of a command which can be used to the use of a model for prediction how long ventilation should be performed is below.

```
python examples2/0302_ventilation_predictor/co2_predictor.py
```

5.2 Temperature and humidity

The model for prediction is based on a decrease in humidity during ventilation for the selected time intervals. It was necessary to consider specific humidity that can be easily converted to relative humidity.

5.2.1 Initialize general models from local database

A local database was created for debugging purposes. The database includes pre-processed data that enables fast computing of required attributes. The example of a model created using data stored in the database follows.

```
"""Initialize general models from local database.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.models.ModelsUtil import ModelsUtil
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.predictor.generic_training_file_from_local_db import training_file_t_h


if __name__ == '__main__':
    cu.setup_logging()

    no_event_shift = int(cu.predictor('attrs.no_event.time_shift'))

    data_t_h = training_file_t_h(cu.package('t_h.event_file.name'), no_event_shift)
    t_h_filename = cu.predictor('generic.t_h.model.name') + '_from_local_db.bin'
    ModelsUtil.write_model(data_t_h, t_h_filename, replace=ModelsUtil.replace_
↪ventilation_length)
```

It is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. A model is created using the function `training_file_t_h` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data stored in the database is below.

```
python examples2/0302_ventilation_predictor/t_h_init_general_models_from_local_db.py
```

5.2.2 Initialize general models from a remote server

The example of a model created using data saved on the server follows.

```
"""Initialize general models from a remote server.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

import json

from dm.models.ModelsUtil import ModelsUtil
from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.predictor.generic_training_file import training_file_t_h
from dm.WundergroundCom import WundergroundCom


if __name__ == '__main__':
    cu.setup_logging()
```

(continues on next page)

(continued from previous page)

```
    cls = cu.setup_clients()

    no_event_shift = int(cu.open_detector('attrs.no_event.time_shift'))

    # file with devices for training
    with open(cu.open_detector('generic.devices.path'), 'r') as f:
        devs = json.load(f)

    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    data_t_h = training_file_t_h(cu.package('t_h.event_file.name'), no_event_shift,␣
↪cls, devs,
                                 lat, lon, w)
    t_h_filename = cu.predictor('generic.t_h.model.name')
    ModelsUtil.write_model(data_t_h, t_h_filename, replace=ModelsUtil.replace_
↪ventilation_length)
```

At first, it is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Then it is required to create a dataset that contains the same number of events when a window was open and when it was closed. To select the events when the window was closed the variable `no_event_shift` is defined. Its value can be set using the parameter `attrs.no_event.time_shift`. Devices that were used to gather data are stored in the variable `devs`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. A model is created using the function `training_file_t_h` that uses `SVM` classifier (`sklearn` library). The created model is written to a binary file using the `write_model` function.

The example of a command which can be used to create a model using data saved on the remote server is below.

```
python examples2/0302_ventilation_predictor/t_h_init_general_models.py
```

5.2.3 Predictor

The example of the model usage for predicting how long ventilation should be performed follows.

```
"""Predictor.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
```

(continues on next page)

(continued from previous page)

```
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'temperature_in2_celsius',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        },
        {
            'db_column_name': 'rh_in2_percentage',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 1,
            'server_name': 'ant-work',
        }
    ]

    notification = ModelsUtil.predictor(devs, cls, lat, lon, actual_time, 't_h', w,␣
↪40)
    ModelsUtil.json_to_file(notification, 't_h_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Prediction is performed on the basis of data measured during a given time interval. Time that defines end of a time interval is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Prediction how long ventilation should be performed to decrease humidity to a given value is performed by the function `predictor` and the result is stored in the variable `notification`.

An output of the prediction is a notification that contains basic information described in the section *Notifications* and the following information:

- **current_value** - current value of humidity [%],
- **estimate_time** - a time when humidity will reach a given value [min],
- **final_value** - value of humidity that should be reached after ventilation [%],
- **type** - what quantity is used for prediction (humidity).

The example of a notification follows.

```
{
    "data": {
        "current_value": 41,
        "estimate_time": 10,
        "final_value": 40,
        "type": "t_h_predictor"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
```

(continues on next page)

(continued from previous page)


```
    "gateway_id": "1816820318180747",
    "raise": true,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```


The example of a command which can be used to the use of a model for prediction how long ventilation should be performed is below.

```
python examples2/0302_ventilation_predictor/t_h_predictor.py
```

CHAPTER

SIX

WHEN TO VENTILATE IN SUMMER

The goal is to determine if the ventilation is useful to decrease the temperature in a room in given time in summer.

The configuration parameters in the examples below can be changed in the configuration file `config.ini` that is described in section *Configuration file*.

```
"""When to ventilate in summer.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'temperature_in_celsius',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        },
        {
            'db_column_name': 'rh_in_percentage',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 1,
            'server_name': 'ant-work',
        }
```

(continues on next page)

(continued from previous page)

```
    ]

    notification = ModelsUtil.when_ventilate_summer(devs, cls, lat, lon, actual_time,␣
↪w, temperature_diff=4)
    ModelsUtil.json_to_file(notification, 'notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Current time is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data and it is obtained by the function `wunderground_api_key`. The sensor used to gather data is stored in the variable `devs`. The determination if the ventilation is useful in current time is performed by the function `when_ventilate_summer` and the result is stored in the variable `notification`.

An output of the determination is a notification that contains basic information described in the section *Notifications* and the following information:

- **temp_in** - indoor temperature,
- **temp_out** - outside temperature,
- **type** - what quantity is used for prediction,
- **ventilate** - if the ventilation is useful,
- **0** - ventilation is useless, the outside temperature is higher than the indoor temperature,
- **1** - ventilation will be probably useless, the outside temperature is similar to indoor temperature,
- **2** - ventilation is useful, the outside temperature is lower than the indoor temperature.

The example of a notification follows.


```
{
    "data": {
        "temp_in": 26.0,
        "temp_out": 3.0,
        "type": "when_ventilate_summer",
        "ventilate": 2
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": true,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```


The example of a command that can be used to determine if ventilation is useful to decrease the temperature in a room in the summer months is below.

```
python examples2/0303_when_to_ventilate/run.py
```

CHAPTER

SEVEN

DEW POINT

The aim is to determine if condensation will occur. Temperature and humidity are used to calculate a dew point. If temperature is lower than the dew point, condensation will appear.

As water vapour concentration grows, the rate of condensation increases. The amount of water grows until the rate of condensation and the rate of evaporation are in equilibrium. Then the water vapour concentration will stop increasing because the air is saturated with water vapour. Relative humidity of saturated air is 100 %. The temperature to which the air has to be cooled to reach saturation is called dew point.

The configuration parameters in the examples below can be changed in the configuration file `config.ini` that is described in section *Configuration file*.

```
"""Dew point.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'temperature_in_celsius',
            'name': 'BeeeOn sensor',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        },
```

(continues on next page)

(continued from previous page)

```
        {
            'db_column_name': 'rh_in_percentage',
            'name': 'BeeeOn sensor',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 1,
            'server_name': 'ant-work',
        }
    ]

    notification = ModelsUtil.dew_point(devs, cls, lat, lon, actual_time, w)
    ModelsUtil.json_to_file(notification, 'notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Current time is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Determination if condensation will occur is performed by the function `dew_point` and the result is stored in the variable `notification`.

An output of the determination is a notification that contains basic information described in the section *Notifications* and the following information:

- **dew_point** - dew point calculated using indoor temperature and humidity,
- **dewing** - if condensation will occur,
- **hum_in** - indoor humidity,
- **temp_in** - indoor temperature,
- **type** - what quantity is used for prediction.

The example of a notification follows.

```
{
    "data": {
        "dew_point": 11.75,
        "dewing": null,
        "hum_in": 41.0,
        "temp_in": 26.0,
        "type": "dew_point"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": true,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

The example of a command that can be used to determine if condensation will occur is below.

```
python examples2/0304_dew_point/run.py
```

CHAPTER

EIGHT

ANOMALY DETECTION

The goal is to detect anomalies on the basis of given difference of quantity values measured in two different time points. One time point is current time and the second one is selected from data that has been recently measured.

The configuration parameters in the examples below can be changed in the configuration file `config.ini` that is described in section *Configuration file*.

8.1 CO2

Detection of anomalies is based on difference in carbon dioxide concentration values measured in two various time points.

```
"""Anomaly detection - CO2.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'co2_in_ppm',
            'name': '',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 2,
```

(continues on next page)

(continued from previous page)

```
            'server_name': 'ant-work',
        }
    ]
    notification = ModelsUtil.anomaly_diff(devs, cls, lat, lon, actual_time, w, 2,␣
↪7200, 'co2')
    ModelsUtil.json_to_file(notification, 'co2_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Current time is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Anomaly detection is performed by the function `anomaly_diff` and the result is stored in the variable `notification`.

An output of the detection is a notification that contains basic information described in the section *Notifications* and the following information:

- **actual_diff** - current difference in carbon dioxide concentration values,
- **anomaly** - if anomaly is detected,
- **min_anomaly_diff** - minimal difference in carbon dioxide concentration values to detect anomaly,
- **min_anomaly_time** - time interval between two time points,
- **type** - what quantity is used for prediction.

The example of a notification follows.


```
{
    "data": {
        "actual_diff": 0.0,
        "anomaly": false,
        "min_anomaly_diff": 2,
        "min_anomaly_time": 120,
        "type": "anomaly_co2"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": false,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```


The example of a command that can be used to detect anomalies on the basis of carbon dioxide concentration difference.

```
python examples2/0305_anomaly_detection/co2_run.py
```

8.2 Temperature

Detection of anomalies is based on difference in temperature values measured in two various time points.

```
"""Anomaly detection - temperature.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'temperature_in_celsius',
            'name': 'BeeeOn sensor',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
            'module_id': 0,
            'server_name': 'ant-work',
        }
    ]
    notification = ModelsUtil.anomaly_diff(devs, cls, lat, lon, actual_time, w, 2,␣
↪7200, 'temperature')
    ModelsUtil.json_to_file(notification, 't_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Current time is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Anomaly detection is performed by the function `anomaly_diff` and the result is stored in the variable `notification`.

An output of the detection is a notification that contains basic information described in the section *Notifications* and the following information:

- **actual_diff** - current difference in temperature values,
- **anomaly** - if anomaly is detected,
- **min_anomaly_diff** - minimal difference in temperature values to detect anomaly,
- **min_anomaly_time** - time interval between two time points,
- **type** - what quantity is used for prediction.

The example of a notification follows.

```
{
    "data": {
        "actual_diff": 0.22,
        "anomaly": false,
        "min_anomaly_diff": 2,
        "min_anomaly_time": 120,
        "type": "anomaly_temperature"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": false,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```

The example of a command that can be used to detect anomalies on the basis of temperature difference.

```
python examples2/0305_anomaly_detection/t_run.py
```

8.3 Humidity

Detection of anomalies is based on difference in humidity values measured in two various time points.

```
"""Anomaly detection - humidity.
"""
from os.path import dirname, abspath, join
import sys
sys.path.append(abspath(join(dirname(__file__), '../..', '')))

from dm.ConnectionUtil import ConnectionUtil as cu
from dm.models.ModelsUtil import ModelsUtil
from dm.WundergroundCom import WundergroundCom
from dm.DateTimeUtil import DateTimeUtil


if __name__ == '__main__':
    cu.setup_logging()
    cls = cu.setup_clients()

    actual_time = int(DateTimeUtil.local_time_str_to_utc('2019/02/20 03:00:00').
↪timestamp())
    lat = 49.1649894
    lon = 16.562262499999974

    w = WundergroundCom()
    w.api_key = cu.wunderground_api_key()

    devs = [
        {
            'db_column_name': 'rh_in_percentage',
            'name': 'BeeeOn sensor',
            'gateway_id': '1816820318180747',
            'device_id': '0xa900811026800001',
```

(continues on next page)

(continued from previous page)

```
            'module_id': 1,
            'server_name': 'ant-work',
        }
    ]
    notification = ModelsUtil.anomaly_diff(devs, cls, lat, lon, actual_time, w, 2,␣
↪7200, 'humidity')
    ModelsUtil.json_to_file(notification, 'h_notification.doc.json', log_
↪notification=True)
```

It is necessary to set clients that can communicate with a remote server using the function `setup_clients`. Current time is stored in the variable `actual_time`. Latitude and longitude that enable to get weather data according to a sensor position are stored in the variable `lat` and `lon`. The API key allows to get weather data from server and it is obtained by the function `wunderground_api_key`. The sensors used to gather data are stored in the variable `devs`. Anomaly detection is performed by the function `anomaly_diff` and the result is stored in the variable `notification`.

An output of the detection is a notification that contains basic information described in the section *Notifications* and the following information:

- **actual_diff** - current difference in humidity values,
- **anomaly** - if anomaly is detected,
- **min_anomaly_diff** - minimal difference in humidity values to detect anomaly,
- **min_anomaly_time** - time interval between two time points,
- **type** - what quantity is used for prediction.

The example of a notification follows.


```
{
    "data": {
        "actual_diff": 1.5,
        "anomaly": false,
        "min_anomaly_diff": 2,
        "min_anomaly_time": 120,
        "type": "anomaly_humidity"
    },
    "device_id": "0xa900811026800001",
    "event": "env-notification-pre",
    "gateway_id": "1816820318180747",
    "raise": false,
    "raise_catch": false,
    "readable": "2019-02-20 03:00:00",
    "timestamp": 1550628000
}
```


The example of a command that can be used to detect anomalies on the basis of humidity difference.

```
python examples2/0305_anomaly_detection/h_run.py
```

CHAPTER

NINE

DM PACKAGE

9.1 Subpackages

9.1.1 dm.attrs package

Submodules

dm.attrs.AbstractPrepareAttr module

Calculates number of positive differences in given time points, geometric mean, arithmetic mean, variance and standard deviation of differences.

class dm.attrs.AbstractPrepareAttr.**AbstractPrepareAttr**(*row_selector*, *interval_selector*, *tr=None*)

Bases: abc.ABC

arithmetic_mean(*new_column_name*, *precision*, *values_before*, *values_after*, *prefix*)

It computes arithmetic mean from given values before and after event.

Parameters

- **new_column_name** – name of attribute
- **precision** – precision of calculation
- **values_before** – list of values before event
- **values_after** – list of values after event
- **prefix** – prefix of attribute name

Returns pair of lists, each list contains pairs of attribute name and arithmetic mean

attr_name(*column_name*, *prefix*, *interval_type*, *interval*)

It generates a name of attribute.

Parameters

- **column_name** – name of attribute
- **prefix** – prefix of attribute name
- **interval_type** – type of interval - before or after
- **interval** – time interval from which the attribute is calculated

Returns name of attribute

abstract execute(***kwargs*)
It ensures calculation of the required attribute.

Parameters **kwargs** –

Returns pair of lists, each list contains pair of attribute name and its value

geometric_mean(*new_column_name*, *precision*, *values_before*, *values_after*, *prefix*)
It computes geometric mean from given values before and after event.

Parameters

- **new_column_name** – name of attribute
- **precision** – precision of calculation
- **values_before** – list of values before event
- **values_after** – list of values after event
- **prefix** – prefix of attribute name

Returns pair of lists, each list contains pairs of attribute name and geometric mean

standard_deviation(*new_column_name*, *precision*, *values_before*, *values_after*, *prefix*)
It computes standard deviation from given values before and after event.

Parameters

- **new_column_name** – name of column
- **precision** – precision of calculation
- **values_before** – list of values before event
- **values_after** – list of values after event
- **prefix** – prefix of attribute name

Returns pair of lists, each list contains pairs of attribute name and standard deviation

variance(*new_column_name*, *precision*, *values_before*, *values_after*, *prefix*)
It computes variance from given values before and after event.

Parameters

- **new_column_name** – name of column
- **precision** – precision of calculation
- **values_before** – list of values before event
- **values_after** – list of values after event
- **prefix** – prefix of attribute name

Returns pair of lists, each list contains pairs of attribute name and variance

dm.attrs.CO2VentilationLength module

Gets current value CO2 and measured value CO2 in a given time point.

class dm.attrs.CO2VentilationLength.**CO2VentilationLength**(*row_selector*, *interval_selector*, *tr=None*)
Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp_start*, *timestamp_end*, *compute_timestamp*, *intervals*, *method*, *co2_out*, *column*, *precision*, *prefix*, *enable_actual_value=True*)

It computes how long ventilation was performed to decrease a CO2 concentration.

Parameters

- **timestamp_start** – timestamp when ventilation started
- **timestamp_end** – timestamp when ventilation finished
- **compute_timestamp** – selected timestamp during ventilation
- **intervals** – deprecated
- **method** – deprecated
- **co2_out** – concentration of outside CO2
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **prefix** – prefix of attribute name
- **enable_actual_value** – if the actual value should be included in output

Returns pair of lists, the first list contains at most two pairs, the first one (optional) includes attribute name and actual value, the second one includes attribute name and ventilation length

dm.attrs.DiffInLinear module

Calculates difference between quantity values after linearization (selects linearized values at the moment of window opening and closing).

class dm.attrs.DiffInLinear.**DiffInLinear**(*row_selector*, *interval_selector*, *tr=None*)

Bases: *dm.attrs.InLinear.InLinear*

execute(*timestamp_before*, *timestamp_after*, *column*, *precision*, *start_before*, *end_before*, *start_after*, *end_after*, *prefix*, *new_column_name*)

It computes difference between quantity values after linearization.

After linearization time points at the moment of window opening and window closing are selected to compute difference.

Parameters

- **timestamp_before** – timestamp selected from linearized time interval before event
- **timestamp_after** – timestamp selected from linearized time interval after event
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **start_before** – time shift before event that denotes start of time interval that is linearised (in seconds)
- **end_before** – time shift before event that denotes end of time interval that is linearised (in seconds)
- **start_after** – time shift after event that denotes start of time interval that is linearised (in seconds)
- **end_after** – time shift after event that denotes end of time interval that is linearised (in seconds)

- **prefix** – prefix of attribute name
- **new_column_name** – name of attribute

Returns pair of lists, the first list contains pair including attribute name and difference between quantity values after linearization

dm.attrs.DifferenceBetweenRealLinear module

Calculates differences between real and linearized values of quantity in given time points.

class dm.attrs.DifferenceBetweenRealLinear.**DifferenceBetweenRealLinear**(*row_selector*, *interval_selector*, *tr=None*)

Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp*, *column*, *precision*, *intervals_before*, *intervals_after*, *window_size_before*, *window_size_after*, *prefix*, *new_column_name*)

It computes difference between real and linearised course of given quantity.

Parameters

- **timestamp** – timestamp which is in the middle of time interval used for calculation
- **column** – name of column that contains real values
- **precision** – precision of calculation
- **intervals_before** – list of time points before event
- **intervals_after** – list of time points after event
- **window_size_before** – time interval before event used to calculate linearised course
- **window_size_after** – time interval after event used to calculate linearised course
- **prefix** – prefix of attribute name
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and difference between real and linearised course of given quantity

dm.attrs.FirstDifferenceAttrA module

Calculates first differences using quantity values (not only successive values).

class dm.attrs.FirstDifferenceAttrA.**FirstDifferenceAttrA**(*row_selector*, *interval_selector*, *tr=None*)

Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp*, *column*, *precision*, *intervals_before*, *intervals_after*, *normalize*, *enable_count*, *prefix*, *selected_before*, *selected_after*, *new_column_name*)

It computes the first differences using quantity values.

Not only successive values are used for computation.

Parameters

- **timestamp** – timestamp when event occurred
- **column** – name of column that contains required values

- **precision** – precision of calculation
- **intervals_before** – list of time points before event
- **intervals_after** – list of time points after event
- **normalize** – if the computed should be normalized
- **enable_count** – if number of positives values should be computed
- **prefix** – prefix of attribute name
- **selected_before** – list of lists of selected time points before event
- **selected_after** – list of lists of selected time points after event
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and the first difference

dm.attrs.FirstDifferenceAttrB module

Calculates first differences using quantity values (only successive values).

class dm.attrs.FirstDifferenceAttrB.**FirstDifferenceAttrB**(*row_selector*, *interval_selector*, *tr=None*)

Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp*, *column*, *precision*, *intervals_before*, *intervals_after*, *normalize*, *enable_count*, *prefix*, *selected_before*, *selected_after*, *new_column_name*)

It computes the first differences using quantity values.

Only successive values are used for computation.

Parameters

- **timestamp** – timestamp when event occurred
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **intervals_before** – list of time points before event
- **intervals_after** – list of time points after event
- **normalize** – if the computed should be normalized
- **enable_count** – if number of positives values should be computed
- **prefix** – prefix of attribute name
- **selected_before** – list of lists of selected time points before event
- **selected_after** – list of lists of selected time points after event
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and the first difference

dm.attrs.GrowthRate module

Calculates growth rates.

The growth rate is calculated as y_t/y_t_-1, where y_t is selected based on forward/backward shift and y_t_-1 is calculated as y_t - value_delay.

class dm.attrs.GrowthRate.**GrowthRate**(*row_selector*, *interval_selector*, *tr=None*)
Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp*, *column*, *precision*, *intervals_before*, *intervals_after*, *value_delay*, *prefix*, *new_column_name*)
It computes growth rates.

Parameters

- **timestamp** – timestamp when event occurred
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **intervals_before** – list of time points before event
- **intervals_after** – list of time points after event
- **value_delay** – time shift used to get a value marked as y_t_-1
- **prefix** – prefix of attribute name
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and growth rate

dm.attrs.InLinear module

Calculates quantity values after linearization.

class dm.attrs.InLinear.**InLinear**(*row_selector*, *interval_selector*, *tr=None*)
Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp_before*, *timestamp_after*, *column*, *precision*, *start_before*, *end_before*, *start_after*, *end_after*, *prefix*, *new_column_name*)
It computes quantity values after linearization.

Parameters

- **timestamp_before** – timestamp selected from linearized time interval before event
- **timestamp_after** – timestamp selected from linearized time interval after event
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **start_before** – time shift before event that denotes start of time interval that is linearised (in seconds)
- **end_before** – time shift before event that denotes end of time interval that is linearised (in seconds)
- **start_after** – time shift after event that denotes start of time interval that is linearised (in seconds)
- **end_after** – time shift after event that denotes end of time interval that is linearised (in seconds)
- **prefix** – prefix of attribute name
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and value after linearization

dm.attrs.InOutDiff module

Calculates differences between quantity values measured indoor and outdoor.

class dm.attrs.InOutDiff.**InOutDiff**(*row_selector*, *interval_selector*, *tr=None*)
Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp*, *column*, *precision*, *intervals_before*, *intervals_after*, *prefix*, *new_column_name*)
It computes differences between quantity values measured indoor and outdoor.

Parameters

- **timestamp** – timestamp when event occurred
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **intervals_before** – list of time points before event
- **intervals_after** – list of time points after event
- **prefix** – prefix of attribute name
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and geometric mean

dm.attrs.Regression module

Calculates exponential regression from given CO2 values.

class dm.attrs.Regression.**Regression**(*row_selector*, *interval_selector*, *method*)
Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*timestamp_start*, *timestamp_end*, *column*, *precision*, *prefix*, *enable_error*, *new_column_name*)
It computes exponential regression from given CO2 values.

Parameters

- **timestamp_start** – timestamp that denotes the start of time interval from which the regression is computed
- **timestamp_end** – timestamp that denotes the end of time interval from which the regression is computed
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **prefix** – prefix of attribute name
- **enable_error** – if an error of exponential regression is in the output
- **new_column_name** – name of attribute

Returns pair of lists, the first list contains pair of attribute name and exponential regression from given CO2 values and pair of attribute name and error of exponential regression (optional)

static gen_f_lambda(*co2_start*, *co2_out*)

static gen_f_prietok(*co2_start*, *co2_out*, *volume*)

dm.attrs.SecondDifferenceAttr module

Calculates second differences using quantity values.

class dm.attrs.SecondDifferenceAttr.**SecondDifferenceAttr**(*row_selector*, *interval_selector*, *tr=None*)

Bases: *dm.attrs.FirstDifferenceAttrB.FirstDifferenceAttrB*

execute(*timestamp*, *column*, *precision*, *intervals_before*, *intervals_after*, *normalize*, *enable_count*, *prefix*, *selected_before*, *selected_after*, *new_column_name*)

It computes the second differences using quantity values.

Parameters

- **timestamp** – timestamp when event occurred
- **column** – name of column that contains required values
- **precision** – precision of calculation
- **intervals_before** – list of time points before event
- **intervals_after** – list of time points after event
- **normalize** – if the computed should be normalized
- **enable_count** – if number of positives values should be computed
- **prefix** – prefix of attribute name
- **selected_before** – list of lists of selected time points before event
- **selected_after** – list of lists of selected time points after event
- **new_column_name** – name of attribute

Returns pair of lists, each list contains pairs of attribute name and the second difference

dm.attrs.VentilationLength module

Assigns given ventilation length to a class.

class dm.attrs.VentilationLength.**VentilationLength**(*row_selector*, *interval_selector*, *tr=None*)

Bases: *dm.attrs.AbstractPrepareAttr.AbstractPrepareAttr*

execute(*event_start*, *event_end*, *intervals*, *threshold*, *prefix*)

It assigns given ventilation length to a class.

Parameters

- **event_start** – timestamp that denotes the start of the event
- **event_end** – timestamp that denotes the end of the event
- **intervals** – intervals representing classes to which the events can be assigned
- **threshold** – threshold used in evaluation to which class the event should be assigned
- **prefix** – prefix of attribute name

Returns pair of lists, each list contains pairs of attribute name and assigned interval (class)

Module contents

9.1.2 dm.co2regression package

Submodules

dm.co2regression.AbstractRegression module

Abstract class for calculation of CO2 regression using various methods.

class dm.co2regression.AbstractRegression.**AbstractRegression**(*co2_out*)
Bases: abc.ABC

abstract compute_curve(*x*, *y*)
It computes values after exponential regression in given time points.

Parameters

- **x** – list of time points
- **y** – list of CO2 values in given time points

Returns list of values after exponential regression calculated in given time points

abstract compute_parameter(*x*, *y*)
It computes air change rate and its error.

Parameters

- **x** – list of time points
- **y** – list of CO2 values in given time points

Returns pair of air change rate and its error

dm.co2regression.ExpRegressionWithDelay module

Calculation of CO2 regression using outdoor CO2, room volume, window_size and threshold.

Calculates regression when the CO2 decrease is noticed using threshold in window.

class dm.co2regression.ExpRegressionWithDelay.**ExpRegressionWithDelay**(*co2_out*, *volume*, *window_size*, *threshold*)
Bases: *dm.co2regression.SimpleExpRegression.SimpleExpRegression*

compute_curve(*x*, *y*)
It computes values after exponential regression in given time points.

Parameters

- **x** – list of time points
- **y** – list of CO2 values in given time points

Returns list of values after exponential regression calculated in given time points

compute_parameter(*x*, *y*)
It computes air change rate and its error.

The computation starts if a change in CO2 is higher than the threshold.

Parameters

- **x** – list of time points
- **y** – list of CO2 values in given time points

Returns pair of air change rate and its error

dm.co2regression.SimpleExpRegression module

Calculation of CO2 regression using outdoor CO2, room volume.

Calculates regression when the window is open.

class dm.co2regression.SimpleExpRegression.**SimpleExpRegression**(*co2_out*, *volume*)
Bases: *dm.co2regression.AbstractRegression.AbstractRegression*

compute_curve(*x*, *y*)
It computes values after exponential regression in given time points.

Parameters

- **x** – list of time points
- **y** – list of CO2 values in given time points

Returns list of values after exponential regression calculated in given time points

compute_parameter(*x*, *y*)
It computes air change rate and its error.

The computation requires room volume.

Parameters

- **x** – list of time points
- **y** – list of CO2 values in given time points

Returns pair of air change rate and its error

static gen_f(*co2_start*, *co2_out*)
It generates a function.

Parameters

- **co2_start** – initial CO2 concentration
- **co2_out** – outdoor CO2 concentration

Returns function that requires timestamp and air change rate

static gen_f_volume(*co2_start*, *co2_out*, *volume*)
It generates a function.

Parameters

- **co2_start** – initial CO2 concentration
- **co2_out** – outdoor CO2 concentration

- **volume** – room volume

Returns function that requires timestamp and air change rate

Module contents

9.1.3 dm.coefficients package

Submodules

dm.coefficients.AbstractLineCoefficients module

Gets coefficients of equation in form ax + by + c = 0 from equation in form y = kx + q.

class dm.coefficients.AbstractLineCoefficients.**AbstractLineCoefficients**

Bases: abc.ABC

abstract calculate(*data*, *interval*, *col1*, *col2*, *col3*, *point_x*, *point_y*)

It calculates a slope of line(s).

Parameters

- **data** – training data
- **interval** – interval that denotes a ventilation length (class, cluster)
- **col1** – column name containing values of specific humidity measured when the events started
- **col2** – column name containing values of specific humidity measured when the events finished
- **col3** – column name containing values of differences between indoor and outdoor specific humidity when the events started
- **point_x** – x-coordinate of a point (cluster centroid)
- **point_y** – y-coordinate of a point (cluster centroid)

Returns slope of line(s)

convert_line(*coeffs*)

It converts line equation y = kx + q to the form ax + by + c = 0 (general form).

Parameters **coeffs** – coefficients of a line

Returns general form of a line

dm.coefficients.CenterLineSlope module

Calculates slope of a line passing through a point.

class dm.coefficients.CenterLineSlope.**CenterLineSlope**

Bases: *dm.coefficients.AbstractLineCoefficients.AbstractLineCoefficients*

calculate(*data*, *interval*, *col1*, *col2*, *col3*, *point_x*, *point_y*)

It calculates a slope of a line passing through a point.

Parameters

- **data** – training data

- **interval** – interval that denotes a ventilation length (class, cluster)
- **col1** – column name containing values of specific humidity measured when the events started
- **col2** – column name containing values of specific humidity measured when the events finished
- **col3** – column name containing values of differences between indoor and outdoor specific humidity when the events started
- **point_x** – x-coordinate of a point (cluster centroid)
- **point_y** – y-coordinate of a point (cluster centroid)

Returns slope of a line passing through a point

dm.coefficients.DistanceToLine module

Calculates distance between two points and between line and point.

Finds clusters of data using indoor specific humidity decrease and difference between indoor and outdoor humidity and calculates distance between point and cluster centroid and distance of point to cluster trendline.

class dm.coefficients.DistanceToLine.**DistanceToLine**(*training*)
Bases: object

distance_point_line(*a1*, *a2*, *a*, *b*, *c*)
It calculates distance from point to line.

Parameters

- **a1** – point coordinate x
- **a2** – point coordinate y
- **a** – parameter of the line equation
- **b** – parameter of the line equation
- **c** – parameter of the line equation

Returns distance from point to line

distance_point_point_Euclidean(*a1*, *a2*, *b1*, *b2*)
It calculates distance from point to point (Euclidean).

Parameters

- **a1** – point 1 coordinate x
- **a2** – point 1 coordinate y
- **b1** – point 2 coordinate x
- **b2** – point 2 coordinate y

Returns distance from point to point

exec(*intervals*, *data_testing*, *col1*, *col2*, *col3*, *strategy*, *strategyFlag*, *one_line*, *test_points*, *cluster_boundaries*, *cluster_boundaries_all*, *precision=2*)

It calculates coefficients used in general form of a line, x-coordinate and y-coordinate of a cluster centroid and it also creates object for matplotlib using humidity_clusters function. Moreover, it can create an output file containing trendline or clusters. It creates testing set and it alternatively plots testing data points.

Parameters

- **intervals** – list of intervals that denote ventilation lengths (class, cluster)
- **data_testing** – training data
- **col1** – column name containing values of specific humidity measured when the events started
- **col2** – column name containing values of specific humidity measured when the events finished
- **col3** – column name containing values of differences between indoor and outdoor specific humidity when the events started
- **strategy** – strategy how to compute a slope of line(s)
- **strategyFlag** – flag that denotes a used strategy for computation of a slope of line(s)
- **one_line** – if only one line should be plotted
- **test_points** – true if test points are plotted
- **cluster_boundaries** – if cluster boundaries should be plotted
- **cluster_boundaries_all** – if all cluster boundaries should be plotted
- **precision** –

Returns coefficients used in general form of a line, x-coordinate and y-coordinate of a cluster centroid and object for matplotlib

humidity_clusters(*training*, *col1*, *col2*, *col3*, *intervals*, *strategy*, *strategy_flag*, *one_line*, *cluster_boundaries*, *cluster_boundaries_all*)

It calculates coefficients used in general form of a line, x-coordinate and y-coordinate of a cluster centroid and it creates object for matplotlib

Parameters

- **training** – training data
- **col1** – column name containing values of specific humidity measured when the events started
- **col2** – column name containing values of specific humidity measured when the events finished
- **col3** – column name containing values of differences between indoor and outdoor specific humidity when the events started
- **intervals** – list of intervals that denote ventilation lengths (class, cluster)
- **strategy** – strategy how to compute a slope of line(s)
- **strategy_flag** – flag that denotes a used strategy for computation of a slope of line(s)
- **one_line** – if only one line should be plotted
- **cluster_boundaries** – if cluster boundaries should be plotted
- **cluster_boundaries_all** – if all cluster boundaries should be plotted

Returns coefficients used in general form of a line, x-coordinate and y-coordinate of a cluster centroid and object for matplotlib

static select_attributes(*data*, *attributes*)
It selects required attributes.

Parameters

- **data** – training or testing data
- **attributes** – list of required attributes

Returns list of required attributes

static ventilation_length_events(*training: list*, *ventilation_length: int*)
It gets events assigned to a given class.

It means that it gets events when ventilation lasted approximately the same time.

Parameters

- **training** – training data
- **ventilation_length** – length of ventilation that corresponds with a class

Returns list of events assigned to a given class

dm.coefficients.MathLineAvgSlope module

dm.coefficients.PolyfitLineAvgSlope module

Calculates average slope of multiple lines.

class dm.coefficients.PolyfitLineAvgSlope.**PolyfitLineAvgSlope**
Bases: *dm.coefficients.AbstractLineCoefficients.AbstractLineCoefficients*

calculate(*data*, *interval*, *col1*, *col2*, *col3*, *point_x*, *point_y*)
It calculates average slope of multiple lines.

It uses the polyfit function from numpy library.

Parameters

- **data** – training data
- **interval** – interval that denotes a ventilation length (class, cluster)
- **col1** – column name containing values of specific humidity measured when the events started
- **col2** – column name containing values of specific humidity measured when the events finished
- **col3** – column name containing values of differences between indoor and outdoor specific humidity when the events started
- **point_x** – deprecated
- **point_y** – deprecated

Returns average slope of multiple lines

Module contents

9.1.4 dm.models package

Subpackages

dm.models.estimate package

Submodules

dm.models.estimate.Estimate module

Estimates course of quantities.

class dm.models.estimate.Estimate.**Estimate**
Bases: object

static co2_level(*co2_value*)
It determines level of CO2 concentration.

Parameters **co2_value** – value of CO2

Returns level of CO2 concentration and order of level

static compute_lin_reg(*values*)
It computes linear regression from given values.

Parameters **values** – list of values

Returns pair of intercept and slope

static compute_min_win_size(*last_close_event_time*, *actual_time*)
It computes minimal window size.

Parameters

- **last_close_event_time** – last time when window was closed
- **actual_time** – current time

Returns minimal window size

static t_h_level(*temp*, *rh*)
It determines level of humidity.

Parameters

- **temp** – value of temperature
- **rh** – value of humidity

Returns level of absolute humidity, level of relative humidity and order of level

Module contents

dm.models.open_detector package

Submodules

dm.models.open_detector.adapted_models module

Creates models for window opening detection.

dm.models.open_detector.adapted_models.**filter_only_open**(*values*, *attr_name='value'*)

It gets events when a window was open.

Parameters

- **values** – dictionary of values
- **attr_name** – name of attribute

Returns list of events when a window was open

dm.models.open_detector.adapted_models.**prepare_adapted_data_co2**(*devs*, *cls*, *start*, *end*, *lat*, *lon*, *weather*)

It creates a model on the basis of CO2 concentration.

Parameters

- **devs** – list of devices
- **cls** – list of clients
- **start** – timestamp that denotes start of required time interval
- **end** – timestamp that denotes end of required time interval
- **lat** – latitude where data was gathered
- **lon** – longitude where data was gathered
- **weather** – object used to get information about weather

Returns model on the basis of CO2 concentration

dm.models.open_detector.adapted_models.**prepare_adapted_data_t_h**(*devs*, *cls*, *start*, *end*, *lat*, *lon*, *weather*)

It creates a model on the basis of temperature and humidity.

Parameters

- **devs** – list of devices
- **cls** – list of clients
- **start** – timestamp that denotes start of required time interval
- **end** – timestamp that denotes end of required time interval
- **lat** – latitude where data was gathered
- **lon** – longitude where data was gathered
- **weather** – object used to get information about weather

Returns model on the basis of temperature and humidity

dm.models.open_detector.create_attrs module

Creates attributes used in models for window opening detector and predictor of optimal ventilation length.

class dm.models.open_detector.create_attrs.**ColumnMapper**

Bases: object

NO_EVENTS_RECORDS_CO2 = [('2018/10/08 00:07:30', 'nothing'), ('2018/10/08 00:55:30', '

NO_EVENTS_RECORDS_T_H = [('2018/11/27 08:10:30', 'nothing'), ('2018/11/29 02:08:00', '

OPEN_CO2 = {'co2_in_ppm': 'co2_in'}

OPEN_T_H = {'rh_in2_absolute_g_m3': 'rh_in_absolute', 'rh_in2_specific_g_kg': 'rh_in

PREDICTOR_CO2 = {'co2_in_ppm_diff': 'co2_in_diff', 'rh_in_absolute_g_m3_diff': 'rh_i

PREDICTOR_T_H = {'rh_in2_absolute_g_m3_diff': 'rh_in_absolute_diff', 'rh_in2_specific

dm.models.open_detector.create_attrs.**func_co2**(*timestamp*, *row_selector*, *interval_selector*, *columns_map*, *end=None*)

It calculates attributes used for detector creation on the basis of CO2 concentration that detects window opening.

Parameters

- **timestamp** – timestamp when an event occurred
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **columns_map** – mapping of column names
- **end** – deprecated

Returns attributes used for detector creation on the basis of CO2 concentration

dm.models.open_detector.create_attrs.**func_predict_co2**(*timestamp*, *row_selector*, *interval_selector*, *columns_map*, *end=None*)

It calculates attributes used for predictor creation on the basis of CO2 concentration.

Parameters

- **timestamp** – timestamp when an event occurred
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **columns_map** – mapping of column names
- **end** – deprecated timestamp that denotes end of event

Returns attributes used for predictor creation on the basis of CO2 concentration

dm.models.open_detector.create_attrs.**func_predict_t_h**(*timestamp*, *row_selector*, *interval_selector*, *columns_map*, *end=None*)

It calculates attributes used for predictor creation on the basis of temperature and humidity.

Parameters

- **timestamp** – timestamp when an event occurred
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **columns_map** – mapping of column names

- **end** – timestamp that denotes end of event

Returns attributes used for predictor creation on the basis of temperature and humidity

dm.models.open_detector.create_attrs.**func_t_h**(*timestamp*, *row_selector*, *interval_selector*, *columns_map*, *end=None*)

It calculates attributes used for detector creation on the basis of temperature and humidity that detects window opening.

Parameters

- **timestamp** – timestamp when an event occurred
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **columns_map** – mapping of column names
- **end** – deprecated

Returns attributes used for detector creation on the basis of temperature and humidity

dm.models.open_detector.generic_training_file module

Creates adapted models for window opening detection.

dm.models.open_detector.generic_training_file.**add_func**(*events*, *cls*, *devs*, *model_type*, *interval_extension*, *mapper*, *func*, *weather*)

It adds events to training dataset.

The training dataset contains events when a window was closed.

Parameters

- **events** – list of events
- **cls** – list of clients
- **devs** – list of devices
- **model_type** – type of model - based on CO2 concentration or temperature and humidity
- **interval_extension** – time shift that is subtracted or added to start or end of an interval respectively
- **mapper** – dictionary containing mapping of attribute name in database to the name used in dataset
- **func** – object that determines function used for attribute calculation
- **weather** – object used to get information about weather

Returns records containing information about given events

dm.models.open_detector.generic_training_file.**generic_training_file**(*events_file*, *no_event_time_shift*, *model_type*, *columns_map*, *cls*, *devs*, *weather*)

It creates generic training dataset.

The training dataset contains the same number of events when a window was open and when it was closed.

Parameters

- **events_file** – file containing events
- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed
- **model_type** – type of model - based on CO2 concentration or temperature and humidity
- **columns_map** – mapping of column names
- **cls** – list of clients
- **devs** – list of devices
- **weather** – object used to get information about weather

Returns generic training dataset

dm.models.open_detector.generic_training_file.**training_data**(*json_f*, *cls*, *devs*, *model_type*, *no_event_time_shift*, *interval_extension*, *mapper*, *func*, *weather*)

It creates training dataset.

The training dataset contains events when a window was open.

Parameters

- **json_f** – dictionary that contains information about events
- **cls** – list of clients
- **devs** – list of devices
- **model_type** – type of model - based on CO2 concentration or temperature and humidity
- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed
- **interval_extension** – time shift that is subtracted or added to start or end of an interval respectively
- **mapper** – dictionary containing mapping of attribute name in database to the name used in dataset
- **func** – object that determines function used for attribute calculation
- **weather** – object used to get information about weather

Returns training dataset

dm.models.open_detector.generic_training_file_from_local_db module

Detector for window opening based on CO2 decrease with iterations.

dm.models.open_detector.generic_training_file_from_local_db.**generic_testing**(*directory*, *columns_map*)

It creates several testing datasets.

Parameters

- **directory** – name of directory
- **columns_map** – mapping of column names

Returns None

dm.models.open_detector.generic_training_file_from_local_db.**testing_month**(*table_name*, *start*, *directory*, *columns_map*)

It creates testing dataset month by month.

Parameters

- **table_name** – name of table
- **start** – timestamp that denotes start of time interval
- **end** – timestamp that denotes end of time interval
- **directory** – name of directory
- **columns_map** – mapping of column names

Returns None

dm.models.open_detector.generic_training_file_from_local_db.**testing_set**(*table_name: str*, *start*, *end*, *filename*, *columns_map*)

It creates testing dataset.

Parameters

- **table_name** – name of table
- **start** – timestamp that denotes start of time interval
- **end** – timestamp that denotes end of time interval
- **filename** – name of output file
- **columns_map** – mapping of column names

Returns None

dm.models.open_detector.generic_training_file_from_local_db.**training_set_co2**(*events_file*, *no_event_time_shift*, *table_name*, *output_filename*, *columns_map*)

It creates balanced training dataset for detector creation based on CO2 concentration and CSV file containing the dataset.

Parameters

- **events_file** – file containing events

- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed
- **table_name** – name of table
- **output_filename** – name of output file
- **columns_map** – mapping of column names

Returns balanced training dataset for detector creation based on CO2 concentration

dm.models.open_detector.generic_training_file_from_local_db.**training_set_t_h**(*events_file*, *no_event_time_shift*, *table_name*, *output_filename*, *columns_map*)

It creates balanced training dataset for detector creation based on temperature and humidity and CSV file containing the dataset.

Parameters

- **events_file** – file containing events
- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed
- **table_name** – name of table
- **output_filename** – name of output file
- **columns_map** – mapping of column names

Returns balanced training dataset for detector creation based on temperature and humidity

Module contents

dm.models.predictor package

Submodules

dm.models.predictor.THPredictorUtil module

Support for ventilation length prediction.

Creates training and testing sets for ventilation length prediction, including calculation of distance between a data point and cluster trendline (cluster centroid).

dm.models.predictor.THPredictorUtil.**training_testing_data**(*data*, *splitting*)

It creates training and testing data set.

Parameters

- **data** – dictionary of data
- **splitting** – the percentage part of data used for training purposes

Returns training, testing data sets and minimal number of events that lasted certain time

dm.models.predictor.THPredictorUtil.**training_testing_data_only_distance**(*training*, *testing*, *strategy*, *strategyFlag*, *one_line*, *test_points*, *cluster_boundaries*, *cluster_boundaries_all*, *training_file*, *testing_file*)

It creates training and testing data sets including only attributes related to distance.

Parameters

- **training** – list of training data
- **testing** – list of testing data
- **strategy** – strategy how to compute a slope of line(s)
- **strategyFlag** – flag that denotes a used strategy for computation of a slope of line(s)
- **one_line** – if only one line should be plotted
- **test_points** – true if test points are plotted
- **cluster_boundaries** – if cluster boundaries should be plotted
- **cluster_boundaries_all** – if all cluster boundaries should be plotted
- **training_file** – filename to which training data set is written
- **testing_file** – filename to which training data set is written

Returns training and testing data sets including only attributes related to distance

dm.models.predictor.THPredictorUtil.**training_testing_data_with_distance**(*training*, *testing*, *strategy*, *strategyFlag*, *one_line*, *test_points*, *cluster_boundaries*, *cluster_boundaries_all*, *training_file*, *testing_file*)

It creates training and testing data sets including attributes related to distance.

Parameters

- **training** – list of training data
- **testing** – list of testing data
- **strategy** – strategy how to compute a slope of line(s)
- **strategyFlag** – flag that denotes a used strategy for computation of a slope of line(s)
- **one_line** – if only one line should be plotted
- **test_points** – true if test points are plotted
- **cluster_boundaries** – if cluster boundaries should be plotted
- **cluster_boundaries_all** – if all cluster boundaries should be plotted
- **training_file** – filename to which training data set is written
- **testing_file** – filename to which training data set is written

Returns training and testing data sets including attributes related to distance

dm.models.predictor.THPredictorUtil.**training_testing_data_without_distance**(*training*, *testing*, *strategy*, *strategyFlag*, *one_line*, *test_points*, *cluster_boundaries*, *cluster_boundaries_all*, *training_file*, *testing_file*)

It creates training and testing data sets without attributes related to distance.

Parameters

- **training** – list of training data
- **testing** – list of testing data
- **strategy** – deprecated
- **strategyFlag** – deprecated
- **one_line** – deprecated
- **test_points** – deprecated
- **cluster_boundaries** – deprecated
- **cluster_boundaries_all** – deprecated
- **training_file** – filename to which training data set is written
- **testing_file** – filename to which training data set is written

Returns training and testing data sets without attributes related to distance

dm.models.predictor.generic_training_file module

Creates adapted models for predictor of optimal ventilation length.

dm.models.predictor.generic_training_file.**create_bins**(*data*, *bins*)

It creates required number of bins.

Parameters

- **data** – dictionary of data
- **bins** – number of bins

Returns dictionary of data

dm.models.predictor.generic_training_file.**training_data**(*json_f*, *cls*, *devs*, *model_type*, *interval_extension*, *mapper*, *func*, *lat*, *lon*, *weather*)

It creates training data.

Parameters

- **json_f** – dictionary of data
- **cls** – list of clients
- **devs** – list of devices
- **model_type** – type of model - based on CO2 concentration or temperature and humidity
- **interval_extension** – time shift that is subtracted or added to start or end of an interval respectively
- **mapper** – dictionary containing mapping of attribute name in database to the name used in dataset
- **func** – object that determines function used for attribute calculation
- **lat** – latitude of a locality
- **lon** – longitude of a locality
- **weather** – object used to get information about weather

Returns dictionary of training data

dm.models.predictor.generic_training_file.**training_file_co2**(*events_file: str*, *no_event_time_shift: int*, *cls*, *devs*, *lat*, *lon*, *weather*)

It creates training data for predictor based on CO2 concentration.

Parameters

- **events_file** – file containing events
- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed
- **cls** – list of clients
- **devs** – list of devices
- **lat** – latitude of a locality
- **lon** – longitude of a locality
- **weather** – object used to get information about weather

Returns dictionary of training data for predictor based on CO2 concentration

dm.models.predictor.generic_training_file.**training_file_t_h**(*events_file: str*, *no_event_time_shift: int*, *cls*, *devs*, *lat*, *lon*, *weather*)

It creates training data for predictor based on temperature and humidity.

Parameters

- **events_file** – file containing events

- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed
- **cls** – list of clients
- **devs** – list of devices
- **lat** – latitude of a locality
- **lon** – longitude of a locality
- **weather** – object used to get information about weather

Returns dictionary of training data for predictor based on temperature and humidity

dm.models.predictor.generic_training_file_from_local_db module

Creates training datasets from local database.

dm.models.predictor.generic_training_file_from_local_db.**create_bins**(*data*, *bins*)

It creates required number of bins.

Parameters

- **data** – dictionary of data
- **bins** – number of bins

Returns dictionary of data

dm.models.predictor.generic_training_file_from_local_db.**training_file_co2**(*events_file: str*, *no_event_time_shift: int*)

It creates training data for predictor based on CO2 concentration.

Parameters

- **events_file** – file containing events
- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed

Returns dictionary of training data for predictor based on CO2 concentration.

dm.models.predictor.generic_training_file_from_local_db.**training_file_t_h**(*events_file: str*, *no_event_time_shift: int*)

It creates training data for predictor based on temperature and humidity.

Parameters

- **events_file** – file containing events
- **no_event_time_shift** – number of seconds subtracted from start of event to define event when a window was closed

Returns dictionary of training data for predictor based on temperature and humidity.

Module contents

Submodules

dm.models.ModelsUtil module

class dm.models.ModelsUtil.**ModelsUtil**
Bases: object

static anomaly_diff(*devs*, *cls*, *lat*, *lon*, *actual_time*, *weather*, *min_diff*, *diff_time*, *a_type*)
It detect an anomaly within given time interval.

Parameters

- **devs** – information about device
- **cls** – list of clients
- **lat** – latitude of locality
- **lon** – longitude of locality
- **actual_time** – current timestamp
- **weather** – class that obtains information about weather
- **min_diff** – minimal difference between measured values
- **diff_time** – window size for anomaly detection
- **a_type** – type of anomaly detection (temperature, humidity or co2)

Returns notification that contains information about anomaly

static dew_point(*devs*, *cls*, *lat*, *lon*, *actual_time*, *weather*)
It calculates dew point.

Parameters

- **devs** – information about device
- **cls** – list of clients
- **lat** – latitude of locality
- **lon** – longitude of locality
- **actual_time** – current timestamp
- **weather** – class that obtains information about weather

Returns notification that contains information about dew point

static estimate_co2(*device*, *cls*, *actual_time*)
It estimates values in given time on the basis of CO2 concentration.

Parameters

- **device** – information about device
- **cls** – list of clients
- **actual_time** – current timestamp

Returns notification that contains information about estimation

static estimate_open_co2(*device*, *cls*, *lat*, *lon*, *actual_time*, *weather*)
It estimates if window was open on the basis of CO2 concentration.

Parameters

- **device** – information about device
- **cls** – list of clients
- **lat** – latitude of locality
- **lon** – longitude of locality
- **actual_time** – current timestamp
- **weather** – class that obtains information about weather

Returns notification that contains information about estimation

static estimate_open_t_h(*device*, *cls*, *lat*, *lon*, *actual_time*, *weather*)
It estimates if window was open on the basis of temperature and humidity.

Parameters

- **device** – information about device
- **cls** – list of clients
- **lat** – latitude of locality
- **lon** – longitude of locality
- **actual_time** – current timestamp
- **weather** – class that obtains information about weather

Returns notification that contains information about estimation

static estimate_t_h(*device*, *cls*, *actual_time*)
It estimates values in given time on the basis of humidity.

Parameters

- **device** – information about device
- **cls** – list of clients
- **actual_time** – current timestamp

Returns notification that contains information about estimation

static json_to_file(*json_data*, *filename*, *log_notification*)
It saves JSON structure into the file.

Parameters

- **json_data** – JSON structure containing data
- **filename** – name of output file
- **log_notification** – true if notification is logged

Returns None

static load_model(*type*, *device*)
It loads model for window opening detector from file.

Parameters

- **type** – type of model, t_h or co2

- **device** – information about device

Returns loaded model

static load_model_predictor(*type*, *device*)
It loads model for predictor from file.

Parameters

- **type** – type of model, t_h or co2
- **device** – information about device

Returns loaded model

static predictor(*device*, *cls*, *lat*, *lon*, *actual_time*, *module_type*, *weather*, *decrease=None*)
It predicts length of ventilation to decrease quantity to required level.

Parameters

- **device** – information about device
- **cls** – list of clients
- **lat** – latitude of locality
- **lon** – longitude of locality
- **actual_time** – current timestamp
- **module_type** – type of module, t_h or CO2
- **weather** – class that obtains information about weather
- **decrease** – value that defines how much quantity value should be decreased

Returns notification that contains information about estimation

static read_generic_data(*model_type*)
It gets list of attributes from file.

The attributes are used in model for window opening detector.

Parameters **model_type** – type of model, t_h or co2

Returns list of attributes

static replace_co2_ventilation_len(*data*)
It moves value of CO2 concentration after regression to event attribute.

Parameters **data** – list of data

Returns modified list of data

static replace_nothing_open(*data*)
It converts value of event attribute to boolean.

Attribute value open is represented by 1 and close by 0.

Parameters **data** – list of data

Returns modified list of data

static replace_ventilation_length(*data*)
It removes quotation marks in value of ventilation length attribute.

Parameters **data** – list of data

Returns modified list of data

static when_ventilate_summer(*devs*, *cls*, *lat*, *lon*, *actual_time*, *weather*, *temperature_diff*)
It determines if ventilation should be performed on the basis of temperature difference.

Parameters

- **devs** – information about device
- **cls** – list of clients
- **lat** – latitude of locality
- **lon** – longitude of locality
- **actual_time** – current timestamp
- **weather** – class that obtains information about weather
- **temperature_diff** – difference between inside and outside temperature

Returns notification that contains information about estimation

static write_model(*attrs*, *filename*, *replace*, *event_column='event'*)
It saves model to the output file.

Parameters

- **attrs** – list of attributes
- **filename** – name of output file
- **replace** – function that replaces an attribute
- **event_column** – name of attribute that denotes class

Returns None

Module contents

9.1.5 dm.selectors package

Subpackages

dm.selectors.from_server package

Submodules

dm.selectors.from_server.CachedDataFromServer module

Selector that gets data from server.

class dm.selectors.from_server.CachedDataFromServer.**CachedDataFromServer**
Bases: *dm.selectors.AbstractSelector.AbstractSelector*

init_cache(*measured*, *weather*)
It initializes cache using measured data and weather data.

Parameters

- **measured** – list of measured data
- **weather** – list of weather data

Returns None

interval(*column_name*, *start*, *end*)
It selects an interval from cache.

Parameters

- **column_name** – name of column that contains required values
- **start** – timestamp that denotes start of the required interval
- **end** – timestamp that denotes end of the required interval

Returns data from the column in given interval (start, end)

row(*column_name*, *time*)
It selects one row from cache.

Parameters

- **column_name** – name of column that contains required values
- **time** – timestamp of required data

Returns data from the column in given time

Module contents

dm.selectors.interval package

Submodules

dm.selectors.interval.AbstractTableIntervalSelector module

Abstract class for interval selector.

Selector can select given intervals of values from a given table.

class dm.selectors.interval.AbstractTableIntervalSelector.**AbstractTableIntervalSelector**(*con*
ta-
*ble_*

Bases: *dm.selectors.AbstractSelector.AbstractSelector*

abstract interval(*column_name*, *start*, *end*)
It selects a required interval from source.

Parameters

- **column_name** – name of column that contains required values
- **start** – timestamp that denotes start of the required interval
- **end** – timestamp that denotes end of the required interval

Returns data from the column in given interval (start, end)

dm.selectors.interval.SimpleIntervalSelector module

Simple interval selector from database.

Selector can select given intervals of values from a given table without cache.

class dm.selectors.interval.SimpleIntervalSelector.**SimpleTableIntervalSelector**(*con*, *table_name*)

Bases: *dm.selectors.interval.AbstractTableIntervalSelector.AbstractTableIntervalSelector*

interval(*column_name*, *start*, *end*)

It select a required interval from database.

Parameters

- **column_name** – name of column that contains required values
- **start** – timestamp that denotes start of the required interval
- **end** – timestamp that denotes end of the required interval

Returns data from the column in given interval (start, end)

Module contents

dm.selectors.row package

Submodules

dm.selectors.row.AbstractTableRowSelector module

Abstract class for one value selector.

Selector can select value based on given time from a table.

class dm.selectors.row.AbstractTableRowSelector.**AbstractRowSelector**(*con*, *table_name*)

Bases: *dm.selectors.AbstractSelector.AbstractSelector*

abstract row(*column_name*, *time*)

It selects one row from source.

Parameters

- **column_name** – name of column that contains required values
- **time** – timestamp of required data

Returns data from the column in given time

dm.selectors.row.CachedDiffRowWithIntervalSelector module

Cached row selector for selection of differences of values.

class dm.selectors.row.CachedDiffRowWithIntervalSelector.**CachedDiffRowWithIntervalSelector**

Bases: *dm.selectors.row.CachedRowWithIntervalSelector.CachedRowWithIntervalSelector*

row(*column_name*, *time*)
It selects one row from cache.

It also performs difference between two obtained values.

Parameters

- **column_name** – name of column that contains required values
- **time** – timestamp of required data

Returns data from the column in given time

dm.selectors.row.CachedRowWithIntervalSelector module

Cached interval selector from database.

Selector can select given intervals of values from a given table and selected rows are stored into cache.

class dm.selectors.row.CachedRowWithIntervalSelector.**CachedRowWithIntervalSelector**(*con*, *table_name*, *start*, *end*)

Bases: *dm.selectors.row.SimpleCachedRowSelector.SimpleCachedRowSelector*

row(*column_name*, *time*)
It selects one row from cache.

If the method is called for the first time, cache will be initialized.

Parameters

- **column_name** – name of column that contains required values
- **time** – timestamp of required data

Returns data from the column in given time

dm.selectors.row.LinearSimpleCachedRowSelector module

dm.selectors.row.SimpleCachedRowSelector module

Cached selector for one value from database.

Selector can select value based on given time from a table in database and selected values are stored in cache.

class dm.selectors.row.SimpleCachedRowSelector.**SimpleCachedRowSelector**(*con*, *table_name*)

Bases: *dm.selectors.row.AbstractTableRowSelector.AbstractRowSelector*

row(*column_name*, *time*)
It selects one row from cache or database.

If the method is called for the first time, cache will be initialized. The second call of the method returns loaded data stored in cache.

Parameters

- **column_name** – name of column that contains required values

- **time** – timestamp of required data

Returns data from the column in given time

dm.selectors.row.SimpleRowSelector module

Module contents

Submodules

dm.selectors.AbstractSelector module

Abstract class for selector from arbitrary source.

class dm.selectors.AbstractSelector.**AbstractSelector**
Bases: abc.ABC

clear()
It clears selector cache.

Returns None

Module contents

9.2 Submodules

9.3 dm.AttributeUtil module

Creates (additional) training set and testing set, keeps number of records in both sets in equilibrium.

class dm.AttributeUtil.**AttributeUtil**
Bases: object

static additional_training_set(*no_event_records*, *func*, *row_selector*, *interval_selector*, *columns_map*, *print_each=10*)
It additionally generates training data based on given time points.

Parameters

- **no_event_records** – list of no events
- **func** – function that computes the attributes
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **columns_map** – dictionary containing mapping of attribute name in database to the name used in dataset
- **print_each** – step in printing of log messages, other log messages will not be printed

Returns list containing pairs of attribute name and value

static balance_set(*training_set*, *additional_training_set*)
It computes training data whereas opposite events are replaced.

It computes training data whereas opposite events are replaced with events from additional training set, the result is a balanced list.

Parameters

- **training_set** – list of training data
- **additional_training_set** – list of additional training data

Returns list containing pairs of attribute name and value that is balanced

static cached_training_data(*events*, *func*, *row_selector*, *interval_selector*, *event_type*, *file_path*, *columns_map*, *print_each=10*)

It gets training data.

If method is called for the first time, training data will be computed and stored to file. Otherwise it only reads training data from the file.

Parameters

- **events** – list of events
- **func** – function that computes the attributes
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **event_type** – type of event - open or close
- **file_path** – path to a cached file
- **columns_map** – dictionary containing mapping of attribute name in database to the name used in dataset
- **print_each** – step in printing of log messages, other log messages will not be printed

Returns pair of lists, the first list contains names of attributes and the second one contains pairs of attribute name and value

static testing_data(*con*, *table_name*, *start*, *end*, *write_each*, *func*, *row_selector*, *interval_selector*, *event_type*, *columns_map*, *log_every_hour=3*)

It generates testing data.

Parameters

- **con** – connection to the database
- **table_name** – name of table
- **start** – start of interval from which testing data is generated
- **end** – end of interval from which testing data is generated
- **write_each** – step in writing of computed attributes to the output file
- **func** – function that computes the attributes
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **event_type** – type of event - open or close
- **columns_map** – dictionary containing mapping of attribute name in database to the name used in dataset
- **log_every_hour** – step in printing of log messages, other log messages will not be printed

Returns list containing pairs of attribute name and value

static testing_data_with_write(*con*, *table_name*, *start*, *end*, *write_each*, *func*, *row_selector*, *interval_selector*, *event_type*, *output_filename*, *columns_map*, *row_count=2048*, *log_every_hour=1*)

It generates testing data, continuous writing to a file is optional.

Parameters

- **con** – connection to the database
- **table_name** – name of table
- **start** – start of interval from which testing data is generated
- **end** – end of interval from which testing data is generated
- **write_each** – step in writing of computed attributes to the output file
- **func** – function that computes the attributes
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **event_type** – type of event - open or close
- **output_filename** – output filename
- **columns_map** – dictionary containing mapping of attribute name in database to the name used in dataset
- **row_count** – number of attributes that are written at one time
- **log_every_hour** – step in printing of log messages, other log messages will not be printed

Returns list containing pairs of attribute name and value

static testing_one_row(*func*, *timestamp*, *row_selector*, *interval_selector*, *event_type*, *c_mapper*, *end*)

It computes one row with attributes.

Parameters

- **func** – function that computes the attributes
- **timestamp** – timestamp when an event occurred
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **event_type** – type of event - open or close
- **c_mapper** – dictionary containing mapping of attribute name in database to the name used in dataset
- **end** – if the end of an event is available

Returns list containing pairs of attribute name and value

static training_data(*events*, *func*, *row_selector*, *interval_selector*, *event_type*, *columns_map*, *print_each=10*)

It generates training data.

Parameters

- **events** – list of events
- **func** – function that computes the attributes
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **event_type** – type of event - open or close
- **columns_map** – dictionary containing mapping of attribute name in database to the name used in dataset
- **print_each** – step in printing of log messages, other log messages will not be printed

Returns pair of lists, the first list contains names of attributes and the second one contains pairs of attribute name and value

static training_data_without_opposite(*events*, *func*, *row_selector*, *interval_selector*, *columns_map*, *print_each=10*)

It computes training data without opposite.

Opposite event represents an event when nothing occurred.

Parameters

- **events** – list of events
- **func** – function that computes the attributes
- **row_selector** – selector of row
- **interval_selector** – selector of interval
- **columns_map** – containing mapping of attribute name in database to the name used in dataset
- **print_each** – step in printing of log messages, other log messages will not be printed

Returns list containing pairs of attribute name and value

9.4 dm.BeeeOnClient module

REST API client for communication with BeeeOn server.

class dm.BeeeOnClient.**BeeeOnClient**(*host*, *port*)

Bases: object

Client for communication with server supporting BeeeOn api.

property api_key

device_info(*gateway_id*, *device_id*)

It gets information about device.

Parameters

- **gateway_id** – identifier of gateway
- **device_id** – identifier of device

Returns json that contains information about device

history(*gateway*, *device*, *sensor*, *start*, *end*, *interval=1*, *aggregation='avg'*)

It gets historical data from server.

Parameters

- **gateway** – identifier of gateway
- **device** – identifier of device
- **sensor** – identifier of sensor
- **start** – start of interval from which data is generated
- **end** – end of interval from which data is generated
- **interval** – time interval from which data are aggregated
- **aggregation** – type of aggregation

Returns json that contains historical data from given time interval

refresh_token()
It refreshes token.

Returns token id

sensors_info(*gateway_id*, *device_id*)
It gets information about sensor.

Parameters

- **gateway_id** – identifier of gateway
- **device_id** – identifier of device

Returns json that contains information about sensor

property token_id

9.5 dm.CSVUtil module

Creates a csv file from OrderedDict.

class dm.CSVUtil.**CSVUtil**
Bases: object

static create_csv_file(*data: list*, *filename: str*, *enable_append=False*)
It creates a CSV file.

Parameters

- **data** – list of pairs containing attribute name and its value
- **filename** – output filename
- **enable_append** – if it is true, data is appended to the end of file

Returns None

static create_csv_io(*data: list*)
It creates StringIO from given data.

Parameters **data** – list of pairs containing attribute name and its value

Returns StringIO containing given data

9.6 dm.ConnectionUtil module

Utils to create connections to db, REST client and utils for environment setting.

class dm.ConnectionUtil.**ConnectionUtil**
Bases: object

MAX_TESTABLE_EVENTS = 75

static ant_work_api_key(*server_type*, *config_file='/etc/dp/config.ini'*)
It gets information about api key from configuration file.

Parameters

- **server_type** – type of server - dm
- **config_file** – path to configuration file

Returns information about api key

static create_con(*config_file='/etc/dp/config.ini'*)
It creates connection to the database.

Parameters **config_file** – path to the configuration file

Returns connection to the database

static estimate(*key*, *config_file='/etc/dp/config.ini'*)
It gets information about estimation from configuration file.

Parameters

- **key** – key for obtaining of data from information about estimation
- **config_file** – path to configuration file

Returns information about estimation obtained using the key

static get_value(*key*, *config_file*)
It gets value using given key from configuration file.

Parameters

- **key** – key for obtaining of data from information
- **config_file** – path to the configuration file

Returns required data obtained using the key

static is_testable_system()
It gets information if system is travis for testing.

Returns true if system is travis for testing, otherwise false

static open_detector(*key*, *config_file='/etc/dp/config.ini'*)
It gets information about open detector from configuration file.

Parameters

- **key** – key for obtaining of data from information about open detector
- **config_file** – path to configuration file

Returns information about open detector obtained using the key

static open_weather_api_key(*config_file='/etc/dp/config.ini'*)
It gets information about OpenWeather.org api key from configuration file.

Parameters **config_file** – path to configuration file

Returns information about api key

static package(*key*, *config_file='/etc/dp/config.ini'*)
It gets information about package from configuration file.

Parameters

- **key** – key for obtaining of data from information about package
- **config_file** – path to configuration file

Returns information about package obtained using the key

static predictor(*key*, *config_file='/etc/dp/config.ini'*)
It gets information about OpenWeather.org api key from configuration file.

Parameters

- **key** – key for obtaining of data from information about predictor
- **config_file** – path to configuration file

Returns information about predictor obtained using the key

static rapid_miner(*config_file='/etc/dp/config.ini'*)
It gets information about RapidMiner.

Parameters **config_file** – path to the configuration file

Returns information about RapidMiner

static rehivetech_api_key(*server_type*, *config_file='/etc/dp/config.ini'*)
It gets information about api key from configuration file.

Parameters

- **server_type** – type of server - dm or acontroller
- **config_file** – path to configuration file

Returns information about api key

static setup_clients()
It sets clients used for communication with servers.

Returns dictionary of set clients

static setup_clients_logout(*cls*)
It log outs clients.

Parameters **cls** – list of clients for communication with server

Returns None

static setup_logging(*log_level=10*)
It sets loggers for logging.

Parameters **log_level** – type of logging level

Returns None

static wunderground_api_key(*config_file='/etc/dp/config.ini'*)
It gets information about WunderGround.com api key from configuration file.

Parameters **config_file** – path to configuration file

Returns information about api key

9.7 dm.DBUtil module

Defines functions for working with database.

class dm.DBUtil.**DBUtil**
Bases: object

static check_timestamp_order(*con*, *table_name*)
It checks if a given table in database does not contain missing value.

Parameters

- **con** – connection to the database
- **table_name** – name of table

Returns None

static create_table(*conn*, *table_name*)
It creates table containing measured values.

http://www.mysqltutorial.org/mysql-decimal/ :param conn: connection to the database :param table_name: name of table :return: None

static delete_from_time(*con*, *table_name*, *delay*)
It removes several last rows from a given table.

Parameters

- **con** – connection to the database
- **table_name** – name of table
- **delay** – value used to define size of window from which rows are removed

Returns None

static drop_table(*conn*, *table_name*)
It removes a given table.

Parameters

- **conn** – connection to the database
- **table_name** – name of table

Returns None

static first_inserted_values(*conn*, *table_name*)
It gets the first inserted value into given table.

Parameters

- **conn** – connection to the database
- **table_name** – name of table

Returns first inserted value

static insert_value(*conn*, *task*, *enable_commit*, *table_name*)
It inserts values into the table.

Parameters

- **conn** – connection to the database
- **task** – list of pairs that contain name of column and value

- **enable_commit** – if true, result of a query is written into the database
- **table_name** – name of table

Returns None

static last_inserted_open_close_state(*conn*, *table_name*)
It gets the last inserted state of window (open or closed).

Parameters

- **conn** – connection to the database
- **table_name** – name of table

Returns last inserted window state

static last_inserted_values(*conn*, *table_name*)
It gets the last inserted value into given table.

Parameters

- **conn** – connection to the database
- **table_name** – name of table

Returns last inserted value

static measured_values_table_column_names()
It return list of available column names in table.

Returns list of column names

static rows_count(*conn*, *table_name*)
It gets number of rows in given table.

Parameters

- **conn** – connection to the database
- **table_name** – name of table

Returns number of rows in the table

static update_attribute(*con*, *table_name*, *attribute*, *value*, *timestamp*)
It updates value of a given attribute in given time.

Parameters

- **con** – connection to the database
- **table_name** – name of table
- **attribute** – name of attribute
- **value** – new value
- **timestamp** – time that denotes a row that will be updated

Returns None

9.8 dm.DateTimeUtil module

Converts various time formats (one to another).

class dm.DateTimeUtil.**DateTimeUtil**
Bases: object

static create_interval_str(*start*, *end*)
It converts given interval expressed using UTC timestamp to string form.

Parameters

- **start** – start of interval
- **end** – end of interval

Returns interval expressed in string form

static local_time_str_to_utc(*date_str*, *timezone='Europe/Prague'*, *format='%Y/%m/%d %H:%M:%S'*)
It converts given local time in string form to UTC form (number).

Parameters

- **date_str** – local time in string form
- **timezone** – timezone of local time
- **format** – output format of time

Returns object containing local time in UTC form

static utc_timestamp_to_local_time(*timestamp*, *timezone='Europe/Prague'*)
It converts given UTC timestamp to local time according to timezone.

Parameters

- **timestamp** – UTC timestamp
- **timezone** – timezone of local time

Returns local time

static utc_timestamp_to_str(*timestamp*, *format='%Y-%m-%d %H:%M:%S'*)
It converts UTC timestamp to local time in string form.

Parameters

- **timestamp** – UTC timestamp
- **format** – output format of time

Returns local time in string form

9.9 dm.Differences module

9.10 dm.ExampleRunner module

Runs detector or more detectors.

class dm.ExampleRunner.**ExampleRunner**
Bases: object

static detector(*before*, *after*, *script_name*, *process*, *enable_training*)
It runs a process in RapidMiner.

Parameters

- **before** – time shift before start of event
- **after** – time shift after start of event
- **script_name** – name of script
- **process** – name of RapidMiner process
- **enable_training** – if training phase should be performed

Returns None

static feature_stats(*script_name*, *enable_training*)
It runs computing of statistics.

Parameters

- **script_name** – name of script
- **enable_training** – if training phase should be performed

Returns None

9.11 dm.FilterUtil module

Defines various filters to select data that meets a required condition.

class dm.FilterUtil.**FilterUtil**
Bases: object

static attribute(*events*, *attribute_name*, *value*)
It gets events in which attribute contains given value.

Parameters

- **events** – list of events
- **attribute_name** – name of attribute
- **value** – required value of attribute

Returns list of events in which attribute contains given value

static attribute_exclude(*events*, *attribute_name*, *value*)
It gets events in which attribute does not contain given value.

Parameters

- **events** – list of events
- **attribute_name** – name of attribute
- **value** – not required value of attribute

Returns list of events in which attribute does not contain given value

static derivation_not_zero(*events*)
It gets events in which derivation is not zero.

Parameters **events** – list of events

Returns list of events in which derivation is not zero

static humidity(*events*, *min_out_specific_humidity*, *min_diff*, *max_diff*)
It gets events that fulfill the conditions.

It gets events in which specific humidity difference was in admissible range and outside specific humidity was not higher than given value.

Parameters

- **events** – list of events
- **min_out_specific_humidity** –
- **min_diff** –
- **max_diff** –

Returns list of events

static measured_values_not_empty(*events*, *attributes*)
It gets events that include valid measured data in given attributes.

Parameters

- **events** – list of events
- **attributes** – names of attributes

Returns list of events that include valid measured data in given attributes

static min_length(*events*, *length*)
It gets events that lasted at least given time.

Parameters

- **events** – list of events
- **length** – minimal length of event

Returns list of events that lasted at least given time

static min_max_time_interval(*events*, *min_time*, *max_time*)
It gets events that lasted admissible time.

Parameters

- **events** – list of events
- **min_time** – minimal length of event
- **max_time** – maximal length of event

Returns list of events that lasted admissible time

static min_time_interval(*events*, *time_interval_length*)
It gets events that lasted at least the required time.

Parameters

- **events** – list of events
- **time_interval_length** – length of time interval

Returns list of events that lasted at least the required time

static min_timestamp(*events*, *timestamp*)
It gets events that started after given time.

Parameters

- **events** – list of events
- **timestamp** – timestamp from which events are considered

Returns list of events that started after given time.

static only_valid_events(*events*)
It gets only valid events.

Parameters **events** – list of events

Returns list of valid events

static temperature_diff(*events*, *min_value*, *max_value*)
It gets events in which temperature difference was in admissible range.

Parameters

- **events** – list of events
- **min_value** – minimal difference of temperature
- **max_value** – maximal difference of temperature

Returns list of events in which temperature difference was in admissible range

static temperature_out_max(*events*, *max_value*)
It gets events in which temperature was not higher than given value.

Parameters

- **events** – list of events
- **max_value** – maximal temperature

Returns list of events in which temperature was not higher than given value

9.12 dm.Graph module

Utils to create graph that can be available in html file.

class dm.Graph.**Graph**(*path*)
Bases: object

static db_to_simple_graph(*event*, *column*, *color*, *label*, *number_output_records*)
It generates a graph based on data in a database.

Parameters

- **event** – an event
- **column** – name of column (attribute)
- **color** – color of curve
- **label** – label for x-axis
- **number_output_records** – number of time points in a graph

Returns graph containing course of a given attribute

gen(*data*, *output*, *scale_padding_min=0*, *scale_padding_max=0*, *g_type='line'*, *min_value=None*, *max_value=None*, *global_range=False*)
It generates a HTML page containing a graph.

Parameters

- **data** – data containing information about events
- **output** – filename
- **scale_padding_min** – padding calculated using the minimal value of an attribute
- **scale_padding_max** – padding calculated using the maximal value of an attribute
- **g_type** – shape of a curve
- **min_value** – minimal value of an attribute in a graph
- **max_value** – maximal value of an attribute in a graph
- **global_range** – if true, range is calculated for each graph from given data, otherwise the same range is used for all graphs

Returns None

9.13 dm.GraphUtil module

Generates histogram, stacked barplot and grouped barplot.

class dm.GraphUtil.**GraphUtil**
Bases: object

static gen_duration_histogram(*events*, *action*, *extensions*, *title*, *intervals*, *threshold*)
It generates histogram of ventilation lengths.

Parameters

- **events** – list of events
- **action** – show|save - for histogram showing or saving
- **extensions** – list of extensions used in case of action 'save'
- **title** – graph title
- **intervals** – list of intervals in minutes for which number of events is calculated
- **threshold** – value that is added to the interval in minutes or is subtracted from the interval to create value interval for a given column in a histogram

Returns None

static gen_grouped_barplot(*first_col*, *second_col*, *third_col*, *action*, *filename*)
It generates grouped bar chart.

Parameters

- **first_col** – deprecated
- **second_col** – data (values of an attribute) used to plot graph
- **third_col** – data (values of an attribute) used to plot graph
- **action** – show|save - for graph showing or saving
- **filename** – output filename

Returns None

static gen_stacked_barplot(*first_col*, *second_col*, *third_col*)
It generates stacked bar graph.

Parameters

- **first_col** – data (values of an attribute) used to plot graph
- **second_col** – data (values of an attribute) used to plot graph
- **third_col** – data (values of an attribute) used to plot graph

Returns None

9.14 dm.HTTPClient module

REST api client with Bearer authentization.

class dm.HTTPClient.**HTTPClient**(*host*, *port*, *end_point*, *method*, *verify_ssl_cert=True*)

Bases: object

Simple HTTP client.

authorize(*session_id*)

It sets authentization.

Parameters **session_id** – session identifier

Returns None

body(*body*)

It sets HTTP body.

Parameters **body** – HTTP body

Returns None

perform()

It performs HTTP request.

It creates HTTP connection, performs HTTP request, processes HTTP response and closes the connection.

Returns HTTP response and its content

9.15 dm.HeatMap module

Graph representing heat map.

class dm.HeatMap.**HeatMap**

Bases: object

static annotate_heatmap(*im, data=None, valfmt='{x:.2f}', textcolors=['black', 'white'], threshold=None, **textkw*)

It annotates a heatmap.

Parameters

- **im** – the AxesImage to be labeled
- **data** – data used to annotate; ff None, the image's data is used
- **valfmt** – the format of the annotations inside the heatmap; this should either use the string format method, e.g. "$ {x:.2f}", or be a matplotlib.ticker.Formatter
- **textcolors** – a list or array of two color specifications; the first is used for values below a threshold, the second for those above

- **threshold** – value in data units according to which the colors from textcolors are applied; if None (the default), the middle of the colormap is used as separation
- ****textkw** –

Returns

static heatmap(*data*, *row_labels*, *col_labels*, *ax=None*, *cbar_kw={}*, *cbarlabel=''*, ***kwargs*)
It create a heatmap from a numpy array and two lists of labels.

Parameters

- **data** – a 2D numpy array of shape (N, M)
- **row_labels** – a list or array of length N with the labels for the rows
- **col_labels** – a list or array of length M with the labels for the columns
- **ax** –

  a matplotlib.axes.Axes instance to which the heatmap is plotted; if it is not provided, current axes are used or

  a new one is created
- **cbar_kw** – a dictionary with arguments to `matplotlib.Figure.colorbar()`.
- **cbarlabel** – the label for the colorbar
- ****kwargs** –

Returns

9.16 dm.OpenWeatherOrg module

Gets information about weather from OpenWeather.org.

class dm.OpenWeatherOrg.**OpenWeatherOrg**(*precision=2*)
Bases: `object`

property api_key

weather_by_city_id(*quantities*, *start*, *end*, *city_id: int*)
It gets weather from city based on its identifier during certain time interval.

Parameters

- **quantities** – list of required quantities
- **start** – timestamp that denotes start of time interval
- **end** – timestamp that denotes end of time interval
- **city_id** – identifier of city

Returns weather data

weather_by_city_name(*quantities*, *start*, *end*, *name*)
It gets weather from a given city during certain time interval.

Parameters

- **quantities** – list of required quantities
- **start** – timestamp that denotes start of time interval

- **end** – timestamp that denotes end of time interval
- **name** – name of city

Returns weather data

weather_by_coordinates(*quantities*, *start*, *end*, *lat*, *lon*)
It gets weather from city based on its coordinates during certain time interval.

Parameters

- **quantities** – list of required quantities
- **start** – timestamp that denotes start of time interval
- **end** – timestamp that denotes end of time interval
- **lat** – latitude
- **lon** – longitude

Returns weather data

9.17 dm.Performance module

Calculates accuracy using standard approach and accuracy considering a tolerance interval.

class dm.Performance.**Performance**(*filename*)
Bases: object

simple()
It calculates standard accuracy.

Returns confusion matrix including number of events and accuracy; list of dates and times of events when a window was closed but window opening was detected (false positive); structure containing information from confusion matrix and sum of false positives, false negatives, true positives and true negatives

with_delay(*before*, *after*)
It calculates accuracy considering a tolerance interval.

Parameters

- **before** – number of seconds before start of an event
- **after** – number of seconds after start of an event

Returns confusion matrix including number of events and accuracy; list of dates and times of events when a window was closed but window opening was detected (false positive); structure containing information from confusion matrix and sum of false positives, false negatives, true positives and true negatives

dm.Performance.**count**(*z*, *val*)
Deprecated.

9.18 dm.PreProcessing module

Utils for date pre-processing.

class dm.PreProcessing.**PreProcessing**
Bases: object

OPEN_CLOSE_ATTR_NAME = 'open_close'

TIME_ATTR_NAME = 'measured_time'

TIME_STRING_ATTR_NAME = 'measured_time_str'

static check_start_end_interval(*items: list*, *time_attribute: str*) → None
It checks if lists of values have the same starting and end time.

Parameters

- **items** – list of values
- **time_attribute** – name of attributes that contains timestamp

Returns None

static cut_interval(*items: list*, *start: int*, *end: int*, *time_attribute: str*) → list
It cuts data to have the same start and end of an interval (starting and end time).

Parameters

- **items** – list of values
- **start** – timestamp from which data is cut
- **end** – timestamp to which data is cut
- **time_attribute** – shift of start and end of time intervals

Returns cut list of values

static db_name_maps(*devices: list*) → list
It gets a list of column names in database that is loaded from list of devices.

A column name is loaded from a file to assign a given value to the right column in database.

Parameters **devices** – list of information about devices

Returns list of column names

static download_data(*clients: list*, *devices: list*, *start: int*, *end: int*) → list
It downloads required data according to a list of devices.

Parameters

- **clients** – list of clients
- **devices** – list of information about devices
- **start** – timestamp from which data is downloaded
- **end** – timestamp to which data is downloaded

Returns list of downloaded data

static extract_items(*data*, *value_column*)
It prepares data.

Parameters

- **data** – dictionary of data
- **value_column** – name of column

Returns list of values and list of timestamps

static generate_data(*values: list*, *value_attribute: str*, *time_attribute: str*, *precision: int = 7*) → list

It modifies downloaded data so that an interval between two adjacent records was one second.

It modifies the data using linearization, it does not consider quantities or their course.

Parameters

- **values** – list of values
- **value_attribute** – name of attribute that contains data
- **time_attribute** – ame of attribute that contains timestamp
- **precision** – precision of calculation

Returns list of values that contains modified downloaded data (with step of one second)

static generate_open_close(*values: list*, *time_attribute_name: str*, *open_close_attribute_name: str*, *start: int*, *end: int*, *last_open_close_state: int*) → list

It modifies downloaded data so that an interval between two adjacent records was one second.

It modifies the data using linearization, it does not consider quantities or their course.

Parameters

- **values** – list of values
- **time_attribute_name** – name of attribute that contains timestamp
- **open_close_attribute_name** – name of attribute that contains state of window (open/closed)
- **start** – timestamp that denotes start of an interval
- **end** – timestamp that denotes end of an interval
- **last_open_close_state** – last recorded state of a window

Returns list of values that contains modified downloaded data (with step of one second)

static insert_values(*conn*, *table_name*, *values*, *maps*, *write_each*, *precision*)

It inserts values to a table.

Parameters

- **conn** – connection to a database
- **table_name** – name of table
- **values** – data related to event
- **maps** – list of column names
- **write_each** – number of events - 1 between two adjacent events that are written to a table
- **precision** – precision of inserted values

Returns None

static join_items(*items: list*, *time_attribute: str*) → list

It joins several lists of values into one list.

The resulting list contains only one attribute that includes time because times are the same. It contains various values that were included in various lists.

THe method has to be called only if passed list of values was checked using the method check_start_end_interval().

Parameters

- **items** – list of values
- **time_attribute** – name of attribute that contains timestamp

Returns list of values including all attributes and time

static ppm_filter(*data*, *ppm_limit=2000*)

It sets carbon dioxide concentration to None if its value is out of range.

Parameters

- **data** – dictionary of data
- **ppm_limit** – maximal admissible value of the concentration

Returns dictionary of data where excessive concentration is set to None

static prepare(*clients: list*, *devices: list*, *start: int*, *end: int*, *last_open_close_state: int*, *time_shift: int*)

It prepares data.

Parameters

- **clients** – list of clients
- **devices** – list of information about devices
- **start** – timestamp from which data is downloaded
- **end** – timestamp to which data is downloaded
- **last_open_close_state** – last recorded state of a window
- **time_shift** – time shift that is subtracted or added to start or end of an interval respectively.

Returns list of attribute names and list of values

static prepare_downloaded_data(*clients: list*, *devices: list*, *start: int*, *end: int*, *time_shift: int*, *last_open_close_state*) → list

It prepares downloaded data.

Parameters

- **clients** – list of clients
- **devices** – list of information about devices
- **start** – timestamp that denotes start of an interval
- **end** – timestamp that denotes end of an interval
- **time_shift** – time shift that is subtracted or added to start or end of an interval respectively
- **last_open_close_state** – last recorded state of a window

Returns dictionary of prepared data

static prepare_value_conversion(*value*)

It converts relative humidity to absolute and specific humidity.

Parameters **value** – data related to an event

Returns data related to an event including absolute and specific humidity

static rename_all_attributes(*items: list*, *devices: list*) → list
It renames all attributes in a list that contains list of values.

It renames individual attributes that is related to a given module. Each downloaded value from list of values with a defined name of attributes by default. New names of attributes are derived from list of devices.

Parameters

- **items** – list of values
- **devices** – list of information about devices

Returns list that contains a modified list of values including renamed attributes

static rename_attribute(*values: list*, *old_attribute: str*, *new_attribute: str*) → list
It renames an attribute in data from an original name to a new name.

Parameters

- **values** – list of values
- **old_attribute** – name of an original attribute
- **new_attribute** – name of a new attribute

Returns list of pairs of a renamed attribute and its value

static value_filter(*data*)
It checks if indoor relative humidity is in an admissible range.

If indoor relative humidity is out of range, it is logged.

Parameters **data** – dictionary of data

Returns dictionary of data

9.19 dm.SQLUtil module

Utils that contain simple SQL queries.

class dm.SQLUtil.**SQLUtil**
Bases: object

static select_interval(*table_name: str*, *start: int*, *end: int*, *columns: str*)
It selects given columns from a table on the basis of given time interval.

Parameters

- **table_name** – name of table
- **start** – timestamp that denotes start of a time interval
- **end** – timestamp that denotes end of a time interval
- **columns** – names of columns

Returns SQL query

static select_interval_size(*table_name: str*, *start: int*, *end: int*, *column: str*)
It finds out number of records on the basis of given time interval.

Parameters

- **table_name** – name of table

- **start** – timestamp that denotes start of a time interval
- **end** – timestamp that denotes end of a time interval
- **column** – name of column

Returns SQL query

static select_one_value(*table_name: str*, *measured_time: int*, *columns: str*)
It selects one record from a table on the basis of given timestamp.

Parameters

- **table_name** – name of table
- **measured_time** – timestamp that is used to select a record
- **columns** – list of columns that are selected

Returns SQL query

9.20 dm.Storage module

Storage that supports selection of events and their values or only selection of the values.

class dm.Storage.**Storage**(*filename: str*, *no_event_time_shift: int*, *table_name: str*)
Bases: object

static dw_columns_ordered(*con*, *start*, *end*, *columns*, *table_name*)
It gets several rows (interval) from a table and orders them.

Parameters

- **con** – connection to a database
- **start** – timestamp that denotes start of a time interval
- **end** – timestamp that denotes end of a time interval
- **columns** – list of columns
- **table_name** – name of table

Returns ordered rows (interval)

load_data(*con*, *start_shift: int*, *end_shift: int*, *column: str*)
It loads information about events. :param con: connection to a database :param start_shift: number of seconds added to start of event :param end_shift: number of seconds added to end of event :param column: name of column :return: dictionary of data

static one_row(*con*, *table_name: str*, *columns: str*, *timestamp: int*)
It gets one row (required columns) from a table.

Parameters

- **con** – connection to a database
- **table_name** – name of table
- **columns** – list of columns
- **timestamp** – timestamp that determines a required row

Returns one row (required columns)

read_meta()
It reads information about events.

Returns list of attributes describing events

static select_interval(*con*, *start*, *end*, *column*, *table_name*, *without_none_value=True*)
It gets several rows (interval) from a table.

Parameters

- **con** – connection to a database
- **start** – timestamp that denotes start of a time interval
- **end** – timestamp that denotes end of a time interval
- **column** – name of column
- **table_name** – name of table
- **without_none_value** – true if the result cannot contain null values

Returns several rows (interval)

9.21 dm.ValueConversionUtil module

Converts relative humidity to absolute humidity, relative humidity to specific humidity and ppm to milligrams per cubic meter.

class dm.ValueConversionUtil.**ValueConversionUtil**
Bases: object

CO_MOLECULAR_WEIGHT = 44.0095

static ah_to_relative_percent(*temp: float*, *ah: float*) → float
It converts absolute humidity to relative humidity.

Parameters

- **temp** – temperature
- **ah** – absolute humidity

Returns relative humidity

static co2_ppm_to_mg_m3(*co2*)
It converts ppm to milligrams per cubic meter.

Parameters **co2** – ppm

Returns milligrams per cubic meter

static rh_to_absolute_g_m3(*temp: float*, *rh: float*) → float
It converts relative humidity to absolute humidity.

Parameters

- **temp** – temperature
- **rh** – relative humidity

Returns absolute humidity

static rh_to_specific_g_kg(*temp: float*, *rh: float*) → float
It converts relative humidity to specific humidity.

Parameters

- **temp** – temperature
- **rh** – relative humidity

Returns specific humidity

9.22 dm.ValueUtil module

Utils for detection of simple events.

class dm.ValueUtil.**ValueUtil**
Bases: object

static delays(*events*, *delays_attr_name*)
It gets list of delays.

Parameters

- **events** – list of events
- **delays_attr_name** – name of attribute that denotes delay

Returns list of delays

static detect_sensor_delay(*values*, *window_size*, *threshold*)
It detects sensor delay.

Parameters

- **values** – list of value
- **window_size** – size of a window in seconds
- **threshold** – threshold in seconds

Returns index of value when a decrease occurred

static detect_window_action(*values_count: int*, *actual_index: int*)
It detects a window action.

Parameters

- **values_count** – count of values
- **actual_index** – actual index

Returns window action

static events_duration(*events*, *max_duration*)
It gets durations of events that lasted at most given time.

Parameters

- **events** – dictionary of events
- **max_duration** – maximal duration of event

Returns list of durations of events

static window_event_value(*measured: dict*, *value_index: int*, *timestamp: int*, *precision: int*)
It gets information about event when a window was open.

Parameters

- **measured** – dictionary of measured values
- **value_index** – index to a dictionary
- **timestamp** – timestamp when an event occurred
- **precision** – precision of output values

Returns information about event when a window was open

static window_no_event_value(*values: tuple*, *precision: int*)
It gets information about event when a window was not open.

Parameters

- **values** – measured values of quantities
- **precision** – precision of output values

Returns information about event when a window was not open

9.23 dm.WundergroundCom module

Weather data extraction from weather.com.

class dm.WundergroundCom.**WundergroundCom**(*precision=1*)
Bases: object

actual_weather(*lat*, *lon*)
It gets information about current weather in a locality.

Parameters

- **lat** – latitude of a locality
- **lon** – longitude of a locality

Returns information about current weather

property api_key

weather_by_coordinates(*quantities*, *start*, *end*, *lat*, *lon*)
It gets information about weather in a locality at required time interval.

Parameters

- **quantities** – deprecated
- **start** – timestamp that denotes start of time interval
- **end** – timestamp that denotes end of time interval
- **lat** – latitude of a locality
- **lon** – longitude of a locality

Returns information about weather

9.24 Module contents

CHAPTER

TEN

AUTHORS

- Jan Kořenek <korenek@fit.vutbr.cz>
- Klára Nečasová <klarksonn@gmail.com>
- Peter Tisovčík <itisovcik@fit.vutbr.cz, mienkofax@gmail.com>

PYTHON MODULE INDEX

d

INDEX

D

E

F

G

H

I

S

T

U

V

W