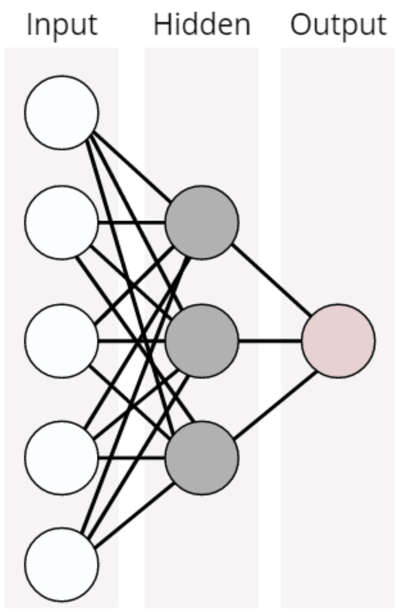
University College
of Southeast Norway

www.usn.no

FMH606 Master's Thesis 2018

Energy and Environmental Technology

Predicting weather using ANN with free open weather data in python



Erik Boye Abrahamsen

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

University College of Southeast Norway

www.usn.no

Course: FMH606 Master's Thesis, 2018

Title: Predicting weather using ANN with free open weather data in python

Number of pages: 27

Keywords: Python interface, Weather prediction, Auto-regressive neural network, ANN, IoT, Big-Data

Student:	Erik Boye Abrahamsen
Supervisor:	Bernt Lie, Ole Magnus Brastein
External partner:	dr. Beathe Furenes
Availability:	Open

Summary:

Predicting the weather is important for a lot of fields including agriculture, construction and hydro-power and flood management. Currently mechanistic meteorology predictions are generated using heavy computing based 3D Navier-Stokes models. Therefore, it is of interest to develop models that can predict weather conditions faster than traditional meteorological models. The field of machine learning has received much interest from the scientific community. Due to its applicability in a variety of fields, it is of interest to study if the use of artificial neural networks can be a good candidate for prediction of weather conditions. Machine learning methods benefit from large datasets. A python interface was developed to make it easier to obtain weather data from free sources, the python interface works well, but is more user-friendly when used with Met supplier compared with Netatmo supplier. Four separate models where trained to predict the temperature 1, 3, 6 and 12 hours ahead. In the first experiment, only temperature was used as input to the networks. This constitutes an auto-regressive neural network(ARNN). In the second experiment, precipitation data was introduced into the network, forming an autoregressive neural network with exogenous inputs (ARX-NN). The results show that the inclusion of precipitation had a negligible effect on accuracy for temperature prediction. Out of the four model types, 1-hour prediction has the best prediction results for both the AR-NN and the ARX-NN.

Preface

I would like to thank everybody in the world.

Porsgrunn, 15.05.2018

Erik Boye Abrahamsen

Contents

Nomenclature

ANN – Artificial neural network

API – Application programming interface

IoT – Internet of things, a term used for gadgets connected to the internet

ML – Machine Learning

MLP – Multilayer perceptron

1 Introduction

Machine Learning (ML) and use of IoT (Internet of Things) and Big Data is receiving increased interest from the industry. To experiment with Machine Learning, it is necessary to have access to “Big Data”; access to such data is not always easy to get from the industry. To this end, it is of interest to develop access to “Big Data”, and build up competence in Machine Learning – with application to an interesting topic.

1.1 Background

Predicting the weather is important for a lot of fields including agriculture, construction and hydro-power and flood management. Currently mechanistic meteorology predictions are generated using heavy computing based 3D Navier-Stokes models and can easily take 12 hours on fast computers[1]. In Norway weather data is publicly available for free from weather stations around in Norway, installed and maintained by meteorological institute of Norway(Met) [2], along with Met there is Netatmo [3], another data supplier that gives out their weather data for free. The difference between the two is that Met uses meteorological standard for placement of sensors, while Netatmo is an internet of things(IoT) device purchased at your local hardware store that is connected to the internet, installed and maintained by you. Both suppliers give an HTTP Get/Request(REST) [4] Application Programming Interface(API) that gives access to their data. Met supplies both historical data and real-time, while Netatmo supplies real-time only. With the access to large quantities of data an interesting approach would be to use novel Machine Learning(ML) algorithms such as an artificial neural network(ANN) to predict the weather conditions for a given geographical location, and with such large quantities compensate for the lack of complex meteorological models.

1.2 Task description and Objectives

The tasks and objectives are listed below, and the original task description is given in Appendix A – Task description.

Overall objectives:

- Research different suppliers for weather data and their APIs e.g. Netatmo
- Develop an API for accessing these data from Python, Julia or MATLAB
- Choose a reduced set of data points, and set up logging of the data with storage in a file if historical data is not available.
- Based on available data, set up a system for machine learning. If y(t) is measurements at time t, then use machine learning to find a mapping F for y(t) = F(y(t-1), y(t-2), …, y(t-T)) for various values of total horizon T and various time steps dT.
- Validate the model, and compare your results to those from a meteorological model. The model should preferably give a distribution of predictions.

After all it boils down to two main objectives, the first is developing and document the data interface used to obtain data from more than one supplier. Then it is the building and prediction of machine learning model, the two are explained in more detail below.

1.2.1 Data Interface

The data interface should help give access weather supplier APIs, it should be well documented and developed in python. The data retrieved should be from all stations within a given rectangle or for a specified station or latitude and longitude coordinates. The interface should have the same commands for all suppliers and supply the data neatly formatted. Handling of the different suppliers should be hidden from the user, it should also be possible to add more suppliers if needed. The user should only need to specify what type of data is needed, from where and from when and should also have the option to save the data to csv file.

1.2.2 Building and prediction of machine learning model

For the ML model an artificial neural network should be built. Using python as programming language, there are several machine learning libraries available, like Scikit-learn [5], Tensorflow [6], Theano [7], Keras [8]. For building the model, Keras with Tensorflow backend has been chosen as it is an easy solution to build different network structures and tune different hyperparameters. The data used to train, tune and test should be separated, where training data consists of 60% of the total data, while the remaining 40% is split equally among tuning and testing. The tuning data is only to be used for tuning the hyperparameters, and when satisfied with the tuning the test data is only be used once in the model to give a true measure of the tuned model. The ML model should predict weather parameters for 48 consecutive hours from 1 to 12 hours ahead.

1.3 Previous Work

Hayatiet.al [9],studied multilayer perceptron (MLP) neural networks trained and tested on ten years of metrological data (1996-2006). A training goal of 10^{-4} was used. The network structure consisted of three layers with a logistic sigmoid activation function in hidden layer and linear function in the output layer. Seven weather variables were used in the study; dry temperature, wet temperature, wind speed, humidity, pressure, sunshine and radiation. The inputs were normalized and used to predict dry air temperature in intervals of 3 hours for a total of 8 predictions per day. The error was calculated using mean absolute error (MAE). Smith et.al [10] focused on developing artificial neural network models to forecast air temperature at hourly intervals from one to 12 hours ahead. 30 models were calibrated for each interval, to study the effect of randomized initial weights on test set prediction accuracy. The network structure consisted of three fully connected hidden layers that used Gaussian-, Gaussian complement and hyperbolic tangent activation functions. The input data was linearly transformed to the range 0.1 to 0.9 and consisted of five weather variables: temperature, relative humidity, wind speed, solar radiation and rainfall. Later seasonal parameters were introduced as input which improved model accuracy.

1.4 Report structure

Chapter 1: Introduction describes the problem description, objectives and previous work

Chapter 2: Python interface is all about the design and goal of the interface and how to use it.

Chapter 3: Machine learning model contains the information about what an artificial neural network is, what type of network used, how it was tuned and about the dataset used in the experiments described.

Chapter 4: Results and discussion contains the results of the experiments along with a discussion of them

Chapter 5: Conclusion.

2 Python Interface

As big data is more available to everybody online and cheap sensors with access to internet is more accessible and user friendly the need for better ways to obtain similar data increases. Here it is proposed and designed a way to make that possible with weather data from publicly available weather data.

2.1 Overview

The main goal of the interface is to provide the user with a way to obtain data from several weather data APIs that is hidden under the same structure. Different APIs have different commands and need different inputs for the same service they provide, this interface will act as a translator and transform your request into the right format, ask the specified data supplier, obtain the data and deliver it to the user. Figure 2-1 shows a general overview of the interface and information flow from user to interface, interface to supplier and then back.

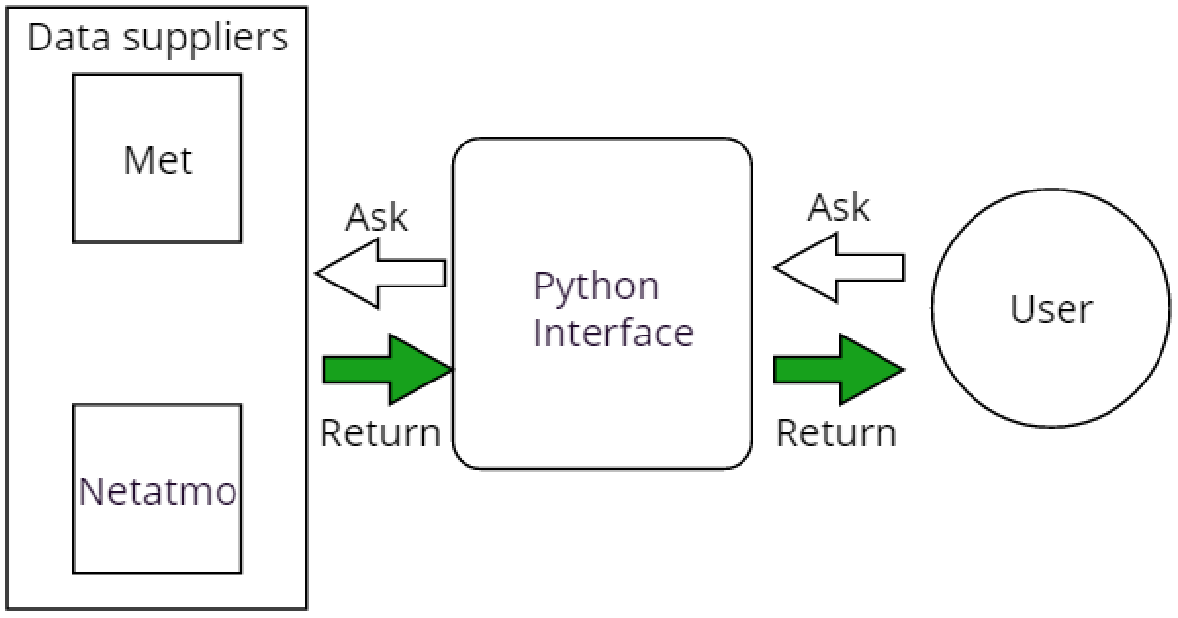


Figure 2-1: Shows the general big picture of the Python interface. The white arrow is the request and the green arrow shows a successful request and then is returning the data.

2.1.1 Design

The interface is built in Python and is a library that contains several classes in an object-oriented structure. Figure 2-2 shows the class layout of the interface and their interaction with each other. The user uses the class. The Client class holds the information about the client, like authentication token, which data supplier it is for, the Geometry class makes the specified square around the given coordinates. In the Weather class, the data to be retrieved can be specified. In the Usefuls class it is some functions that is useful to splitting dates into separate items or compact them again for usage in the Service class, this class is responsible for giving the right commands to the right supplier classes (Met and Netatmo).

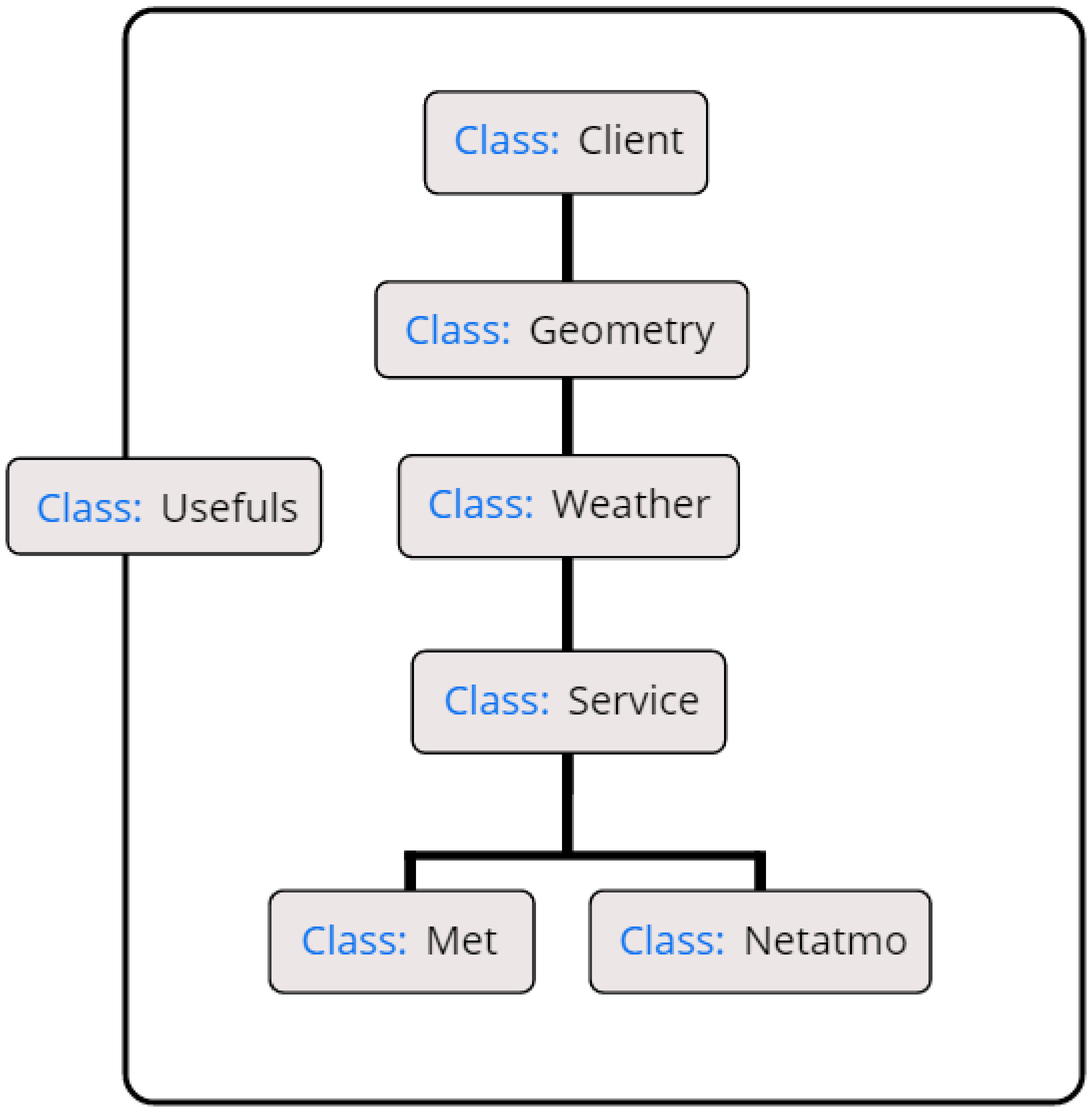


Figure 2-2: Shows the class structure of the interface. The usage of the interface is described under 3.2 Usage and in Appendix B - Python Interface Documentation

Data returned should be formatted into the same way for all the different weather variables. It will be combined in a Pandas[11] data frame which has the possibility to be saved to a csv file. Table 2-1 Shows how the returned data frame can look like for Met data frame, Data value is the measurement at the given station whose number is saved in Station number, timestamp is when it starts to when it ends, notice it ends at End date-1, this is because the date constraint goes from start date up to end date, not including.

Table 2-1: Shows how the returned data frame can look like for Met data frame

Station number	***Timestamp***	***Data value***
#####	Start date	0.3
#####	Start date +1	0.1
#####	Start date +2	None
…	…	…
#####	End date-3	0.2
#####	End date-2	0
#####	End date-1	None

2.2 Usage

Netatmo and Met are different, Netatmo supplies only live data so there is no need for a date constraint, while Met supplies historical data so a date is needed. See appendix B for more information about documentation and usage.

3 Machine Learning model

Artificial neural networks(ANN) have existed in various forms since the 1940s [12], but have received interest in recent years, largely due to the success of Deep Learning algorithms[13]. An artificial neural network is a layered collection of several densely connected units called artificial neurons, each connection has a numerical weight. A single artificial neuron produces a linear output that is simply a weighted sum of inputs, plus a bias, the output is then passed through a function called an activation function. The activation function decides if the output is transmitted to the connected units. A single artificial neuron is shown in Figure 3-1, where *X1* to *Xn* are the inputs, *b* is the bias, *u* is the output from the neuron and *y* output from the activation function, *g*.

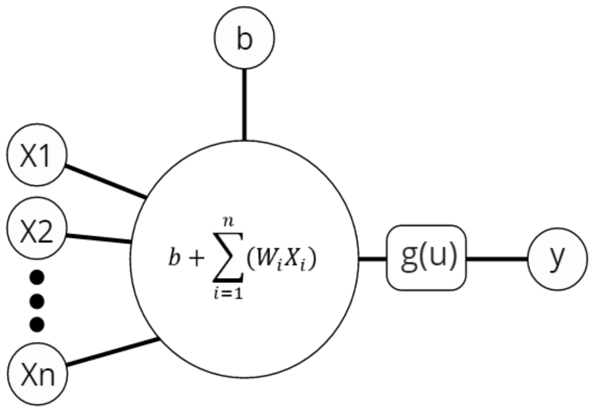


Figure 3-1: Single neuron

Being layered and densely connected means that each unit is connected to all the units in the next layer. Each layer in an artificial neural network can be large with an even larger number of coefficients, this allows the network to fit complex non-linear systems. This descriptive power that comes from this complexity is the reason ANN models can adapt to a large variety of systems. Figure 3-2 shows an illustration of a layered and densely connected artificial neural network with one input layer with five inputs, one hidden layer with three hidden units and one output layer with one output node.

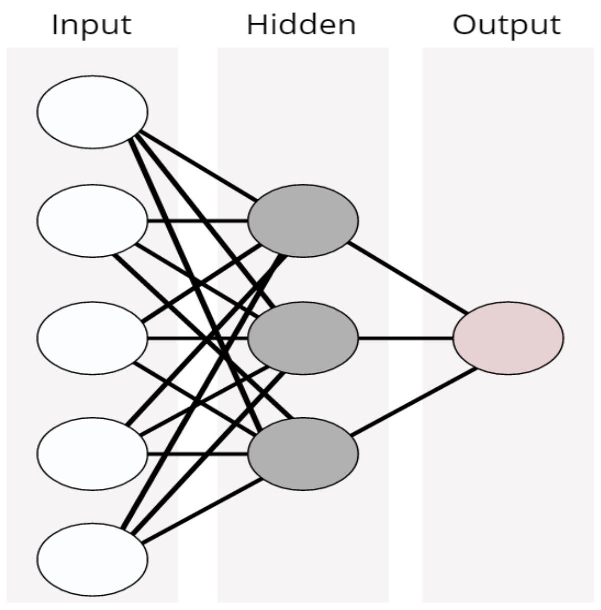


Figure 3-2: Layered and densely connected artificial neural network. Input layer, hidden layer and output layer is illustrated as rectangles.

The activation function takes the decision whether to let the signal pass or not, if it does it transforms the signal linearly or non-linearly, depending on the choice of function. The activation functions also squish the output, since the output of a unit can be large it is useful to restrict the output. Figure 3-3 shows some common activation functions and what their squish range is.

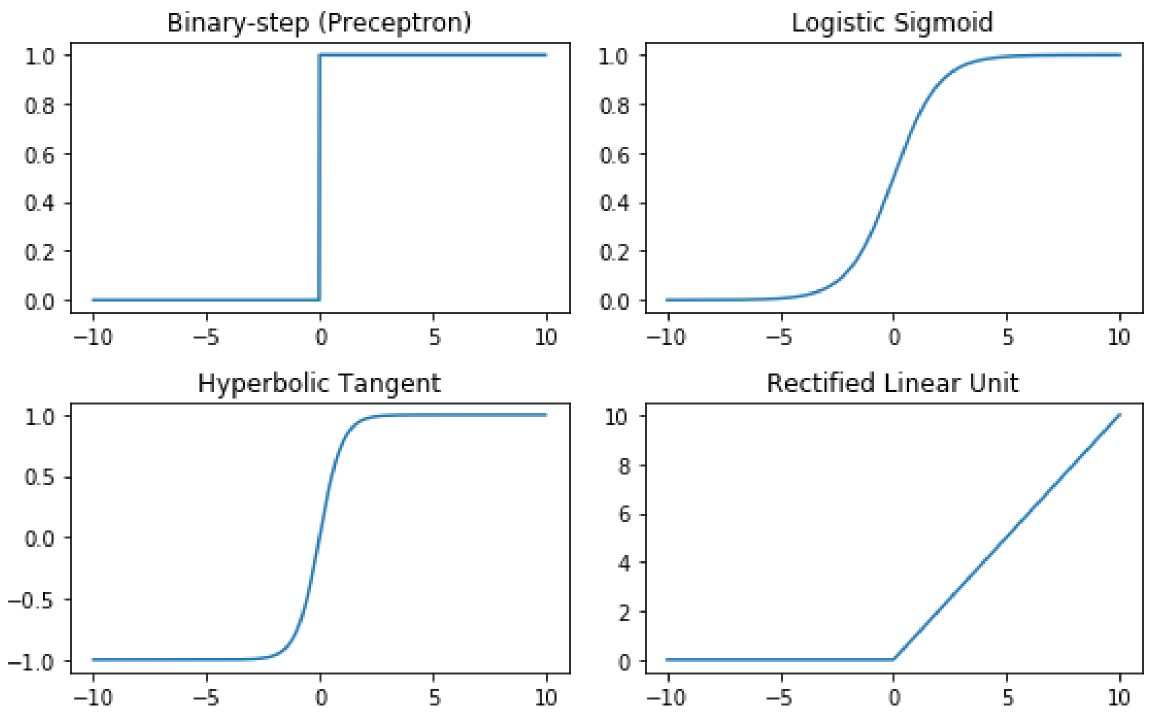


Figure 3-3: Common activation functions. Binary-step (top left) either 0 or 1, Logistic Sigmoid (top right) range 0.0 – 1.0, Hyperbolic Tangent (Bottom left) range -1.0 to 1.0, Rectified Linear Unit(ReLu) (bottom right) either 0 or x.

When training an artificial neural network, the goal is to usually minimize the loss function over the training set. This is essentially parameter optimization, but usually due to the high number of parameters to optimize it is unlikely to find a global solution, therefore a solution that is “good enough” is usually preferred. Training or optimizing means to make something a maximum or minimum this is an iterative process where each iteration is called an epoch [13]. To validate the network each epoch the difference (y - y*) from desired output (y) called a label, and network output (y*) is converted into a metric, known as the loss function, a common metric is the mean squared error(MSE) eq.1.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^*)^2 \qquad \text{(eq.1)}$$

A popular optimization method is gradient decent. The slope of the loss function at any point in the training process is the partial derivative with respect to the weights. When minimizing, each iteration takes the partial derivative at a point, move a small step called the learning rate in the opposite direction of the gradient and repeat until all epochs are done or local minima found. The learning rate and number of epochs is called hyperparameters and should be tuned for each different network case[13].

3.1 Dataset

A total of 32 000 hourly weather data was gathered from [14] using the python interface, at Station SN30255 in Porsgrunn, Norway with coordinates latitude: 59.091 and longitude: 9.66. The weather variables (called features) used in the models, were air-temperature and precipitation. ANNs don not handle missing data very well. A challenge was to obtain data from the same period and same hours. It would be possible to take different hours or data from 2016 and 2017, but the network would then need all future data to be structured in the same way before used for a prediction. Two periods were used together, the first in 2016 was from 01.02.2016 up to and including 31.12.2016, and 2017 was from 02.01.2017 up to and including 31.12.2017. The period was originally from 2016 and 2017, but in 2017 the first month of data was not available at the station. So, for consistency the first month of 2016 was also removed. The data is a series each 1 hour apart. Preparing the data to be easily usable for different predictions ahead in time while still maintaining correct labels was done using a shift to the “right” method illustrated in Table 3-1.

Table 3-1: Illustration of a right shift to make each hour previously used as input data to match with current labels

Label (y)	*1-Hour Shift (X_1)*	*2-Hour Shift (X_2)*	*3-Hour Shift (X_3)*	*…*	*n-hour shift (X_n)*
D_1	D_2 @Label	D_3 @Label	D_4 @Label	…	D_m @Label
D_2	D_3 @Label	D_4 @Label	D_5 @Label	…	D_{m-1} @Label
D_3	D_4 @Label	D_5 @Label	D_6 @Label	…	D_{m-2} @Label
…	D_5 @Label	D_6 @Label	D_7 @Label	…	D_{m-3} @Label
D_m	D_{m-1} @Label	D_{m-2} @Label	D_{m-3} @Label	…	D_{m-n} @Label

When extending the hourly predictions, the lowest data disappears for the other columns, so the larger the horizon gets the more rows must be deleted to maintain a constant width and height. Table 3-2 shows an example of what happens with the bottom data when right shifting. Column y is the label and original data, column X_1 is a 1-hour shift, the total number of usable data drops by 1 (from 7 to 6 and so on.). The number of usable data (N_u) is the difference between total number of data(N_T) and number of hours back (N_B), $N_u = N_T – N_B$, when the data is hourly split and right shifted.

Table 3-2: Illustration of what happens to bottom data when right shifting.

y	*X₁*	*X₂*	*X₃*	*X₄*	*X₅*	*X₆*
7	6	5	4	3	2	1
6	5	4	3	2	1	X
5	4	3	2	1	X	X
4	3	2	1	X	X	X
3	2	1	X	X	X	X
2	1	X	X	X	X	X
1	X	X	X	X	X	X

In Figure 3-4 a plot of the temperature data acquired from 01.02.2017 to 31.12.2017 is plotted. This plot shows how the data is varying over the year, as expected the temperature is highest in the middle and lowest on the start and end of the year. In all the experiments the dataset was split into three parts, 60% used for training, 20% For hyperparameter tuning and 20% for testing, the testing set is completely independent of the training and tuning.

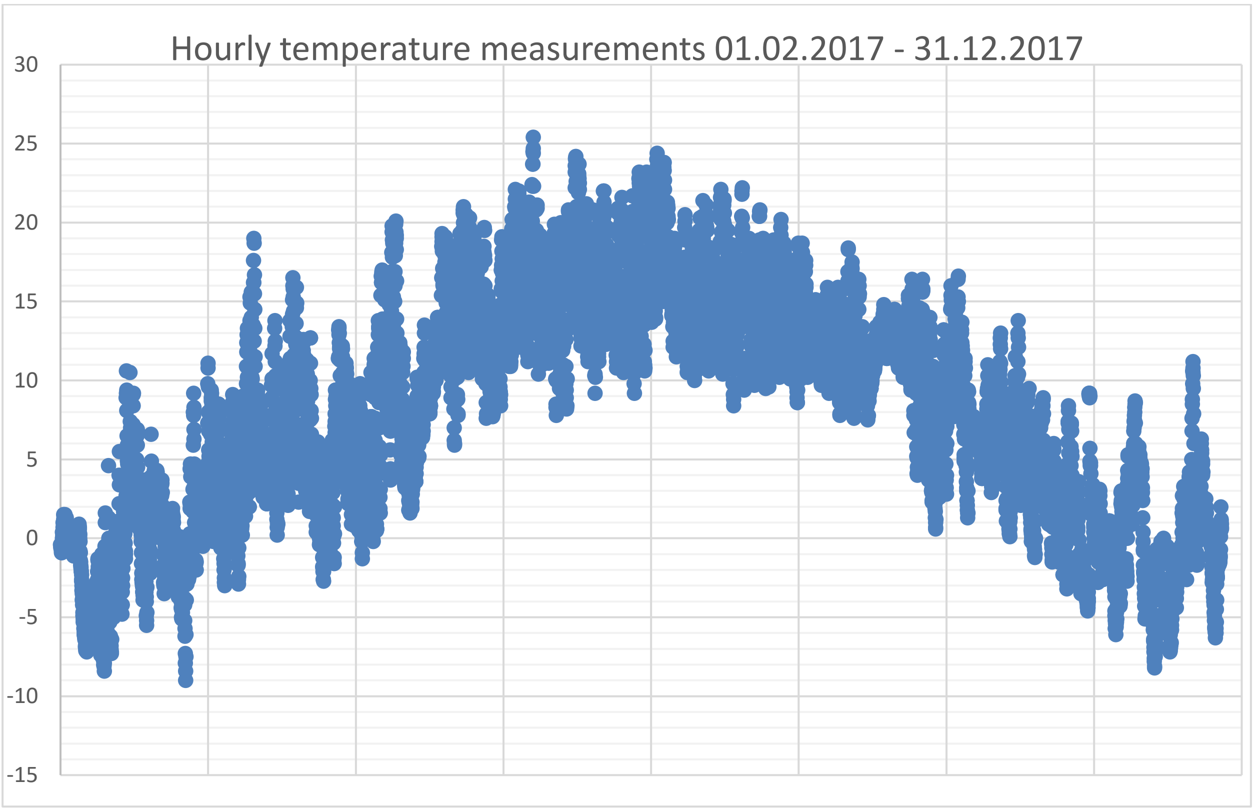


Figure 3-4: Plot of hourly temperature data for the period 01.02.2017 – 31.12.2017

3.2 Experiments

In all experiments the goal was to predict 1-, 3-, 6- and 12-hours ahead, compare them with each other. In all cases the activation functions for hidden layers was rectified linear unit and linear for the output layer, ReLU has been shown to enable better training in 2011 compared to the more widely used Sigmoid function [15]. Separate models were created, one for each 1-, 3-, 6- and 12-hour predictions in all cases the input data was normalized, then the output was denormalized to achieve desired results in readable units. To test the different models, 48 consecutive hours is set to be predicted by the different models.

3.2.1 Experiment 1: Temperature AR-ANN

The first experiment was set up to predict air-temperature by using historical temperature data only. Figure 3-5 shows an illustration of AR-ANN to predict 1 step ahead.

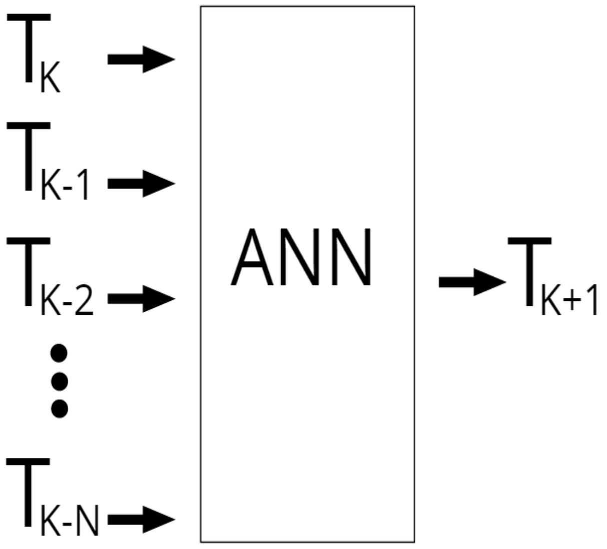


Figure 3-5: AR-ANN to predict 1 step ahead.

3.2.2 Experiment 2: Temperature ARX-ANN

In the second experiment, air-temperature and precipitation were used as input to predict the air-temperature. Figure 3-6 shows an illustration of ARX-ANN to predict 1 step ahead, using two input features.

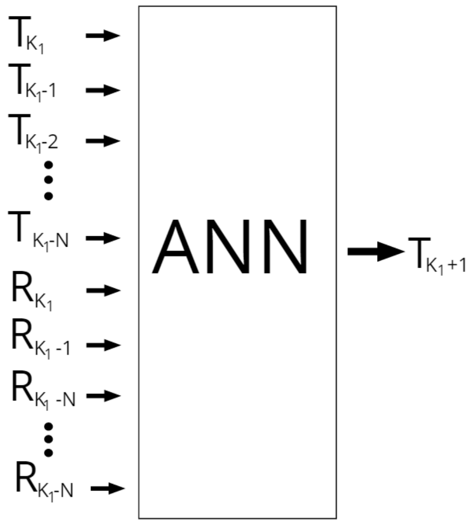


Figure 3-6: ARX-ANN

4 Results and discussion

During the thesis work several other thigs that cannot be documented very well has taken place this includes learning about machine learning, how to use different machine learning frameworks like Tensorflow, Keras and scikit-learn. Building a database and learning SQL for storing and retrieving weather data and started developing the interface in Julia as well, this was discarded in the final design for the interface, but nonetheless consumed a lot of time.

4.1 Python interface

The interface is easy to use, delivers data with simple lines of code as specified, however Met is the supplier that is most suited for delivering data with a Pandas frame that has access to saving methods. This is because Met delivers the specified data in a single nested list. While Netatmo delivers all the data, it does not care if you specify the wanted data according to their API, and the delivered data is in several listed dictionaries and lists, and because of this, sorting for the correct weather data is painfully difficult and have not been done in such a way that the returned data is formatted like in Mets case. The data sorted and delivered with the interface from Netatmo is somewhat sorted in a nested list with timestamp, station id and the different weather variables. To fix this problem a solution might be to do a hard search for specific names like temperature or pressure, one problem with this is might be that the variable values might get mixed due to the way Netatmo delivers data. For instance if searching for Temperature, a dictionary with two elements (one dictionary and one list) is returned, in the dictionary element there is another list with two values. These values correspond to the other list delivered which is strings of weather variables temperature and humidity, already there it is 5 nests for obtaining one value. But when doing this for 10 stations and running 5 nested loops for each variable Netatmo delivers one may easily run into some problems with computing power or time. For obtaining data it is recommended to use Met instead of Netatmo. Met is easier to use, delivers data fast and with great quality. The handling of errors when making a request could have been optimized better, with checks on the parameter inputs, URL lengths generated and such, if your square gets too big and tries to get over 50 stations met can’t deliver it, the URL would be too long. This could be fixed if a batch system was introduced to make sure the length was in order.

4.2 Experiment 1: AR-ANN

The first experiment used historical temperature data to predict the temperature ahead 1, 3, 6 and 12 hours. The predictions of the 48 consecutive hours to be predicted is shown in Figure 4-1. It is worth noting the sudden change in measured value from time 36 to 42, perhaps the sun had been dazzling that day and some clouds got in the way. The shorter models (1 - and 3 hour) almost starts oscillating as a response to this rapid change because they want to keep the error down even when they are way off, but they are still closer to measured data than the longer models, even when they seem to make more sense. At time 8 to 20, the 12-hour model is a lot lower than the 1 -,3 - and 6-hour models a reason for this might be that the models respond to data given and the 12-hour model uses data that is 12 hours prior to the measurement. So, at time 7 it started increasing then at time 15 it flattens out, probably because it used the data at time 3 and it saw that it started to turn so it better slow down. All the models seem to have a problem with the turning points, a reason for this might be that the models have not generalized well enough, maybe more data and better tuning can help.

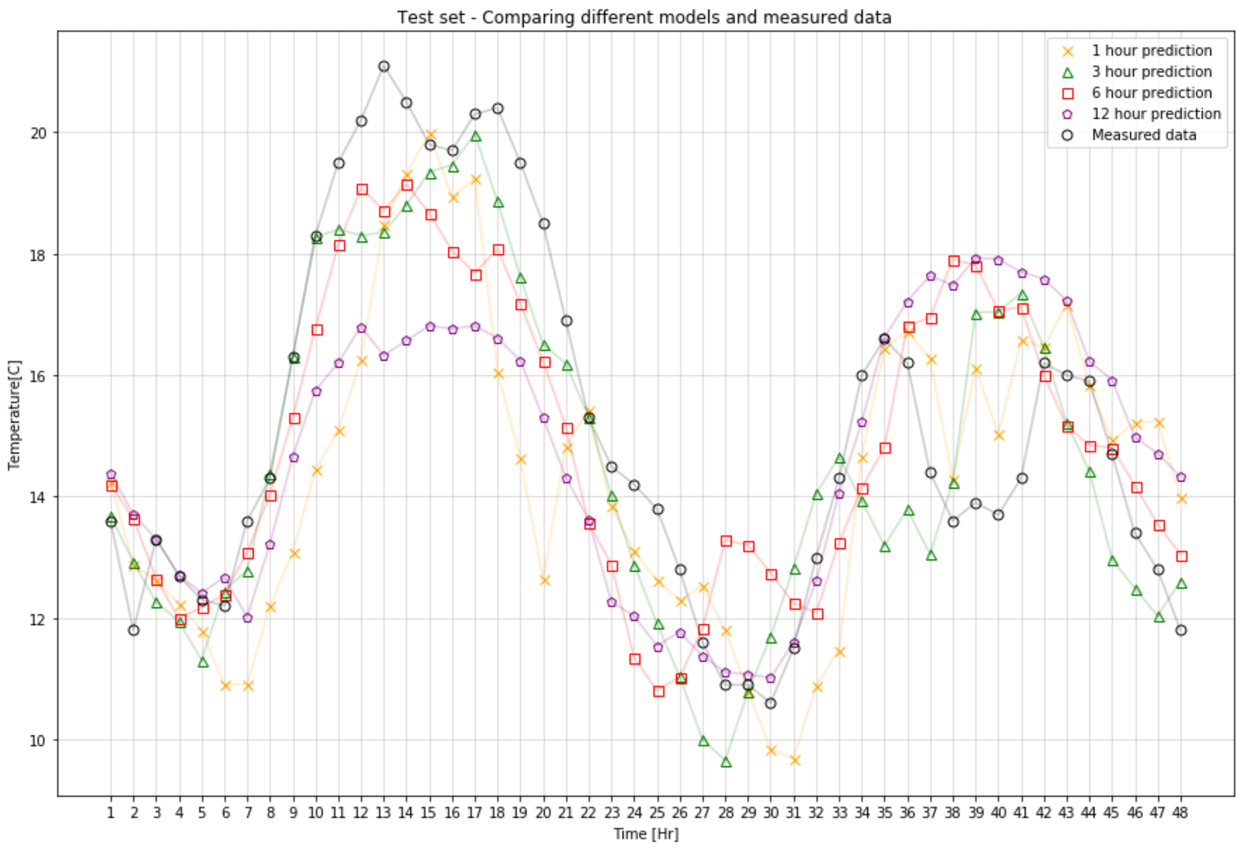


Figure 4-1: Shows the measured and the 4 different models.

Table 4-1 shows the hyperparameters with test error summarized. The test error is calculated when the test data is normalized, so the error is normalized as well. The number of layers in 12-hour prediction is twice as many as in the other models, the horizon is 169 hours (7 days) compared to 6-hours with 48 hours, the learning rate is also significantly lower, the reason for this is because it gave a better error than one, two or three layers, the low epochs and low learning rate is because in instantiating and training different models, it quickly became overfit and as a result this is an example of slowing down the training.

Table 4-1: Hyperparameters for each model

Model	*Test error*	*Epochs*	*Layer structure*	*Regression horizon*	*Learning rate*
1 Hour	0.0101	810	17, 12	24	0.01
3 Hour	0.0318	150	30,20	48	0.01
6 Hour	0.0608	1000	38,24	48	0.001
12 Hour	0.0894	500	112,75,50,34	169	0.0001

Figure 4-2shows a better description of the error in degree Celsius for the 48 hours predicted. It is calculated as the difference between measured value and predicted value, this means that the error that is negative is overshooting and error that is positive is undershooting. In Figure 4-2 some things to note, as mentioned earlier, due to the rapid change in measured data at 36 to 42 the spikes and dip in the 1 – hour model and the huge dip for the 6- and 12- hour models make sense. The models undershoot more than they overshoot and the reason can be poor generalization or poor tuning for the models.

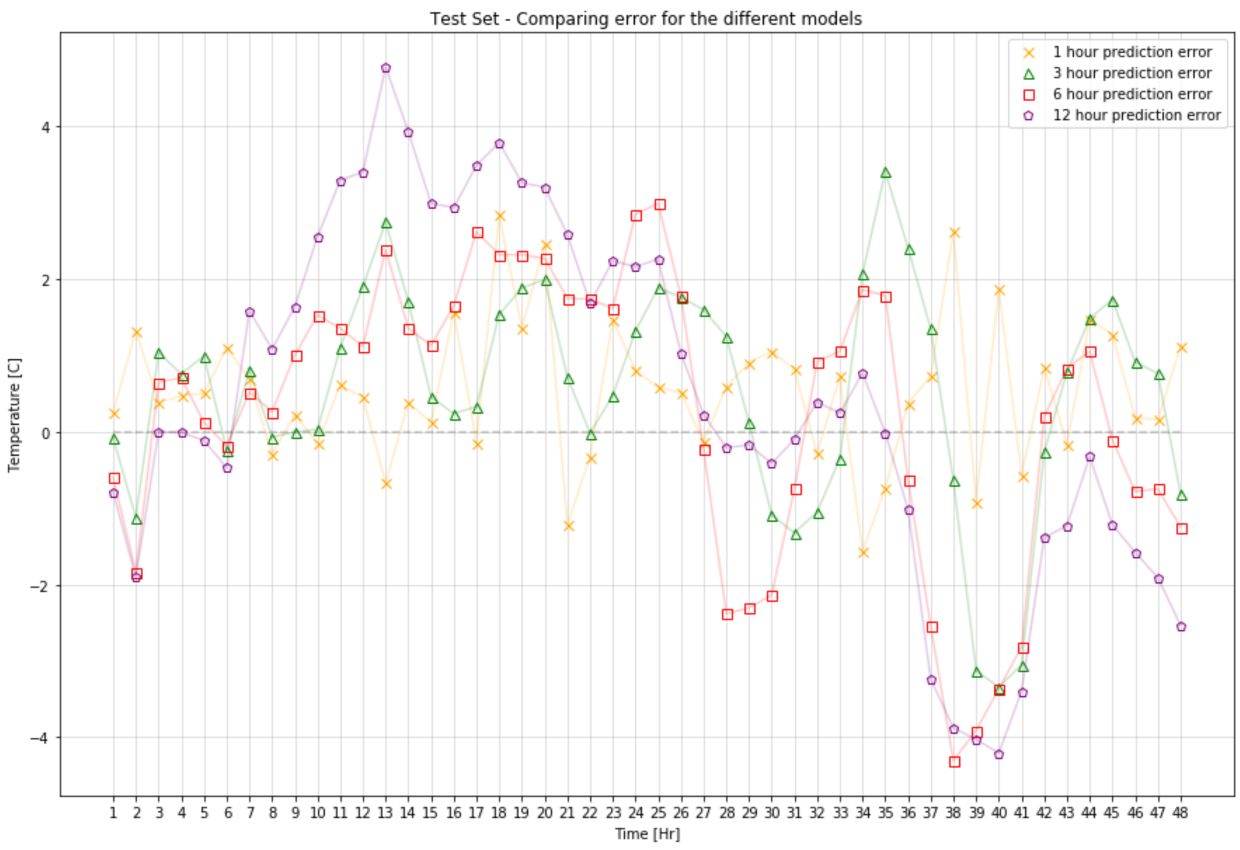


Figure 4-2: This figure shows how many degrees Celsius each point for each AR model was off for the 48 hours predicted. It is calculated as the difference between measured value and predicted value, this means that the error that is negative is overshooting and error that is positive is undershooting.

Table 4-2 shows some notable selections in hyperparameter searching methodology for 1-hour prediction. First a 3-hour horizon was used because as a human you would not need more than three hours of data to make a good enough prediction for 1 hour ahead in time and since a computer can handle a lot more data at a time, a 24-hour horizon is tested against the same structure and it has a better training loss and tuning loss. Picking the number of hidden layers is crucial based on what type of result you are searching for, one hidden layer can approximate any continuous function, while two hidden layers can approximate any arbitrary function. One layer should be sufficient, but two layers will speed up the learning process. One up to three hidden layers were tested and as shown in table 2 three layers gives around the same loss as in two layers therefore a two-layer model is picked. The number of units in each layer is based on some rule of thumb and there is generally no real correct way of determining them, here the three rules have been followed:

1) The number of hidden units in each layer should be between the number of inputs and number of outputs
2) The number of hidden units in each layer should be (2/3)*(number of inputs + number of outputs)
3) The total number of hidden neurons in all layers should be less than 2*(number of inputs) [16].

Table 4-2: 1-hour prediction, selected hyperparameter search

Exp. No.	***Epochs***	***Regression horizon***	***Layer structure***	***Learning rate***	***Training set loss***	***Tuning set loss***
1	1000	3	3	0.001	0.0144	0.0109
2	1000	24	3	0.001	0.0125	0.0105
3	1000	24	17	0.01	0.0116	0.0101
4	1000	24	17,12	0.01	0.0113	0.0100
5	1000	24	17,12,9	0.01	0.0113	0.0101
6	810	24	17,12	0.01	0.0113	0.0100

4.3 Experiment 2: ARX-ANN

In Figure 4-3 the predictions for each model in ARX-ANN is shown. Looking at area around the sudden change in measured value, at time 36 to 42. The shorter models (1 - and 3 hour) seems damped with the extra data, the 1-hour model predicts and reacts nicely to turns, however the 3-hour model also has some problems following along, but it is expected that the most accurate model would be the 1-hour model then the models would gradually get less accurate than the previous one. At time 25 the 3-hour model reaches its all-time low and is the lowest point, it the model reacts too strong to small changes in the data. At time 8 to 20 is a great example of how the models decrease in accuracy when the prediction horizon increases, on top is 1-hour model, then its 3-hour, then 6-hour and last 12-hour model. The error around turning points look better, which can be expected with a lower accuracy, the model does not react faster but, it reacts stronger.

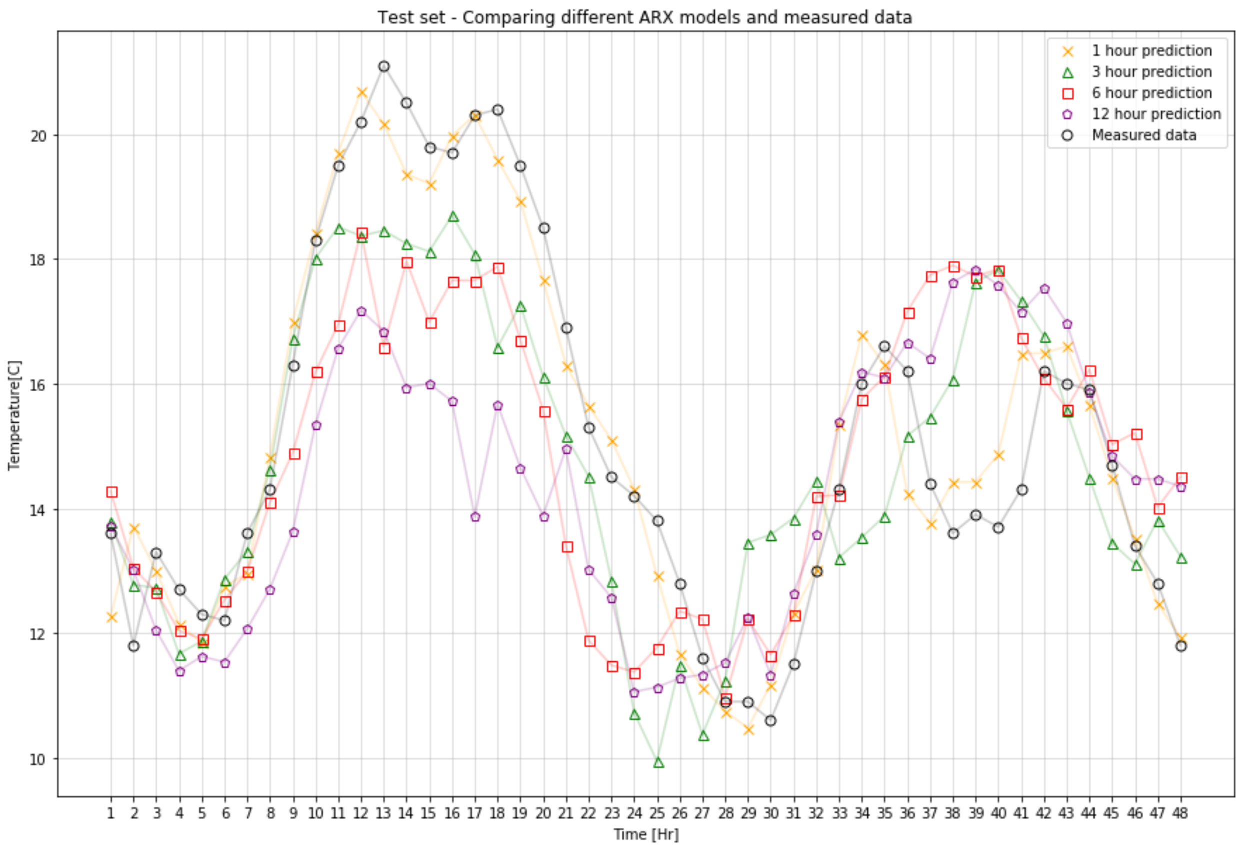


Figure 4-3: Shows the measured and the 4 different ARX models.

Table 4-3 shows the hyperparameters with test error summarized. The test error on the ARX models are higher than on the AR models, this is unexpected and the reason is most likely poor tuning of the models. The number of layers in all models has been set to 2-layer models, this is because any arbitrary function can be modeled using two hidden layers. [16] The hidden structures are based on the horizon so the reason they are similar, except for 6-hour model, which got better tuning with hidden structure as (17,12) instead of (33,12).

Table 4-3: Hyperparameters for each ARX model

Model	***Test error***	***Epochs***	***Layer structure***	***Regression horizon***	***Learning rate***
1 Hour	0.0133	500	17, 12	24	0.01
3 Hour	0.0653	500	33, 23	48	0.001
6 Hour	0.0946	500	17, 12	48	0.01
12 Hour	0.0991	300	17, 12	24	0.001

The error on each 48-hour predicted for each model is shown in Figure 4-4 Some things to note, as mentioned earlier, due to the rapid change in measured data at 36 to 42, the spikes and dips in the 1 – hour model is gone, or damped. However, the 3-hour model has accompanied the 6-hour and 12hour model and the reason for this can be that the model seem to react strongly to small changes in the data. All the models also undershoot more than they overshoot, and the reason can be poor generalization or poor tuning for the models.

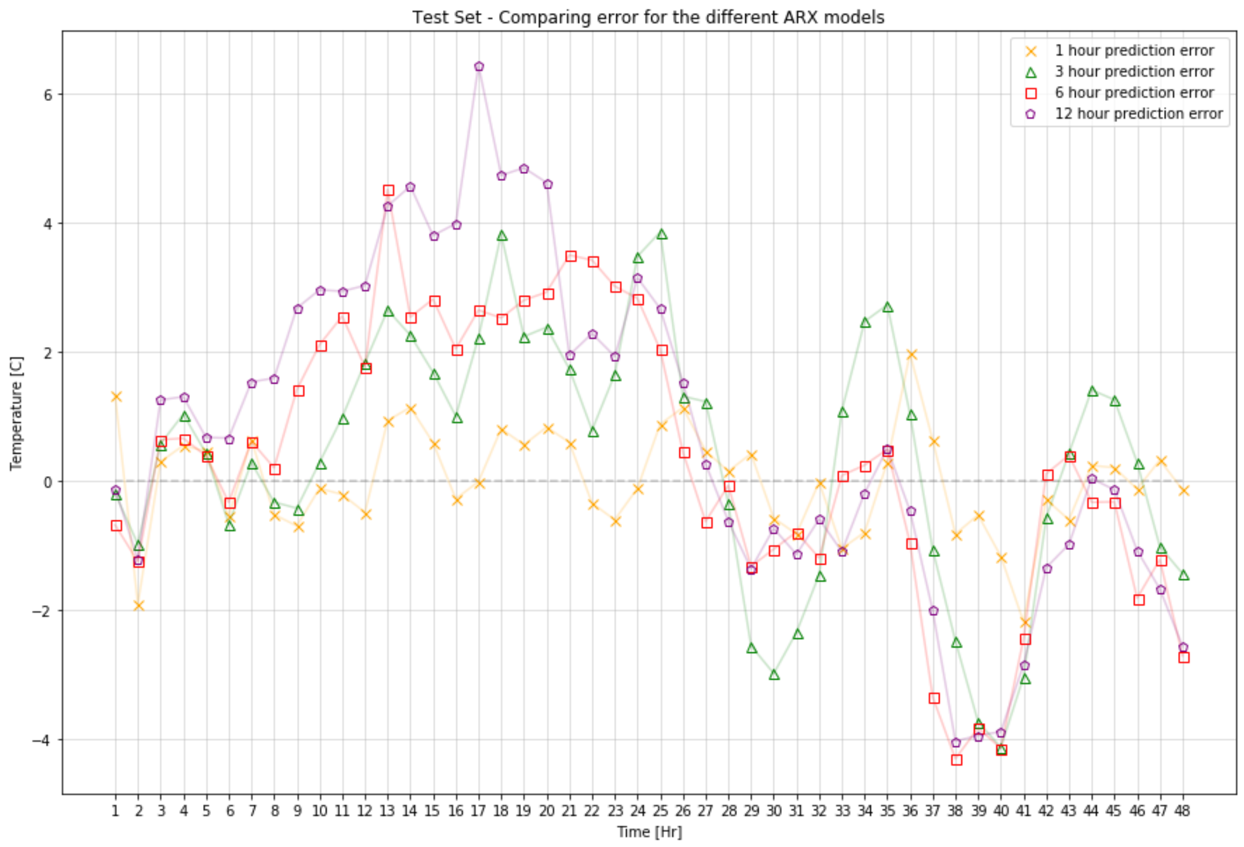


Figure 4-4: This figure shows how many degrees Celsius each point for each ARX model was off for the 48 hours predicted. It is calculated as the difference between measured value and predicted value, this means that the error that is negative is overshooting and error that is positive is undershooting.

5 Conclusion

In this work, python interface was developed to make it easier to obtain weather data from free sources, the python interface works well, but is more user-friendly when used with Met compared with Netatmo. Netatmo delivers their weather data in a messy structure which makes it difficult to extract useful information, sorting of the weather variables has been done as clean as possible with the information needed (station id, variable name, variable value). Met however delivers their data in a structure that is easy to dissect and extract information from, as a result using the met supplier the weather data is delivered in a sleek Pandas data frame that is easy to save and navigate. The artificial neural networks was used to predict the temperature. Four separate models where trained to predict the temperature 1, 3, 6 and 12 hours ahead. In the first experiment, only temperature was used as input to the networks. This constitutes an auto-regressive neural network(ARNN). In the second experiment, precipitation data was introduced into the network, forming an autoregressive neural network with exogenous inputs (ARX-NN). After extensive tuning of hyper parameters for all eight models, the prediction results of the models were compared. The results show that the inclusion of precipitation had a negligible effect on accuracy for temperature prediction. Out of the four model types, 1-hour prediction has the best prediction results for both the AR-NN and the ARX-NN.

5.1 Future Work

The python interface has great potential, trying to extend the interface to include more suppliers and perfecting the data formatting for Netatmo may increase data gathering for use in weather prediction. Since introducing precipitation as an input in the ARX model was shown to slightly improve the performance for temperature prediction, it may be interesting to extend the model with other inputs, or to switch what is being predicted from temperature to precipitation. Mainly, it is interesting to study if introduction of data from other geographical location can improve the prediction results.

Refrences

[1] J. Thibault and I. Senocak, “CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows,” *Inanc Senocak*, Jan. 2009.

[2] “Meteorologisk institutt.” [Online]. Available: https://www.met.no/. [Accessed: 03-May-2018].

[3] “Netatmo Official Site: Welcome to your Smart Home.” [Online]. Available: https://www.netatmo.com//. [Accessed: 03-May-2018].

[4] C. Severance, “Roy T. Fielding: Understanding the REST Style,” *Computer*, vol. 48, no. 6, pp. 7–9, Jun. 2015.

[5] “scikit-learn: machine learning in Python — scikit-learn 0.19.1 documentation.” [Online]. Available: http://scikit-learn.org/stable/. [Accessed: 03-May-2018].

[6] “TensorFlow,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/. [Accessed: 03-May-2018].

[7] “Welcome — Theano 1.0.0 documentation.” [Online]. Available: http://www.deeplearning.net/software/theano/index.html. [Accessed: 03-May-2018].

[8] “Keras Documentation.” [Online]. Available: https://keras.io/. [Accessed: 03-May-2018].

[9] M. Hayati and Z. Mohebi, “Application of Artificial Neural Networks for Temperature Forecasting,” *World Acad. Sci. Eng. Technol.*, vol. 28, p. 5, 2007.

[10] B. A. Smith, R. W. McClendon, and G. Hoogenboom, “Improving Air Temperature Prediction with Artificial Neural Networks,” vol. 3, no. 3, p. 8.

[11] “Python Data Analysis Library — pandas: Python Data Analysis Library.” [Online]. Available: https://pandas.pydata.org/. [Accessed: 08-May-2018].

[12] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[14] “Frost.” [Online]. Available: https://frost.met.no/index.html. [Accessed: 03-May-2018].

[15] X. Glorot, A. Bordes, and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” p. 9.

[16] “The Number of Hidden Layers,” *Heaton Research*. [Online]. Available: http://www.heatonresearch.com/2017/06/01/hidden-layers.html. [Accessed: 03-May-2018].

Appendices

University College of Southeast Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Machine learning with application to weather forecast, etc.

HSN supervisor: Bernt Lie, prof., University College of Southeast Norway (USN)
Liubomyr Vytvytskyi, Ph.D. student, USN
Ole Magnus Brastein, Ph.D. student, USN

External partner: Skagerak Kraft (dr. Beathe Furenes)

Task background:
Machine Learning (ML) and use of IoT (Internet of Things) and Big Data is receiving increased interest from the industry. To experiment with Machine Learning, it is necessary to have access to "Big Data"; access to such data is not always easy to get from the industry. To this end, it is of interest to develop access to "Big Data", and build up competence in Machine Learning – with application to an interesting topic.

Prediction of precipitation (rain) is important in agriculture, but today it is also important for hydropower operation and flood management. Mechanistic meteorology prediction based on 3D CFD/Navier Stokes equations is extremely demanding wrt computing power: generating a 14 day weather forecast can easily take 12 hours on super computers.

Today, plentiful of weather stations are connected to the internet, and are thus available as cheap, distributed sensors (e.g., https://weathermap.netatmo.com/; other sources may be available). For Netatmo weather stations, APIs are available (C, PHP, etc.) for reading public data such as temperature, pressure, humidity, etc. Thus, one could envision developing functions/methods for reading all available data within a certain rectangle of geographic coordinates (or: reading data within the radius of a given coordinate). Because the number of data within each rectangle may vary with time (new installations, dead batteries), a possible solution to handle this is to use the mean value within a rectangle as a scalar measurement, possibly with some statistics of the variation. To this end, one could log weather data for a large number of locations surrounding the location of interest. Example: if we want to predict the weather in Porsgrunn, we could log data in Porsgrunn, Rauland, Oslo, Kragerø, Drangedal, Kristiansand, Evje, Valle, Aalborg, Stavanger, Bergen, Iceland, the Faroe Island, Scotland, England, etc. – location around the current spot of interest, in the direction of weather movement.

A set-up as indicated above will constitute a collection of IoT and will give access to a large number of sensors/data. It is necessary to store the data in some sort of database; storage of data in a text file in the form of CSV data may be adequate initially. Then based on these data, it is possible to train a neural network or similar using ML.

A recent dynamic language, Julia, is well suited for interfacing with C code, and also has good access to machine learning algorithms. Other possibilities are MATLAB or Python. Thus, it is of interest to develop and document a package for interacting with Netatmo weather stations. Next, these data should form the foundation for training and validating an empirical model for weather forecast, and the model should be compared to what can be achieved using a traditional mechanistic model such as data purchased by Skagerak Kraft. The comparison should be done based on accuracy, required computation time, etc.

Task description:
Based on the task background, the following tasks are relevant:

1. Give an overview of API for accessing Netatmo weather stations in existing languages (C, PHP, etc.), and develop an API for accessing these data from Julia (or MATLAB or Python).
2. Based on studies of wind/weather directions, test out possible locations for weather data around a chosen location of interest for Skagerak Kraft, e.g., the Kragerø water ways catchment. Check to see if there is some correlation between the local data (e.g., Kragerø water ways) and the data surrounding this location – the correlation should occur on quarterly hour basis, hourly basis, 3 hour basis, 6 hour basis, 12 hour basis, 24 hour basis, etc. – as long back as possible – to enable as long prediction horizon as possible.
3. After an initial investigation, choose a reduced set of data points, and set up logging of the data with storage in a file.
4. Based on available data, set up a system for machine learning. If y(t) is measurements at time t, then use machine learning to find a mapping F for y(t) = F(y(t-1), y(t-2), …, y(t-T)) for various values of total horizon T and various time steps dT. Validate the model, and compare your results to those from a meteorological model. The model should preferably give a distribution of predictions.
5. Report the work in the Master's Thesis, and possibly in a suitable conference/journal paper.

Student category: IIA, EPE, PT, EET students with a reasonable background and understanding of programming languages and C/C++/C#.

Practical arrangements:
The workplace is Campus Kjølnes of University College of Southeast Norway. There will be weekly meetings with the supervisor from January until mid April, either face-to-face or via Skype, with hand-in of partial reports every 3 weeks.

The thesis work will start January 2, 2018, and the deadline for thesis hand-in is May 15 2018 at 14:00. An oral presentation with examination and grading will take place no later than June 22, 2018.

Signatures:

Student (date and signature):

Supervisor (date and signature):

PYTHON INTERFACE DOCUMENTATION

Rev 01

Erik Boye Abrahamsen

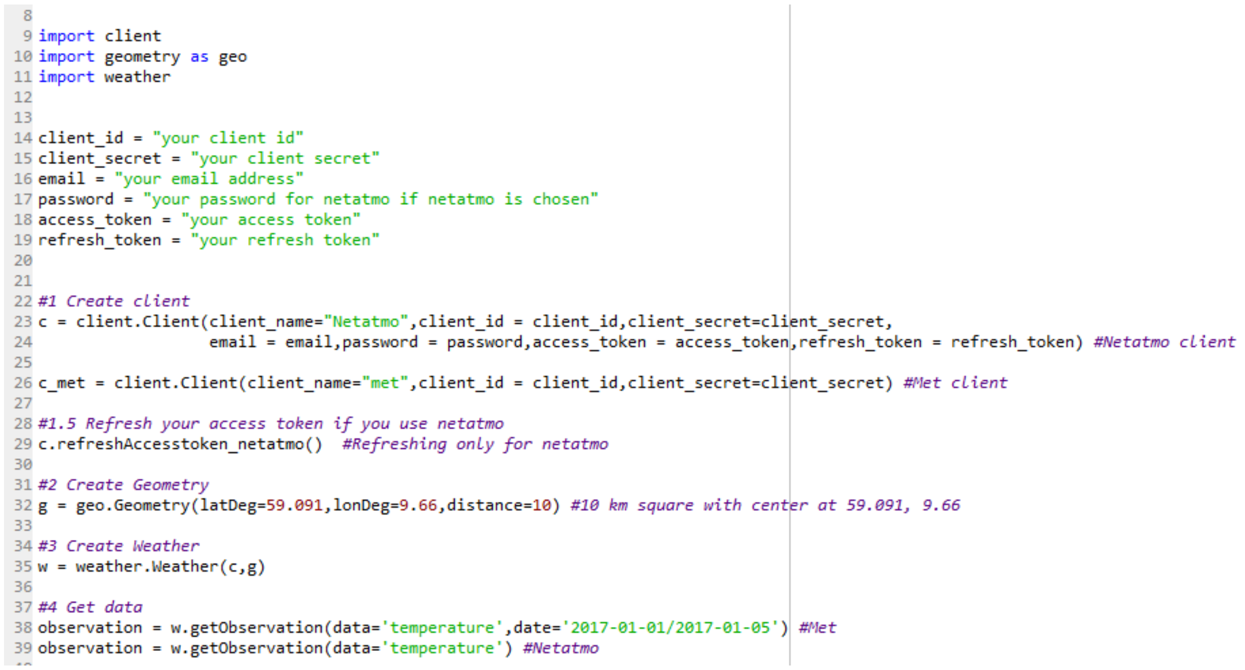
Contents

Quick Start

1. Create a client object. Note this depends on what type of client you want Met or Netatmo.
2. Create a geometry object. This object stores the coordinates for your square
3. Create a weather object and pass inn your client and geometry objects.
4. Use your weather object for getting data by accessing the method getObservation() in the weather class
5. Saving the observation from met can be done by calling observation.to_csv(filename) more information at https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html

NOTE: Currently pandas csv saving is only supported for met. This is because of the huge difference in Netatmo and Met data retrieving. Netatmo needs additional manipulation to be handled in such a way because the data is stored in several layers of nested dicts.

Please see Figure 1 for an illustration of using Netatmo. In the case of using Met, you would not need : email, password, access_token, refresh_token or need to refresh (line 29)



```
 8
 9 import client
10 import geometry as geo
11 import weather
12
13
14 client_id = "your client id"
15 client_secret = "your client secret"
16 email = "your email address"
17 password = "your password for netatmo if netatmo is chosen"
18 access_token = "your access token"
19 refresh_token = "your refresh token"
20
21
22 #1 Create client
23 c = client.Client(client_name="Netatmo",client_id = client_id,client_secret=client_secret,
24                   email = email,password = password,access_token = access_token,refresh_token = refresh_token) #Netatmo client
25
26 c_met = client.Client(client_name="met",client_id = client_id,client_secret=client_secret) #Met client
27
28 #1.5 Refresh your access token if you use netatmo
29 c.refreshAccesstoken_netatmo()  #Refreshing only for netatmo
30
31 #2 Create Geometry
32 g = geo.Geometry(latDeg=59.091,lonDeg=9.66,distance=10) #10 km square with center at 59.091, 9.66
33
34 #3 Create Weather
35 w = weather.Weather(c,g)
36
37 #4 Get data
38 observation = w.getObservation(data='temperature',date='2017-01-01/2017-01-05') #Met
39 observation = w.getObservation(data='temperature') #Netatmo
```

FIGURE 1: QUICK START, SHOWS HOW TO USE EITEHR MET OR NETATMO

Getting access to Netatmo

All information about Netatmo api is located at: https://dev.netatmo.com/resources/technical

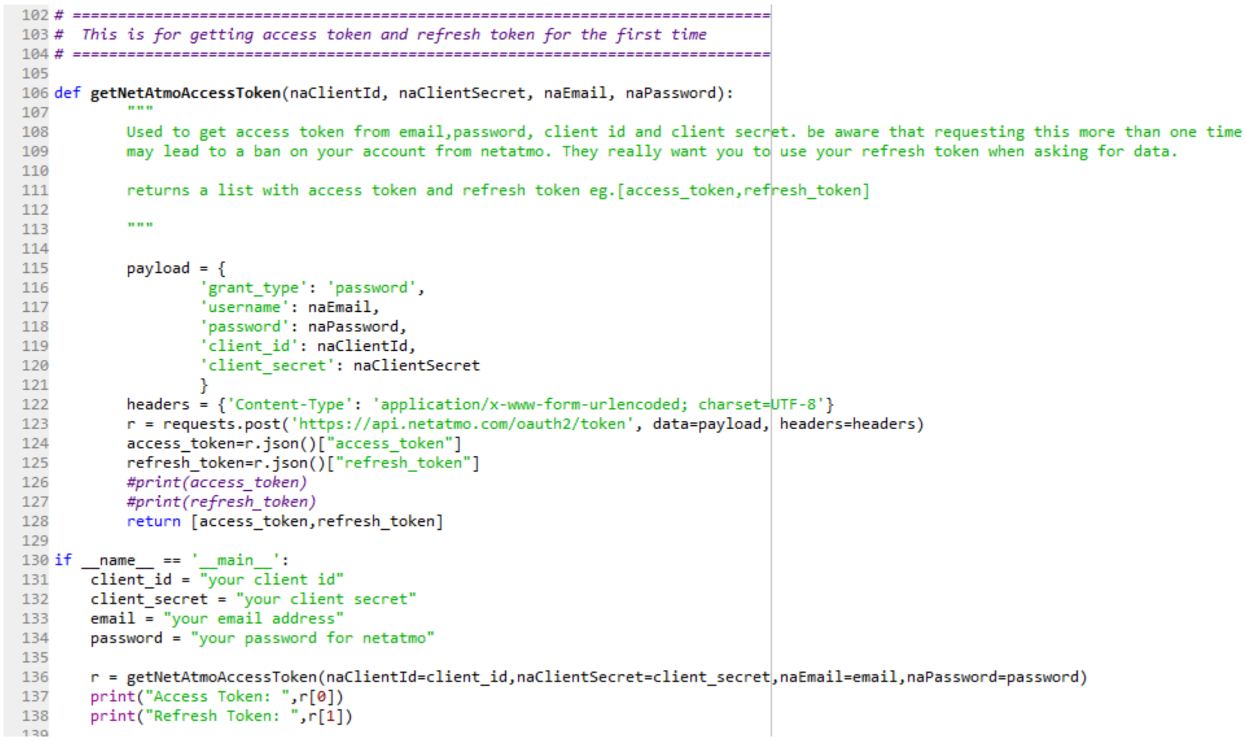
To get access to Netatmo, you need to create a user and an app at their website. This can be done at https://dev.netatmo.com/myaccount/createanapp

When you have obtained your user account, client id and client secret you can obtain an access token and refresh token. (**NOTE:** If you request access token several times instead of using refresh token you might be banned from Netatmo.)

GETTING ACCESS TOKEN AND REFRESH TOKEN

1. Open the python file netatmo.py in your python IDE

2. In the bottom of the file there is already prepared a snippet for asking for access token and refresh token
3. Just fill in **YOUR** email, password, client id and client secret (See Figure 2 for how it should look)
4. After filling in your info run the script. This will produce two print statements in the console.
5. Copy the tokens and save them somewhere, or add them directly to your clientFile.py (This is a helper/skeleton file that can be used to obtain data)



```python
# ============================================================================
#  This is for getting access token and refresh token for the first time
# ============================================================================

def getNetAtmoAccessToken(naClientId, naClientSecret, naEmail, naPassword):
        """
        Used to get access token from email,password, client id and client secret. be aware that requesting this more than one time
        may lead to a ban on your account from netatmo. They really want you to use your refresh token when asking for data.

        returns a list with access token and refresh token eg.[access_token,refresh_token]

        """

        payload = {
                'grant_type': 'password',
                'username': naEmail,
                'password': naPassword,
                'client_id': naClientId,
                'client_secret': naClientSecret
                }
        headers = {'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'}
        r = requests.post('https://api.netatmo.com/oauth2/token', data=payload, headers=headers)
        access_token=r.json()["access_token"]
        refresh_token=r.json()["refresh_token"]
        #print(access_token)
        #print(refresh_token)
        return [access_token,refresh_token]

if __name__ == '__main__':
    client_id = "your client id"
    client_secret = "your client secret"
    email = "your email address"
    password = "your password for netatmo"

    r = getNetAtmoAccessToken(naClientId=client_id,naClientSecret=client_secret,naEmail=email,naPassword=password)
    print("Access Token: ",r[0])
    print("Refresh Token: ",r[1])
```

FIGURE 2: SHOWS CODE SNIPPET TO OBTAIN ACCESS AND REFRESH TOKEN FOR NETATMO

GETTING ACCESS TO MET

All information about Met api is located at: https://frost.met.no/index.html

Met needs your email to give you a client id and client secret. You can register your email at https://frost.met.no/auth/requestCredentials.html

When you have obtained your client id and client secret please do as illustrated in Quick Start.

Class documentation

This section covers the class documentation. The available methods, arguments and returns. In the Figure 3, below a diagram of the class structure is shown.

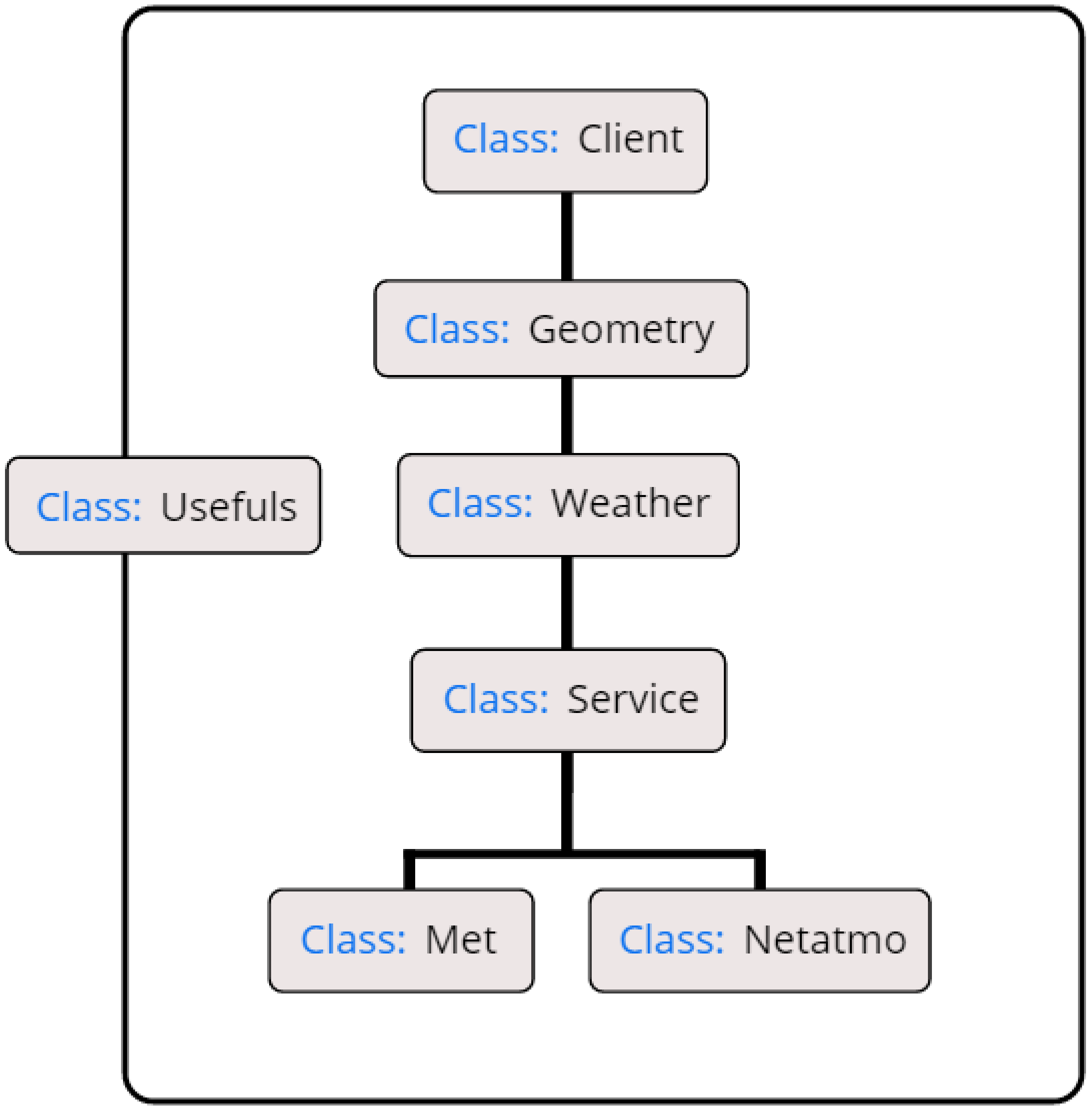


FIGURE 3: CLASS STRUCTURE FOR INTERFACE

CLIENT

File: client.py Dependencies: requests

The client class is the class that handles the client.

CONSTRUCTOR:

Arguments	Description
client_name	This is the name of your selected service, either Met or Netatmo)
client_id	This is your client id from the selected service
client_secret	This is your client secret from the selected service
email	Your email if needed in your selected service
password	Your password if needed in your selected
access_token	This is the access token you have acquired from your selected service
refresh_token	This is the refresh token you have acquired from your selected service

METHODS:

def refreshAccesstoken_netatmo():

It refreshes your access token, it will print to console the new token, please change it with your old one. One token lasts approximately 15 minutes.

Returns: Nothing

Geometry

File: geometry.py   Dependencies: math

Class representing a coordinate on a sphere, most likely Earth.This class is based from the code smaple in this paper. http://janmatuschek.de/LatitudeLongitudeBoundingCoordinatesThe owner of that website, Jan Philip Matuschek, is the full owner of his intellectual property. This class is simply a Python port of his veryuseful Java code. All code written by Jan Philip Matuschek and ported by me (which is all this class) is owned by Jan Philip Matuschek.
IMPORTED USEFUL CODEBLOCKS FROM GITHUB: https://github.com/jfein/PyGeoTools
http://www.hamstermap.com/quickmap.php can be used to visualize points on a map

Constructor:

Arguments	Description
latDeg	The latitude center point of your square
lonDeg	The longitude center point of your square
distance	This is the size of the square in kilometers

Methods:

def getLocation():
Used to get the lat and lon values used to produce the square
Returns: list with center position of square [lat,lon]

def getSquare():
Use to get the square list to be used in services like Met or Netatmo
Returns: coordinates for the squares coordinates

def getLatList():
Returns list of all the lat points, in order nw,sw,se,ne
Returns: list of all the lat points, in order nw,sw,se,ne

def getLonList():
Returns list of all the lon points, in order nw,sw,se,ne
Returns: Returns list of all the lon points, in order nw,sw,se,ne

Weather

File: weather.py   Dependencies: service.py

The weather class is the class that asks the service class for data and is the one to be used for obtaining data

Constructor:

Arguments	Description
client	This is the client object made for your service
geometry	This is the geometry object made

Methods:

def getObservation(data, date = None):
Used to get the wanted observations within the geometry
data is the weather variable wanted
date is a string on the format 2016-01-01/2018-01-01
Returns: The requested data.

SERVICE

File: service.py	Dependencies: met.py, netatmo.py

The service class holds the service providers and asks them for their information. Currently SERVS = ['met','netatmo'], is supported if future suppliers are to be added they must be specified here.

CONSTRUCTOR:

Arguments	Description
client	This is the client object created at the start

METHODS:

def getData(,data,date,geometry):
Used for asking the service providers for the data, either data at point or data inside square geometry
Returns: data from the different services, based on if its point or geometry.

MET

File: met.py Dependencies: usefuls.py, pandas

The met class holds everything that should do with met. It transforms to correct formats and asks the right URLs.
Currently supported weather elements are:
('temperature':'air_temperature',
'hourlytemperature',’dailytemperature',’temperatur','timestemperatur','dagligtemperatur','rain','dailyrain','regn','dagligregn','wind','winddirection','vind','vindretning','humidity','fuktighet','pressure'
,'airpressure','trykk','lufttrykk')

CONSTRUCTOR:

Arguments	Description
client	Client object

METHODS:

def getPointData(data,date,geometry):
Gets the station nearest to the point specified.
Data is the specified data
Date is the specified date
And geometry is the geometry object
Returns: Requested data

def getGeometryData(data,date,geometry):
Returns specified data from all the stations within the given geometry object to the given date.
Note: if error occur like 404. this may mean that your rectangle is too small or that the station found don't have that type of data you're looking for
Returns: Requested data

NETATMO

File: netatmo.py Dependencies: time, pandas, requests

The Netatmo class holds everything that should do with Netatmo. It transforms to correct formats and asks the right URLs.
Currently supported weather elements are:
('temperature','tempratur','temp','rain','regn','wind','vind','humidity','fuktighet')

CONSTRUCTOR:

Arguments	Description
client	This is the client object

METHODS:

def getGeometryData(data,geometry):
Returns specified data from all the stations within the given geometry object at the current time.
Note: if error occur like 404. this may mean that your rectangle is too small or that the station found don't have that type of data you're looking for
Returns: Requested data

USEFULS

File: usefuls.py	Dependencies: datetime, requests

The Usefuls class contains some useful methods when dealing with dates and lists

CONSTRUCTOR:

Arguments	Description

METHODS:

def getDatesList(date):
Splits a date string on the format year-month-day/year2-month2-day2, into separate dates and returns a list where each member is a list with corresponding year, month, day at the indexes
ex: date = "2017-01-01/2017-01-04" returns a list dates = [[2017, 01, 01], [2017 , 01 , 02] , [2017 , 01 , 03]]
Returns: a list where each member is a list with corresponding year, month, day at the indexes

def datesListToStringList(datesList):
Takes in a list of dates then combines them
Returns a list where each member is the combined date eg, datelist = [[2017, 01, 01], [2017, 01 , 02]]
the return of this datelist is stringList = [['2017-01-01'], ['2017-01-02']]
Returns: a list where each member is the combined date

def backToDateString(datesList):
Takes in a date list and makes it into a string again
Returns a string that goes from a dateslist = [[2017, 01, 01] , [2017 , 01 , 02] , [2017 , 01 , 03]] to a string combined = "2017-01-01/2017-01-04"
Returns: a string that goes from a dateslist to a stringdate

def requestData(url,client):
takes in a url and a client object to make a request to specified url
Used for requesting data from url
Returns: response object from request

def chunkIt(seq, num):
Used to divide a list into approx equal parts
Takes in a list(seq) and separates it into (num) equal parts
Returns: a divided list

```python
import requests

class Client(object):

    def __init__(self,client_name,client_id,client_secret,email = None,
                 password = None,access_token = None,refresh_token = None):
        self.client_name = client_name.lower()
        self.client_id = client_id
        self.client_secret = client_secret
        self.email = email
        self.password = password
        self.access_token = access_token
        self.refresh_token = refresh_token

    def _setNewAccessToken(self,newAccess_token):
        self.access_token = newAccess_token

    def refreshAccesstoken_netatmo(self):
        """
        Used for refreshing access token. This is the way Netatmo wants
        their API to be used. requesting new tokens may result in a ban.
        """
        payload = {
                'grant_type': 'refresh_token',
                'refresh_token': self.refresh_token,
                'client_id': self.client_id,
                'client_secret': self.client_secret
                }
        headers = {'Content-Type':
            'application/x-www-form-urlencoded; charset=UTF-8'}
        r = requests.post('https://api.netatmo.com/oauth2/token',
                          data=payload, headers=headers)
        newTok = r.json()["access_token"]
        self._setNewAccessToken(newTok)
        print("AccesToken Refreshed --- New is: ",newTok)
```

```python
import math

class Geometry(object):

    '''
    Class representing a coordinate on a sphere, most likely Earth.

    This class is based from the code smaple in this paper:
        http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates

    The owner of that website, Jan Philip Matuschek, is the full owner of
    his intellectual property. This class is simply a Python port of his very
    useful Java code. All code written by Jan Philip Matuschek and ported by me
    (which is all of this class) is owned by Jan Philip Matuschek.

    IMPORTED USEFUL CODEBLOCKS FROM GITHUB: https://github.com/jfein/PyGeoTools

    http://www.hamstermap.com/quickmap.php can be used to visualize points on a map
    '''

    MIN_LAT = math.radians(-90);
    MAX_LAT = math.radians(90);
    MIN_LON = math.radians(-180);
    MAX_LON = math.radians(180);

    EARTH_RADIUS = 6378.1  # kilometers

# ==============================================================================
#       Constructor
# ==============================================================================
    def __init__(self,latDeg,lonDeg,distance):

        self.latDeg = float(latDeg)
        self.lonDeg = float(lonDeg)
        self.latRad = float(math.radians(latDeg))
        self.lonRad = float(math.radians(lonDeg))
        self.distance = distance
        self._check_bounds()

        if(self.distance > 0):
            self.square = self._setupBox(self.distance)
            self.point = False
        else:
            self.square = [self.latDeg,self.lonDeg]
            self.point = True

# ==============================================================================
#       Private methods
# ==============================================================================

    def _check_bounds(self):
        """
        This method is used to check if the lat and lon is withing the macimum
        and minimums of the earth
        """
        if (self.latRad < Geometry.MIN_LAT
                or self.latRad > Geometry.MAX_LAT
                or self.lonRad < Geometry.MIN_LON
                or self.lonRad > Geometry.MAX_LON):
            raise Exception("Illegal arguments")
    def _setupBox(self,distance,radius=EARTH_RADIUS):

        """
        Used to set up a box that is X distance in km from the center point
        given(latDeg,lonDeg) it will return the lat, lon for all
        four corners of the box.

        Returns a list with cordinates in degrees for each corener in
        bounding box [NW,SW,SE,NE] where each item is a
        list with two points: [lat,lon]

        """

        if radius < 0 or distance < 0:
            raise Exception("Illegal arguments")

        # angular distance in radians on a great circle
        rad_dist = distance / radius

        min_lat = self.latRad - rad_dist
        max_lat = self.latRad + rad_dist

        if min_lat > Geometry.MIN_LAT and max_lat < Geometry.MAX_LAT:
            delta_lon = math.asin(math.sin(rad_dist) / math.cos(self.latRad))

            min_lon = self.lonRad - delta_lon
            if min_lon < Geometry.MIN_LON:
                min_lon += 2 * math.pi

            max_lon = self.lonRad + delta_lon
            if max_lon > Geometry.MAX_LON:
                max_lon -= 2 * math.pi
        # a pole is within the distance
        else:
            min_lat = max(min_lat, Geometry.MIN_LAT)
            max_lat = min(max_lat, Geometry.MAX_LAT)
            min_lon = Geometry.MIN_LON
            max_lon = Geometry.MAX_LON

        SW = [math.degrees(min_lat),math.degrees(min_lon)]
        NE = [math.degrees(max_lat),math.degrees(max_lon)]
        NW = [SW[0],NE[1]]
        SE = [NE[0],SW[1]]
        return [NW,SW,SE,NE]

# ==============================================================================
#       Public Methods
# ==============================================================================

    def getLocation(self):
        """
        Used to get the lat and lon values used to produce the square
        """
        return [self.latDeg,self.lonDeg]

    def getSquare(self):
        """
        Use to get the square list to be used in services like Met or Netatmo
        """
```

```
            return self.square

        def getLatList(self):
             """
             Returns list of all the lat points, in order nw,sw,se,ne
             """
             return [self.square[0][0],self.square[1][0],self.square[2][0],self.square[3][0]]

        def getLonList(self):
            """
            Returns list of all the lon points, in order nw,sw,se,ne
            """
            return [self.square[0][1],self.square[1][1],self.square[2][1],self.square[3][1]]


    if __name__ == '__main__':

        g = Geometry(latDeg = 26.062951, lonDeg = -80.238853,distance = 5)
        g2 = Geometry(latDeg = 59.152676, lonDeg = 9.652863,distance = 10)


        sq =  g2.getSquare()
        lat1 = g.getLatList()
        lat2 = g2.getLatList()
        lon1 = g.getLonList()
        lon2 = g.getLonList()
```

```python
import service

class Weather(object):

# ==============================================================================
# Constructor
# ==============================================================================

    def __init__(self,client,geometry):

        self.client = client
        self.geometry = geometry
        self.service= self._authenticate(self.client)


# ==============================================================================
# Private methods
# ==============================================================================

    def _authenticate(self,client):
        """
        Used for checking that the service exists and creates a service object
        """
        s =service.Service(client)
        return s


# ==============================================================================
#       Public methods
# ==============================================================================

    def getObservation(self, data, date = None):
        """
        Used to get the wanted observations within the geometry
        data is the weather variable wanted
        date is a string on the format 2016-01-01/2018-01-01
        """
        data = data.lower()

        if(self.geometry == None):
                raise Exception("GEOMETRY NOT INSTANTIATED,"
                                +"please use setGeometry(latDeg, lonDeg, distance)")

        return self.service.getData(data,date,self.geometry)
```

```python
import met,netatmo

class Service(object):

    SERVS = ['met','netatmo']

    def __init__(self,client):
        self.source = self._pickSource(client)

    def _pickSource(self,client):
        """
        Used for picking the right source of the data, this is handeled
        automatically aslong as the source is listed
        """
        serviceName = client.client_name
        if(serviceName not in self.SERVS):
            raise Exception("ERROR! "+serviceName+" is not a supported
                                service. Supported services are: ",self.SERVS)
        else:
            if(serviceName == 'met'):
                return met.Met(client)
            elif(serviceName == 'netatmo'):
                return netatmo.Netatmo(client)

    def getData(self,data,date,geometry):
        """
        Used for getting the data
        """
        #self._datesList(date)

        if (geometry.point):
            return self.source.getPointData(data,date,geometry)
        else:
            return self.source.getGeometryData(data,date,geometry)
```

```python
import usefuls
import pandas as pd


class Met(object):

# ==============================================================================
# Constructor
# ==============================================================================
    def __init__(self,client):
        self.geometryPoint = False
        self.client = client
        self.elements = {'temperature':'air_temperature',
                         'hourlytemperature':'air_temperature',
                         'dailytemperature':'air_temperature',
                         'temperatur':'air_temperature',
                         'timestemperatur':'air_temperature',
                         'dagligtemperatur':'air_temperature',
                         'rain':'sum(precipitation_amount PT1H)',
                         'dailyrain':'sum(precipitation_amount PT1H)',
                         'regn':'sum(precipitation_amount PT1H)',
                         'dagligregn':'sum(precipitation_amount PT1H)',
                         'wind':'mean(wind_speed PT1H)',
                         'winddirection':'wind_from_direction',
                         'vind':'mean(wind_speed PT1H)',
                         'vindretning':'wind_from_direction',
                         'humidity':'mean(relative_humidity PT1H)',
                         'fuktighet':'mean(relative_humidity PT1H)',
                         'pressure':'air_pressure_at_sea_level',
                         'airpressure':'air_pressure_at_sea_level',
                         'trykk':'air_pressure_at_sea_level',
                         'lufttrykk':'air_pressure_at_sea_level',
                         '':''}

# ==============================================================================
# Private Methods
# ==============================================================================
    def _getNearestStation(self,location):
        """
        Gets the station nearest the specified point
        """
        url = ('https://frost.met.no/sources/v0.jsonld?types='
        +'SensorSystem&geometry=nearest(POINT({0} {1}))'.format(location[1],
                                   location[0]))
        stationID = usefuls.Usefuls().requestData(url,self.client)
        stationID = stationID['data'][0]['id']
        return stationID




    def _getStationIDsInGeometry(self,geometry):
        """
        The geometry is a polygon so the first and last point must be the same
        to close the geometry
        """
        NW = geometry[0]
        SW = geometry[1]
        SE = geometry[2]
        NE = geometry[3]
        url = ('https://frost.met.no/sources'
        +'/v0.jsonld?types=SensorSystem&geometry='
        +'POLYGON (({0} {1}, {2} {3}, {4} {5}, {6} {7}, {8}'
        +' {9}))'.format(NW[1],NW[0],SW[1],SW[0],SE[1],SE[0],NE[1],NE[0],NW[1],NW[0]))
        stations = usefuls.Usefuls().requestData(url,self.client)
        le = len(stations['data'])
        stationIDs = [0]*le
        for i in range(0,le):
            stationIDs[i] = stations['data'][i]['id']
        return stationIDs



    def _getCorrectURL(self,data,datesList,stationIDs):
        date = usefuls.Usefuls().backToDateString(datesList)
        s = ''

        if(self.geometryPoint):
            if(data == ''):
                url = ('https://frost.met.no/observations'
                       +'/availableTimeSeries/v0.jsonld?referencetime='
                       +date+'&sources='+str(stationIDs))
            else:
                url = ('https://frost.met.no/observations'
                +'/v0.jsonld?referencetime='+date+'&elements='
                +self.elements[data]+'&sources='+str(stationIDs))
        else:
            for i in range(len(stationIDs)):
                s += str(stationIDs[i])+','
            s = s[:-1]
            if(data == ''):
                url = ('https://frost.met.no/observations/'+
                       'availableTimeSeries/v0.jsonld?referencetime='
                       +date+'&sources='+s)
            else:
                url = ('https://frost.met.no/observations'+
                '/v0.jsonld?referencetime='+date+'&elements='
                +self.elements[data]+'&sources='+s)
        return url



    def _preProcessDATA(self,stationIDs,datesList):
        """
        Preprocesses the DATA such that all memory for all excpected
        values are created. will be used as a checking guide to find
        the correct data to place or to place none
        """
        #Makes the first item in each station in the DATA the station id
        DATA = []
        station = []
        for st in stationIDs:
            station.append(st)
            DATA.append(station)
            station = []
        #Makes a list of expected times to be in the json object
        times = []
        dataa = []
        for da in datesList:
            for t in range(24):
                if t >= 10:
                    times.append([da+'T{0}:00:00.000Z'.format(t),None])
                else:
                    times.append([da+'T{0}{1}:00:00.000Z'.format(0,t),None])

            dataa.append(times)
            times = []

        #adds expected times to the DATA list
        for s in DATA:
            s.append(dataa)
        return DATA

    def _postProcessDATA(self,realDATA,expectedDATA):
        stations = []
        for f in range(len(realDATA)):
            for i in range(len(expectedDATA)):

                if expectedDATA[i][0] in realDATA[f][0]:
                    stations.append([expectedDATA[i][0],
                                     [realDATA[f][1],realDATA[f][2]]])
                else:
                    stations.append([expectedDATA[i][0],[None,None]])
        return stations

    def _getSpecifiedData(self,data,datesList,url,stationIDs):
        """
        Picks the data from json, if daily is requested an average will be
        returned, else its hourly for the days specified
        """
        datesList = usefuls.Usefuls().datesListToStringList(datesList)
        DATA = self._preProcessDATA(stationIDs,datesList)

        dta = []

        data = usefuls.Usefuls().requestData(url,self.client)
        data = data['data']

        #adds the real data to a list [[station id, timestamp, value],[..]..]
        for d in range(len(data)):
            dta.append([data[d]['sourceId'],
                        data[d]['referenceTime'],
                        data[d]['observations'][0]['value']])

        #postData = self._postProcessDATA(dta,DATA)

        y = []
        for it in DATA:

            if(it[0] in dta[0]):
                y.append(1)
            else:
                y.append(0)



        #specifiedData = [datesList,DATA,dta]
        specifiedData = dta
        return self._sortToPandas(specifiedData)
```

```python
    def _sortToPandas(self,data):
        df = pd.DataFrame()
        df = df.append(data)
        df.rename(columns={0: 'Station Id',
                           1: 'Timestamp',2:'Value'}, inplace=True)
        return df

# =============================================================================
# Public Methods
# =============================================================================

    def getPointData(self,data,date,geometry):
        """
        Gets the station nearest to the point specified.
        """
        datesList = usefuls.Usefuls().getDatesList(date)
        self.geometryPoint = True
        square = geometry.getSquare()
        nearestStation = self._getNearestStation(square)
        url = self._getCorrectURL(data,datesList,nearestStation)
        data = self._getSpecifiedData(data,datesList,url,nearestStation)
        return data


    def getGeometryData(self,data,date,geometry):
        """
        Returns data from all the stations withing the given geometry.
        Note: if error occour like 404. this may mean that your rectangle is
        too small or that the station found don't have that
        type of data you're looking for
        """
        datesList = usefuls.Usefuls().getDatesList(date)
        self.geometryPoint = False
        square = geometry.getSquare()
        stationIDs = self._getStationIDsInGeometry(square)
        url = self._getCorrectURL(data,datesList,stationIDs)
        data = self._getSpecifiedData(data,datesList,url,stationIDs)
        return data
```

```python
import time
import pandas as pd
import requests

class Netatmo(object):

    NETATMO_OAUTH_URL = 'https://api.netatmo.com/oauth2/token';
    NETATMO_STATION_DATA = 'https://api.netatmo.com/api/getstationsdata';
    NETATMO_PUBLIC_DATA = "https://api.netatmo.com/api/getpublicdata";

# ==============================================================================
# Constructor
# ==============================================================================
    def __init__(self,client):
        self.client = client
        if(self.client.password == None or self.client.email == None or
           self.client.access_token == None or self.client.refresh_token == None):
            raise Exception("Error! Netatmo needs email, password,"+
                            "access token and refresh token aswell."
                            +" If you haven't got a access and refresh token. "
                            +"got into netatmo.py and "
                            +"run the code in __name__ == '__main__' part to obtain it.")

        self.elements = {'temperature':'temperature',
                         'tempratur':'temperature',
                         'temp':'temperature',
                         'rain':'rain',
                         'regn':'rain',
                         'wind':'wind',
                         'vind':'wind',
                         'humidity':'humidity',
                         'fuktighet':'humidity',
                         '':''}


# ==============================================================================
# Public Methods
# ==============================================================================

    def _sortData(self,dat):
        """
        Takes the data, sorts it so it got its correct id and timestamp.
        This sorting is not comleted yet it's missing weather variable sorting
        returns a list [timestamp,id,DATA]
        """
        t = time.ctime(dat['time_server'])
        f = pd.DataFrame()
        d = []
        dd = []
        iid = None
        for f in dat['body']:
            iid = f['_id']
            for k in f['measures']:
                d.append(t)
                d.append(iid)
                d.append(f['measures'][k])
                dd.append(d)
                d = []
            iid = None
        return dd
    def _getPublicData(self,access_token,lat_ne,lon_ne,
                       lat_sw,lon_sw,required_data = '',fltr = "false"):
        """
        **access_token** *(required)*\n
        Obtained from https://api.netatmo.com/oauth2/token \n
        **lat_ne** *(required)*\n Latitude of the north east corner of the
        requested area. -85 <= lat_ne <= 85 and lat_ne>lat_sw.
        \n \n example value: 15  \n
        **lon_ne** *(required)*\n Longitude of the north east corner of the
        requested area.-180 <= lon_ne <= 180 and lon_ne>lon_sw
        \n \n example value: 20 \n
        **lat_sw** *(required)*\n Latitude of the south west corner of the
        requested area. -85 <= lat_sw <= 85 \n \n example value: -15  \n
        **lon_sw** *(required)*\n Longitude of the south west corner of the
        requested area. -180 <= lon_sw <= 180 \n \n example value: -20  \n
        **required_data = "rain"** *(optional)* \n To filter stations based
        on relevant measurements you want (e.g. rain will only return stations
        with rain gauges). Default is no filter. You can find all measurements
        available on the Thermostat page.
        \n \n example value: "rain", "humidity"
        **fltr = False** *(optional)*\n True to exclude station with abnormal
        temperature measures. Default is false. PS: this must be a string\n
        \n \n for more:
        https://dev.netatmo.com/resources/technical/reference/weatherapi/getpublicdata
        """
        params = {'access_token':access_token,'lat_ne':lat_ne
                  ,'lon_ne':lon_ne
                  ,'lat_sw':lat_sw
                  ,'lon_sw':lon_sw
                  ,'required_data': required_data
                  ,'filter':fltr}
        req = requests.post(self.NETATMO_PUBLIC_DATA,params)
        req.raise_for_status()
        dat = req.json()
        return dat

    def getPointData(self,data,date,geometry):
        """
        point geometry is not supported by Netatmo
        """
        raise Exception("ERROR: Point geometry is not supported by Netatmo.")


    def getGeometryData(self,data,date,geometry):
        """
        returns a list with the data obtained from the search withing the rectangle
        """
        square = geometry.getSquare()
        dat = self._getPublicData(access_token=self.client.access_token,
                                  lat_ne=square[3][0],lon_ne = square[3][1],
                                  lat_sw = square[1][0],lon_sw = square[1][1],
                                  required_data = self.elements[data])
        return self._sortData(dat)
        #return dat



# ==============================================================================
#  This is for getting access token and refresh token for the first time
```

```python
# ============================================================================

def getNetAtmoAccessToken(naClientId, naClientSecret, naEmail, naPassword):
        """
        Used to get access token from email,password, client id and client
        secret. be aware that requesting this more than one time
        may lead to a ban on your account from netatmo. They really want you
        to use your refresh token when asking for data.

        returns a list with access token and refresh token eg.[access_token,refresh_token]

        """

        payload = {
                'grant_type': 'password',
                'username': naEmail,
                'password': naPassword,
                'client_id': naClientId,
                'client_secret': naClientSecret
                }
        headers = {'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'}
        r = requests.post('https://api.netatmo.com/oauth2/token',
                          data=payload, headers=headers)
        access_token=r.json()["access_token"]
        refresh_token=r.json()["refresh_token"]
        #print(access_token)
        #print(refresh_token)
        return [access_token,refresh_token]

if __name__ == '__main__':
    client_id = "your client id"
    client_secret = "your client secret"
    email = "your email address"
    password = "your password for netatmo"

    r = getNetAtmoAccessToken(naClientId=client_id,
                              naClientSecret=client_secret,
                              naEmail=email,naPassword=password)
    print("Access Token: ",r[0])
    print("Refresh Token: ",r[1])
```

```
import datetime,requests


class Usefuls(object):

    headers = {'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'}

    def getDatesList(self,date):
        """
        splits a date string on the format year-month-day/year2-month2-day2,
        into separeate dates and
        returns a list where each member is a list with corresponding year,
        month,day  at the indexes
        ex:
            date = "2017-01-01/2017-01-04"  returns a list
            dates = [ [ 2017 , 01 , 01 ] , [ 2017 , 01 , 02 ] , [ 2017 , 01 , 03 ] ]
        """
        dates = []

        if('/' in date):
            dd = date.split('/')
            dF = dd[0].split('-')
            dF = datetime.date(int(dF[0]),int(dF[1]),int(dF[2]))
            dT = dd[1].split('-')
            dT = datetime.date(int(dT[0]),int(dT[1]),int(dT[2]))
            dt = dT -dF

            for i in range(0,dt.days):
                v = dF + datetime.timedelta(days=i)
                dates.append([v.year,v.month,v.day])
        else:
            dF = date.split('-')
            dF = datetime.date(int(dF[0]),int(dF[1]),int(dF[2]))
            dates.append([dF.year,dF.month,dF.day])

        return dates


    def datesListToStringList(self,datesList):
        """
        Returns a list where each member is the combined date eg,
        datelist = [ [ 2017 , 01 , 01 ] , [ 2017 , 01 , 02 ]]
        the return of this datelist is
        stringList = [ [ '2017-01-01' ] , [ '2017-01-02' ] ]
        """
        stringList = []
        for i in range(len(datesList)):
            y = str(datesList[i][0])
            m = datesList[i][1]
            d = datesList[i][2]

            if m <10:
                m = '0{0}'.format(datesList[i][1])
            else:
                m = str(datesList[i][1])

            if d <10:
                d = '0{0}'.format(datesList[i][2])
            else:
                d = str(datesList[i][2])
            st = y+'-'+m+'-'+d
            stringList.append(st)
        return stringList


    def backToDateString(self,datesList):
        """
        Returns a string that goes from a
        dateslist = [ [ 2017 , 01 , 01 ] , [ 2017 , 01 , 02 ] , [ 2017 , 01 , 03 ] ]
        to a string combined = "2017-01-01/2017-01-04"
        """
        firstD = (str(datesList[0][0])
        +'-'+str(datesList[0][1])+'-'+str(datesList[0][2]))
        lastD = datetime.date(datesList[len(datesList)-1][0]
        ,datesList[len(datesList)-1][1]
        ,datesList[len(datesList)-1][2]) + datetime.timedelta(days=1)
        combined = (firstD+'/'
                    +str(lastD.year)+'-'+str(lastD.month)+'-'+str(lastD.day))
        return combined

    def requestData(self,url,client):
        """
        Used for requesting data from url
        """
        print(url)

        if(client.client_name == 'met'):
            r = requests.get(url,auth=(client.client_id, ''),headers=self.headers)
            if r.status_code == 200:
                return r.json()
            else:
                raise Exception("Error: JSON status: {0}".format(r.status_code))



    def chunkIt(self,seq, num):
        """
        Used to divide a list into approx equal parts
        """
        avg = len(seq) / float(num)
        out = []
        last = 0.0

        while last < len(seq):
            out.append(seq[int(last):int(last + avg)])
            last += avg

        return out
```

```python
import client
import geometry as geo
import weather


client_id = "your client id"
client_secret = "your client secret"
email = "your email address"
password = "your password for netatmo if netatmo is chosen"
access_token = "your access token"
refresh_token = "your refresh token"


#1 Create client
c = client.Client(client_name="Netatmo",client_id = client_id,
                  client_secret=client_secret,
                  email = email,password = password,
                  access_token = access_token,refresh_token = refresh_token)

c_met = client.Client(client_name="met",
                      client_id = client_id,
                      client_secret=client_secret) #Met client

#1.5 Refresh your access token if you use netatmo
c.refreshAccesstoken_netatmo()  #Refreshing only for netatmo

#2 Create Geometry
g = geo.Geometry(latDeg=59.091,lonDeg=9.66,distance=10)
#10 km square with center at 59.091, 9.66

#3 Create Weather
w = weather.Weather(c,g)

#4 Get data
#Met
observation = w.getObservation(data='temperature',date='2017-01-01/2017-01-05')
observation = w.getObservation(data='temperature') #Netatmo
```