



Studieprogram: Informasjonsteknologi

Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo

Besøksadresse: Holbergs plass, Oslo

PROSJEKT NR.
2018-7

TILGJENGELIGHET
Åpen

Telefon: 22 45 32 00

Telefaks: 22 45 32 05

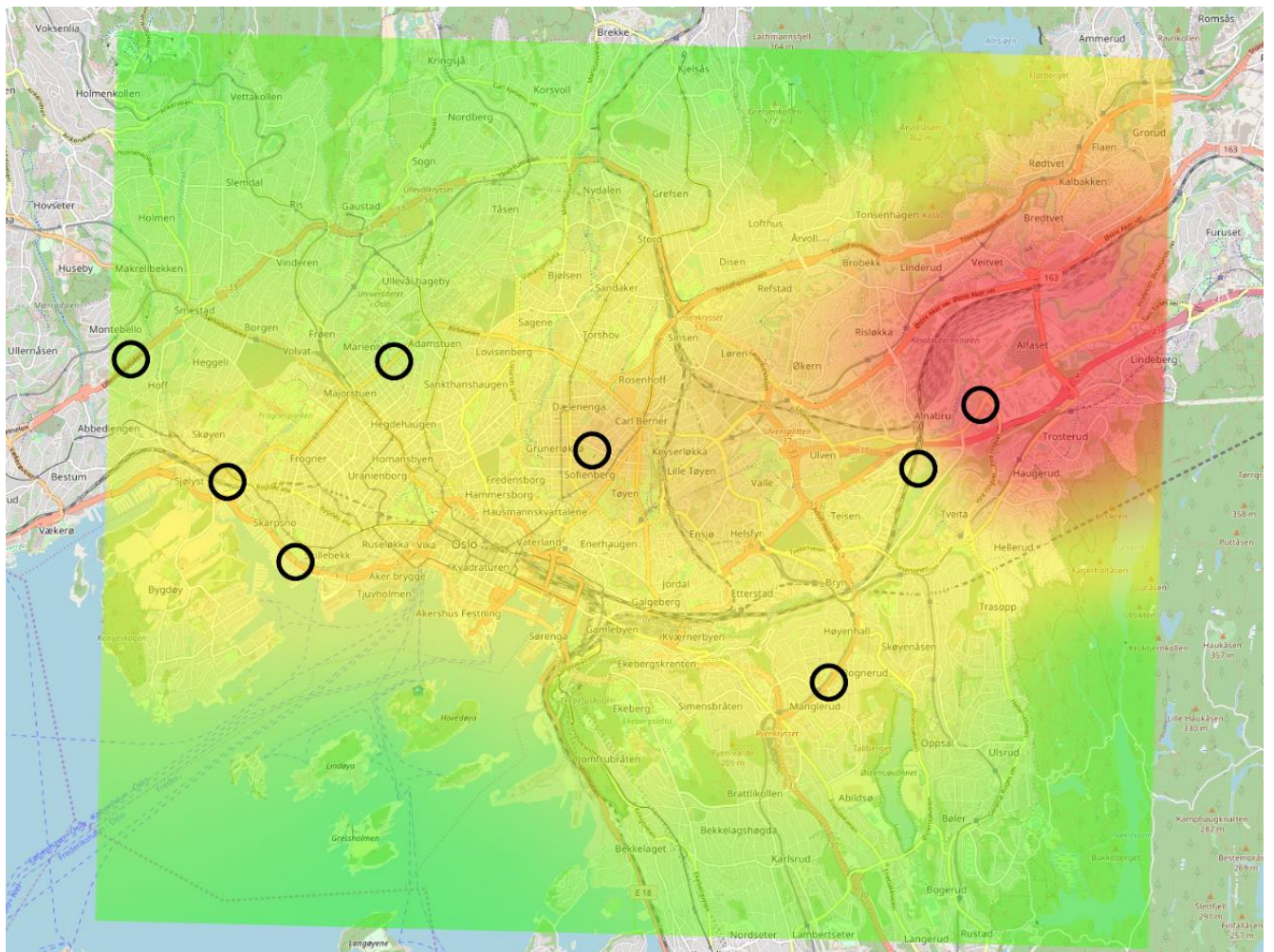
BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL Visualizing air quality and pollution in Oslo	DATO 2018-05-23
	ANTALL SIDER / BILAG 98
PROSJEKTDeltakere Skjalg Gustav Eriksen Michael D. Kardzhilov	INTERN VEILEDER André R. Brodtkorb
OPPDRAUGSGIVER IBM	KONTAKTPERSON Lars Hovind
SAMMENDRAG In this report, we document the bachelor project of creating a web application for visualizing air pollution in Oslo. We describe our solution, its technical quality, and the process of developing the application and how to utilize it.	
3 STIKKORD Air pollution / luftkvalitet	
IBM Cloud	
Interpolation / interpolation	

Visualizing air quality and pollution in Oslo



IBM Cloud



Skjalg Gustav Eriksen and Michael D. Kardzhilov

1 Foreword

In this report we document the process and results of our bachelor assignment. It has been a challenging semester, with new deadlines every week. This is our first time using an agile workstyle, and we often undershot our weekly goals. But through hard work and dedication we managed to achieve satisfactory results. It has been an exciting change to work with open source software, using their documentation and searching github for solutions.

The way we came into contact with IBM and this project was through the OsloMet “Prosjektforslag” page where clients can come into contact with students. We would like to thank IBM for giving us an interesting and challenging project to work with. They were very helpful, invited us to meetings and gave us a direct line of contact in case we had problems or questions.

During the project we received a lot of directional advice from our advisor André R. Brodtkorb who did a great job guiding us, and keeping us focused on what needed to be done. He inspired us with high energy and a positive outlook on a weekly basis during our meetings.

1.1 About us

We are a group of two students at the Oslo Metropolitan University (Oslo Met or OMU; Norwegian: Oslomet – storbyuniversitetet), **Skjalg Gustav Eriksen** and **Michael D. Kardzhilov** bachelor's degree in Software Engineering (Dataingeniør). We established the group fall of 2017 but we first met during our last year of highschool (videregående skole). We enrolled into Oslo Met together and have shared most of our classes since.

We had already completed multiple projects as a group during our studies which meant that we were familiar with each others strengths, weaknesses and limits. Using what we learned from our past projects we were able to cooperate very well with clear communication.



Skjalg Gustav Eriksen



Michael D. Kardzhilov

1.2 Reader's guide

The following report is written for a reader interested in computer science, but we have made an effort to make it so that it can also be appreciated by anyone who wants to get a general overview over our work. The report consists of six main sections.

1. In this first section, we establish what the document is, give thanks and introduce ourselves.
2. In the second section we introduce the client and their project proposal.
3. In the third section we establish the scope and requirements for the project.
4. In the fourth section we show our development process. Here we intend to show how and why we ended up with our current solutions and explain any mistake or dead end we encountered along the way.
5. In the fifth section we document our product. This section is similar to a product manual, but we also explain how our product works, in addition to how to use it.
6. In the sixth and final section, there is a summary for our project and documentation. In this section we go over what we were satisfied with, and what we learned.

Index

1 Foreword	2
1.1 About us	3
1.2 Reader's guide	4
2 Introduction.....	7
2.1 The client	7
2.2 The project and focus.....	7
3 Project scope and system requirements	8
3.1 Functional requirements.....	9
3.2 Non-functional requirements	9
3.3 User stories.....	9
4 Developmental process	10
4.1 Planning and method.....	10
4.2 IBM Cloud	17
4.3 Air quality data	18
4.4 Calculating air quality.....	20
4.5 Displaying our results	29
4.6 Managing data.....	42
5 Product documentation	52
5.1 Product description	52
5.2 Project scope and the product	52
5.3 Interface	53
5.4 Program setup	57
5.5 Running on IBM Cloud	60
5.6 Cloudant noSQL database.....	61
5.7 Program components	64

5.8 Further development	96
6 Summary	98

2 Introduction

Fresh air is something people associate with being environmentally friendly. Norway aims to be one of the leading green countries, which has led to increased interest in clean air. The rise of asthma and other air related ailments have also helped increase the interest in air quality.

Oslo municipality has taken many steps to motivate the use of alternative modes of transport, to and around the city. Cycling is one of the most environmentally friendly transport modes. One of the ways they have encouraged cycling is by adding a cyclist counter that displays how many cyclists have gone past a road¹. Another strategy the city has instituted to help alleviate the air pollution is a “dieselforbud” (diesel ban) in which they do not allow diesel vehicles on the road, when the air quality is very low.

2.1 The client

Our client is International Business Machines(IBM). They are a leading international technological company consisting of nearly 400.000 employees with about 1000 employed in Norway. They deliver various technologies, products and services, with a focus on cognitive technology and cloud based solutions². They are looking for solutions implemented in their cloud service IBM Cloud.

2.2 The project and focus

The project proposal had the following description.

“Etablere et datagrunnlag basert på dagens kilder relevante for luftkvalitet, topografi, trafikk og vær, utvikle en analysemodell for dataene basert på GPS-koordinater og utvikle en varslingstjeneste ut til smarttelefoner.”

From project proposal³

English translation:

“Establish a data foundation based on current sources relevant to air quality, topography, traffic, and

¹ <http://euroskilt.no/uploads/flyer-a4-sykkelteller-print.pdf> Accessed 4 May. 2018.

²Source: IBM’s project proposal, <https://www.cs.hioa.no/data/bachelorprosjekt/Prosjekter-2017-2018/HiOA%20IBM%20Bacheloroppgave%202.pdf> Accessed 4 May. 2018.

³ Source: IBM’s project proposal, <https://www.cs.hioa.no/data/bachelorprosjekt/Prosjekter-2017-2018/HiOA%20IBM%20Bacheloroppgave%202.pdf> Accessed 4 May. 2018.

weather, develop an analysis model for data based on GPS coordinates and develop a notification service for smartphones.”

We had a meeting about the project and general design before the bachelor period started to establish the scope of the project and hear what was important when developing a solution for the client. An important aspect of the project is that the client only has two requirements. The first is that it should be a project that has to do with air quality in Oslo. The second is that it will run on IBM’s cloud platform. This means that we had a lot of flexibility in our design choices.

Based on the project proposal we decided to put our focus on making a realistic model and making a professional looking representation. With respect to time and available sources we also decided to restrict the model to sources relevant to air quality. We omitted parts of the project proposal’s suggested functionality as we are a small group. Pivoting the project resulted in us focusing on a desktop environment instead of a mobile one with notifications.

3 Project scope and system requirements

The goal for this project is to create a system which generates a real time overview of air quality in Oslo. This will be accomplished by interpolating⁴ a dataset we have available. The dataset will be acquired using NILU’s⁵ open API which outputs data captured by measurement stations. There are stations in many parts of Norway, including 13 placed in different parts of Oslo.

The system will be implemented in IBM Cloud, and will function as a cloud native web app⁶. Other flexible requirements include displaying the air quality for your current location, and predicting how the air quality will develop forward in time. To guide us in developing this project, we have summarized the requirements as a set of functional and nonfunctional requirements. We have also designed a set of user stories that will work as project milestones and goals.

⁴ "Interpolation - Wikipedia." <https://en.wikipedia.org/wiki/Interpolation>. Accessed 14 May. 2018.

⁵ "NILU – Norsk institutt for luftforskning." <https://www.nilu.no/>. Accessed 14 May. 2018.

⁶ "Traditional versus Cloud Native Web Applications" 1 Sep. 2015, <https://datacenteroverlords.com/2015/09/01/traditional-versus-cloud-native-web-applications/>. Accessed 14 May. 2018.

3.1 Functional requirements

- The system should give a real-time picture of the air quality in Oslo.
- The system should be able to access historical air quality data.
- The system should gather data from measurement stations located in Oslo

3.2 Non-functional requirements

- The system should be developed using IBM Cloud
- The system should use a map as interface for the user to display air quality.

3.3 User stories

- A. As a person near an air monitor, I want to find out what the air quality is like here.
- B. As a person who has young children, I want to check the air quality near my children's school.
- C. As a person who rides a bike to work, I want to check the air quality on my route to work.
- D. As a person who rides a bike to work, I have multiple potential routes I can take to work, I want to check which one has the best air quality.
- E. As a parent with kids that are coughing a lot after going to festival the other day, I want to check how the air quality was there, to see if that's to blame for their coughing.

4 Developmental process

The following chapter will cover our development phases. Each subchapter discusses how we solved a specific aspect of our project.

4.1 Planning and method

Before starting the development we needed to plan out how we were going to set up our work structure, define goals and set deadlines.

We worked incrementally on a week by week basis where each week was a new sprint. To start a new sprint we would have a group meeting with our advisor. In the meeting we would discuss the progress towards the week's goals and the goals for the following week. The goals we set were decided together with the advisor. When setting the goals we discussed and proposed changes as well as direction for the project, while we kept in mind the milestones we had set. This workstyle did not directly follow a specific work approach such as SCRUM⁷ or the waterfall model⁸, but it did follow the core principles of an agile model.

- ***Individuals and Interactions*** over processes and tools
- ***Working Software*** over comprehensive documentation
- ***Customer Collaboration*** over contract negotiation
- ***Responding to Change*** over following a plan

Agile software development values⁹

With the nature of our work style being agile we expect to redefine and change our requirements during our project period.

⁷ "What is Scrum? - Scrum.org." <https://www.scrum.org/resources/what-is-scrum>. Accessed 14 May. 2018.

⁸ "Waterfall model - Wikipedia." https://en.wikipedia.org/wiki/Waterfall_model. Accessed 14 May. 2018.

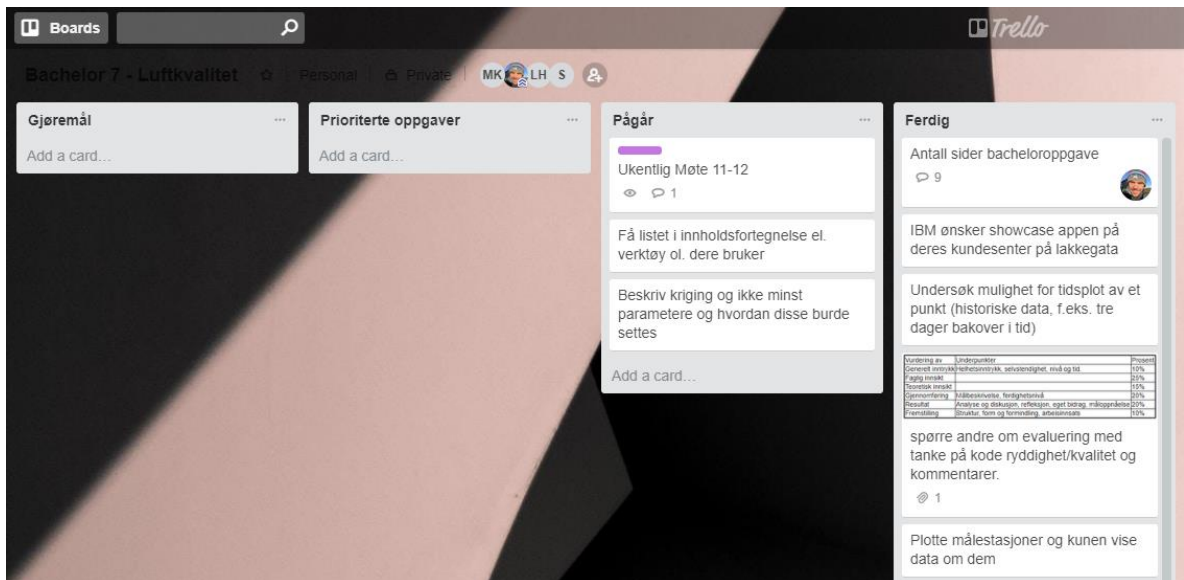
⁹ "Agile software development - Wikipedia." https://en.wikipedia.org/wiki/Agile_software_development. Accessed 14 May. 2018.

Trello

One of the main tools we used to stay organised and structured as a group was Trello.

“A Trello board is a list of lists, filled with cards, used by you and your team. It’s a lot more than that, though. Trello has everything you need to organize projects of any size.”

[-https://trello.com/tour](https://trello.com/tour)



Screenshot of our Trello board

Trello let us make a board that we could attach cards to. An example of a card could be “Write a short text explaining what Trello is”. That card would then be placed under one of our four lists, “Todo”, “Prioritised”, “In-Progress” and “Done” depending on its progress and importance. If we start working on it, we would move it from “Todo” to “In-Progress”, and once its done it would be moved to “Done”.

During our weekly meetings with our advisor we would review our weeks progress. If a card moved to “Done” was within an acceptable standard we would archive it removing it from the board. If however a card that was moved to “Done” did not meet the set standards, it would be moved back to “In-progress” for further development.

Some other functions trello offers include setting due dates, uploading pictures and marking cards with colours to distinguish them easier or mark impotence. Trello is useful for visualizing an overview over what we are working on each week.

Gantt chart

To get an overview of how much time we have and which aspects of the project need to get started or finished by a certain date we use a gantt chart.

The gantt diagram was named after Henry Gantt¹⁰ (1861–1919) who designed the chart around 1910–1915. Gantt diagrams¹¹ displays time in the X axis and tasks on different rows in the Y axis stretching out over time. Underneath is an example of a simple Gantt diagram.

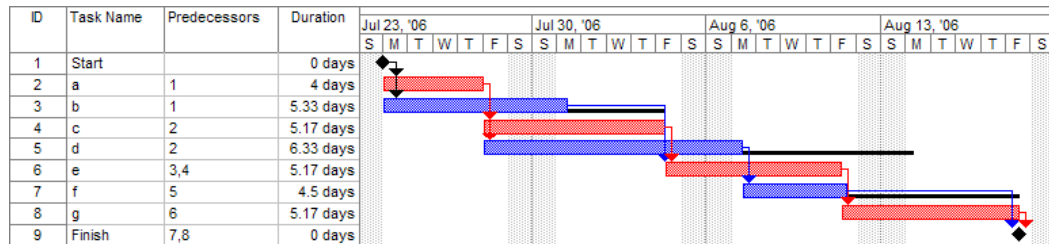
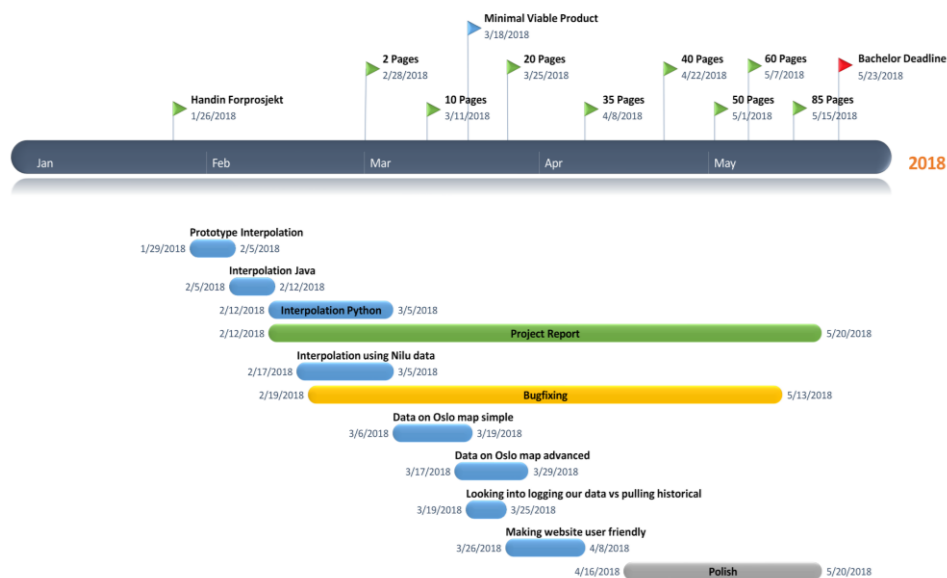


Illustration from wikipedia article¹²

We have developed a Gantt chart as shown below. According to our plan, we should complete the first prototype by March 18th. From March 17th to the 29th we will look into more advanced potential data sources such as traffic reports, to tie into our estimation. We will work on bug fixes and writing the documentation from mid February throughout the rest of the semester.



Our final gantt chart¹³

¹⁰ "Henry Gantt - Wikipedia." https://en.wikipedia.org/wiki/Henry_Gantt. Accessed 15 May. 2018.

¹¹ https://en.wikipedia.org/wiki/Gantt_chart Accessed 11 May. 2018.

¹² https://en.wikipedia.org/wiki/File:Pert_example_gantt_chart.gif Accessed 11 May. 2018.

¹³ Made using Microsoft Powerpoint.

Risk management

As a part of our risk management we identified risks our project could face and performed a risk analysis. A risk analysis is assessing the likelihood and consequences of each identified risk.¹⁴ We also wrote comments on each risk for how we would address them if they arose.

Writing a risk analysis before we started working gave us a different perspective on the task. It showed us what our biggest potential downfalls were. This let us prioritise critical aspects of the project so that we are prepared for potential setbacks.

Risk analysis

Risk	Probability	Severity	Comment/note
Planned estimates			Risk related to scheduling.
Incorrect time estimation for a task	Medium	Medium	Gross mistakes in the planned estimates could cause a lot of problems. To combat this we will have flexible plans and time periods dedicated to solve the eventual problems that can occur. If that does not work, drop the less important aspects or functions of the system to get a minimal viable product.
Incorrect time estimation for solving bugs or other system	Medium	High	To combat this we will have flexible plans and time periods dedicated to solve the eventual problems that can occur.

¹⁴ Sommerville, Ian F. (2016) *Software engineering* (10th edition). Edinburgh Gate: Pearson Education Limited. Page 644-645

defects			<p>If that does not work, drop the less important aspects or functions of the system to get a minimal viable product.</p>

Human			Risk related to people and EHS (environment, health and safety)
Medical emergencies	Medium	High	<p>Depending on the severity of the medical emergency it could derail our plans.</p> <p>To prevent this we need to make sure that the whole group is well versed in all aspects of the projects, so that they can continue development in the absence of others.</p>
Unplanned life events	Very low	High	<p>In case of serious unplanned life events members might need to leave the group.</p> <p>There is not much we can do to prevent this.</p>
Cooperation issues	Very low	Medium	<p>Preventable by following our timetable/calendar.</p>
Low morale	Low to Medium	Medium	<p>Low morale could slow down our progress.</p> <p>In order to prevent this we have</p>

	weekly meetings where we look at how far we have come and and set goals for the following week.
--	---

System Specification			Risks related to the projects specifications
Incomplete project	Low	High	In order to prevent this we will focus on the prototype first.
System specification changes	Low	Medium	<p>Changing what our system specifications are could cause us to lose a lot of time.</p> <p>To prevent this we will have a lot of contact with our advisor and client so that we all agree on the system specifications.</p>

Technology			Risks related to the technology which will be used during development
IBM Cloud issues	High	Low	Issues with IBM Cloud could be

			<p>critical since it is our development platform.</p> <p>To prevent this we will implement our soluciones locally before running them on the cloud. We will also be in contact with IBM personnel and a dedicated slack page for questions and help related to IBM Cloud for bachelor students. In addition to that IBM Cloud has great documentation and there are a lot of third party tutorials and guides online relating to it.</p>
Air data acquisition	Low	High	<p>If we have problems accessing measurement data as planned, further development could be halted. Making it a high severity risk.</p> <p>In order to prevent this we need to secure alternatives that have similar or related data which we could use instead in case problems arise. In a worst case scenario we could generate random data so that we can make a proof of concept.</p>
Loss of project data	Low	High	<p>If big parts of the project get lost or deleted it would set us back greatly.</p> <p>To prevent this we will utilise cloud services to backup our data, in case of local data loss.</p>

We did not have to take any major action for the risks we mapped at the beginning of the project.

4.2 IBM Cloud

As part of our agreement with the client, we will be developing our product to run as an IBM Cloud web app. IBM Cloud is a Cloud platform for developers that lets you develop apps and has a rich catalog of cloud services you can connect to your app. IBM Cloud is geared towards making apps for businesses and gives you a lot of options to scale your application. IBM Cloud is explained as follows by their platform documentation on github.

IBM's innovative cloud computing platform combines platform as a service (PaaS) with infrastructure as a service (IaaS) and includes a rich catalog of cloud services that can be easily integrated with PaaS and IaaS to build business applications rapidly.
[...]

From IBM documentation¹⁵

Through academic initiative we got access to trial accounts to use their service, but they are also offer Lite accounts for free, with some restrictions.

Restrictions

When we started the project we signed up for IBM Cloud Trial accounts using the IBM academic initiative¹⁶. Trial accounts are similar to the 2018 IBM Cloud Lite accounts, the main difference being that Trial accounts receive 2 GB of Cloud Foundry runtime memory while Lite accounts only get 256 MB.

IBM Cloud Lite accounts are restricted by:

- 256 MB of instantaneous Cloud Foundry runtime memory, plus 2 GB with Kubernetes Clusters.
- Access to usage capped plans for select services, such as API Connect, Watson Conversation, Watson Discovery, Internet of Things Platform, Data Science Experience and many more. Check out the full list of available services.
- Efficiency features, such as auto sleep and garbage collection, to help you better manage your resources.
- Usage tracking and cap alerts that notify you when you're approaching your data thresholds.

¹⁵ <https://github.com/IBM-Bluemix-Docs/overview/blob/master/ibm-cloud.md> Accessed 2 May. 2018.

¹⁶ "Watson - IBM Academic Initiative - OnTheHub."
https://ibm.onthehub.com/WebStore/ProductsByMajorVersionList.aspx?cmi_mnuMain_child=b3047d0a-2563-e611-9420-b8ca3a5db7a1&cmi_mnuMain=67016802-5765-e611-9420-b8ca3a5db7a1. Accessed 19 May. 2018.

A closer look

An important restriction listed is:

- 256 MB of instantaneous Cloud Foundry runtime memory, plus 2 GB with Kubernetes Clusters.

Cloud Foundry is an open source environment in which a web app runs¹⁸. Runtime memory translates to how much RAM an application can use. Kubernetes Clusters is a open source container system for deployment, scaling, and management of containerized applications¹⁹. Having 2 GB in a Kubernetes cluster would be the same as having 2 GB disk space for your application to use.

For our project running it on 256 MB memory should be sufficient but our development has been on trial accounts using all the available resources.

4.3 Air quality data

In order to create a picture of the air quality in Oslo, we need data relating to air pollution.

NILU

Norwegian Institute for Air Research²⁰ is a nonprofit institute. They have over 40 years of experience studying and analysing air pollution issues. NILU has a network of air measurement stations²¹ across Norway. These stations collect data on a variety of air components, and release this data using an open API which we will go more in depth about later.



NILU's official logo

¹⁷ "Introducing the IBM Cloud Lite account - IBM Cloud Blog." 1 Nov. 2017, <https://www.ibm.com/blogs/bluemix/2017/11/introducing-ibm-cloud-lite-account/>. Accessed 15 May. 2018.

¹⁸ "Agile Software Development - Application Platform | Cloud Foundry." <https://www.cloudfoundry.org/why-cloud-foundry/>. Accessed 15 May. 2018.

¹⁹ "Kubernetes.io." <https://kubernetes.io/>. Accessed 15 May. 2018.

²⁰ <https://www.nilu.no/OmNILU/tabid/67/language/en-GB/Default.aspx> Accessed 15 May. 2018.

²¹ <http://www.miljodirektoratet.no/Documents/publikasjoner/M358/M358.pdf> Accessed 15 May. 2018.

Air components

Air components refer to the different particles and gases that make up the measured air. NILU's measurement stations measure a variety of particles and gases. As of writing this report the air components measured in Oslo are:

- PM10
- PM2.5
- NO2
- CO

PM10 and PM2.5²² refer to particulate matter 10/2.5 micrometers or less in diameter. These air components can be very harmful in large quantities, but are generally harmless at normal levels. NO2 refers to nitrogen dioxide. The health risks NO2 poses are mostly limited to children, old adults and people with respiratory illnesses²³. CO refers to carbon monoxide, high levels of this gas can cause headache, dizziness, vomiting, nausea, unconscious and eventual death²⁴.

The components are measured by the measurement stations but not all stations measure all the different components. Most of the components have six or more stations which lets us gather enough data to create a reliable estimations of the pollution for that component. We have set our minimum measurements requirement to 4 for the sake of reliability, however this is a problem for CO.

CO only has two stations measuring it at most times, as of writing this report. This means we cannot reliably generate estimates for it. With hopes for an increase in the number of measurement stations in the future we have prepared our system to generate estimates for components that might get added as long as there have more than 4 measurements.

Other sources

Having spent many weeks searching for more air quality data, we were unable to find other reliable sources for Oslo. There were some private firms that had made headlines about tracking air quality in Oslo, but we were unable to achieve any meaningful contact.

²² <http://www.npi.gov.au/resource/particulate-matter-pm10-and-pm25> Accessed 15 May. 2018.

²³ <http://www.environment.gov.au/protection/publications/factsheet-nitrogen-dioxide-no2> Accessed 15 May. 2018.

²⁴ <https://ephtracking.cdc.gov/showCoRisk.action> Accessed 15 May. 2018.

4.4 Calculating air quality

To calculate a picture of the air quality based on NILU's data, we would consider each measurement station a point in a X,Y coordinate system. We needed to find a method for estimating the values in the gaps between our points.

Interpolation

To make an overall representation of the air quality in Oslo we need to interpolate the data we have. Interpolation is a mathematical technique to estimate a function based data points. This allows us to estimate the air quality in Oslo, based on the measurements collected by NILU.

This simple illustration is an example of how linear interpolation works in two dimensions. Given the data points A-H we can estimate the red line.

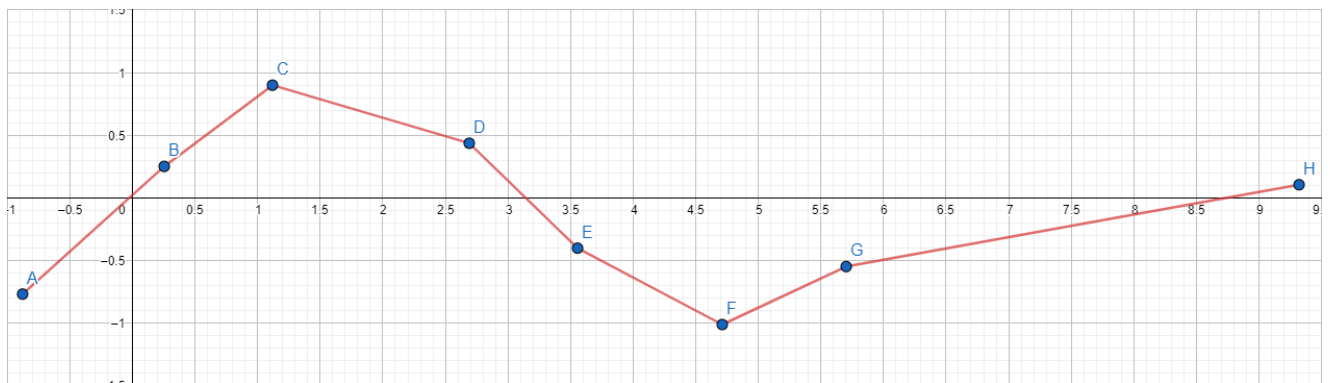


Illustration made in Geogebra

The data points A-H are taken from the sine function $F(x) = \sin(x)$, looking at them together we can see how our estimate compares to the function.

An important note is that we are estimating. We can not say that the red line is representative of the function that we gathered our data points from. Instead we can say that there is a likelihood that there are similarities.

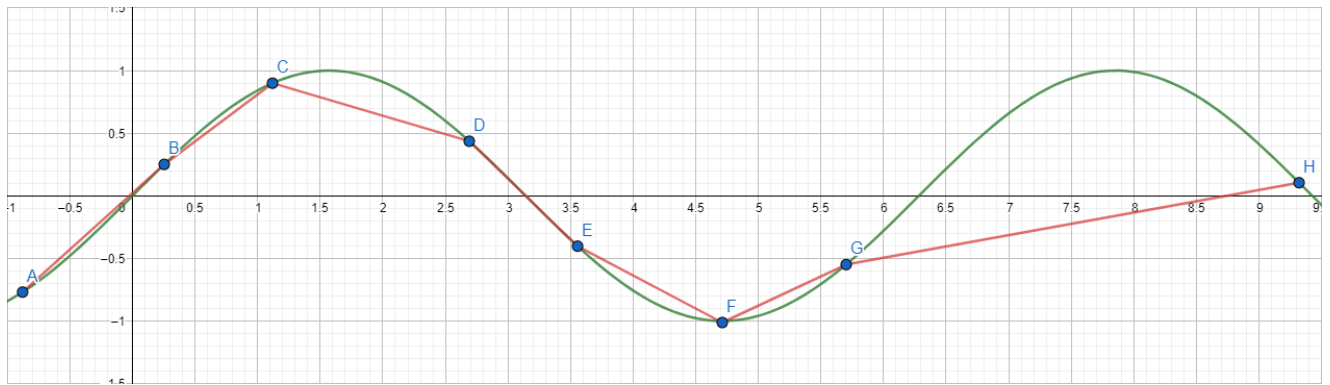


Illustration made in Geogebra

Looking at the actual function used represented in green, we can make some observations. One of the main ones is that the more data you have, the more accurate an estimate you can make. The right side having very few data points would make it hard to estimate correctly where the second peak of the graph is. The second is that we have a lot of sharp edges while our function does not. There is no way of remedying this in linear interpolation unless we have an infinite amount of data points.

Kriging

We use interpolation to estimate the air quality of Oslo based on the measurement station data from NILU, and we have used Kriging²⁵ to perform this interpolation. Kriging is considered to be the industry standard for our type of estimation. When estimating the air quality we want the highest precision possible. We have established that accuracy increases with the number of data points used, but another factor is the interpolation method. Picking the right method is highly dependent on what you are interpolating. For our project we are interpolating data in a geographic region. Geostatistics kriging²⁶ is an interpolation method used for spatial analysis and computer experiments. To implement Kriging in our solution, we have had to spend quite some time to study image coordinate systems (location for (0, 0)), the meaning of the different kriging parameters, and interpolation colormaps.

Kriging is explained as follows by Connor Johnson:

“Kriging is a set of techniques for interpolation. It differs from other interpolation techniques in that it sacrifices smoothness for the integrity of sampled points. Most interpolation techniques will over or undershoot the value of the function at sampled locations, but kriging honors those measurements and keeps them fixed.”

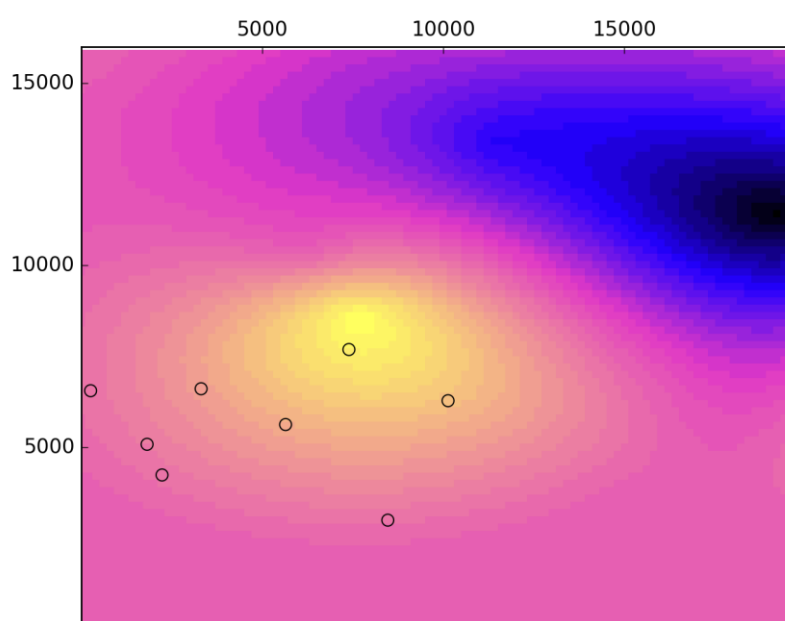
²⁵ "Kriging - Wikipedia." <https://en.wikipedia.org/wiki/Kriging>. Accessed 15 May. 2018.

²⁶ "Kriging - Wikipedia." <https://en.wikipedia.org/wiki/Kriging>. Accessed 13 May. 2018.

The last sentence in the quote is especially important. Knowing that the data from our sources is accurate we want to keep it the same in our interpolation.

To interpolate with kriging we use Conner Johnson's Kriging implementation under the MIT license, and we follow his example. For our case, each measurement station measures multiple components, and interpolating all the data we get from NILU's stations crashes the code. This happens because when it interpolates using kriging, it performs operations on the distance between the points, and having two points on top of each other, say value for PM10 and PM2.5 from the same station would be the equivalent to dividing by zero. To prevent this from happening we simply run the kriging on the data of one component at a time. This gives us an interpolated image of the air quality in Oslo for each component we run it on.

Kriging and its variables



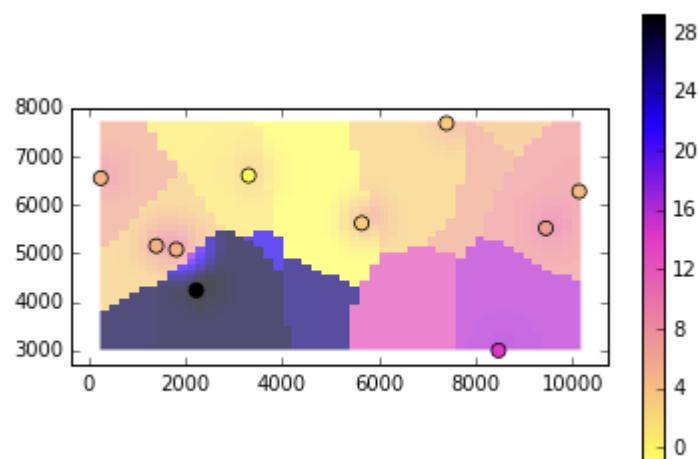
First result from data for PM10.

It's apparent there are a couple problems with this result. The most apparent problem that needs to be addressed is the placement of the points on the interpolated image. The points are the measurement

²⁷ Kriging <http://connor-johnson.com/2014/03/20/simple-kriging-in-python/> Accessed 13 May. 2018.

stations, meaning that they should line up with the hot and cold spots on the map. Another issue we notice is that even if we managed to align the points to the hot and cold spots through scaling and rotating the points, we still end up with many points that do not make sense or end up outside the frame.

Some hours of work and help from advisor shows that this happens because the orientation is wrong, and the first result we got is rotated then stretched. After working on the issue together with our advisor we finally managed to fix it, and he showed us a useful new tool called Jupyter Notebook²⁸. Jupyter Notebook is a Python web interface that allows for Python code to be executed segment by segment. Using notebook for our code we corrected the orientation and plotting, gave us the following result.



Result with fixed orientation, data for PM10

In addition to the orientation change we also lowered the resolution and accuracy. This was done in order to troubleshoot our changes faster, as our original parameters took roughly 50 min to finish calculating. The changes made the interpolation look more promising, and only took around 4 min to finish. Comparing to the first result all the colours matches up with the points on the image. While this new render is closer to what we need it is still not pleasing as the image is very blocky. We want a smooth transition from one colour to another instead of blocks of colours.

To improve our results we have to look at the kriging function.

```
def krige( P, model, hs, bw, u, N ):
    ...

    Input      (P)      ndarray, data
```

²⁸ "Jupyter Notebook." <http://jupyter.org/>. Accessed 22 May. 2018.


```

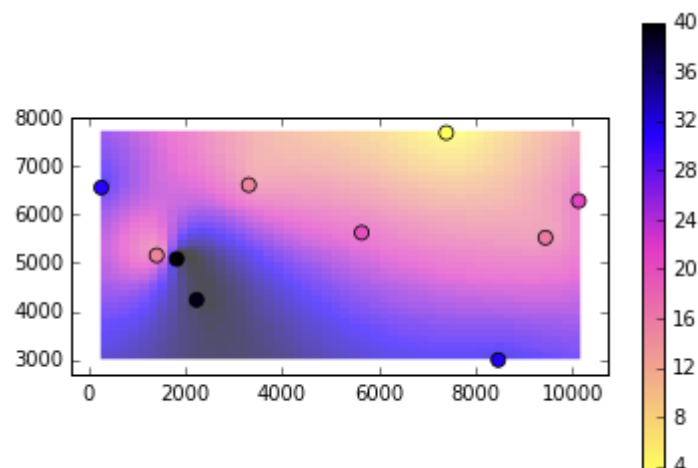
(model) modeling function
    - spherical
    - exponential
    - gaussian
(hs) kriging distances
(bw) krigin bandwidth
(u)  unsampled point
(N)  number of neighboring
      points to consider
...

```

Code snippet from krige.py

P is the input data, in our case it is the measurements from the NILU stations. The **model** refers to the function used to estimate correlation between points based on the distance between them. **hs** is a list of kriging distances. **bw** is the bandwidth of the distances in **hs**. **bw** and **hs** is used in the model. The **u** parameter is the single point or pixel you want to calculate. **N** is the number of other points around **u** you want to consider in the calculation. A thorough explanation of these parameters can also be found at Conner's website²⁹.

The blocky image only took two points into consideration when calculating. We increased **N**, number of points to consider up to a higher number. We also tuned the distances and bandwidth.

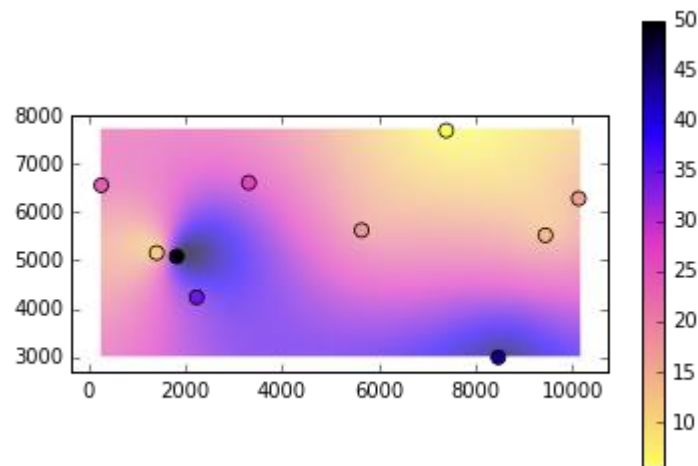


Result with improved kriging variables, data for PM10

The tuning of **N** had the highest impact on the end result, while distances had a minor impact.

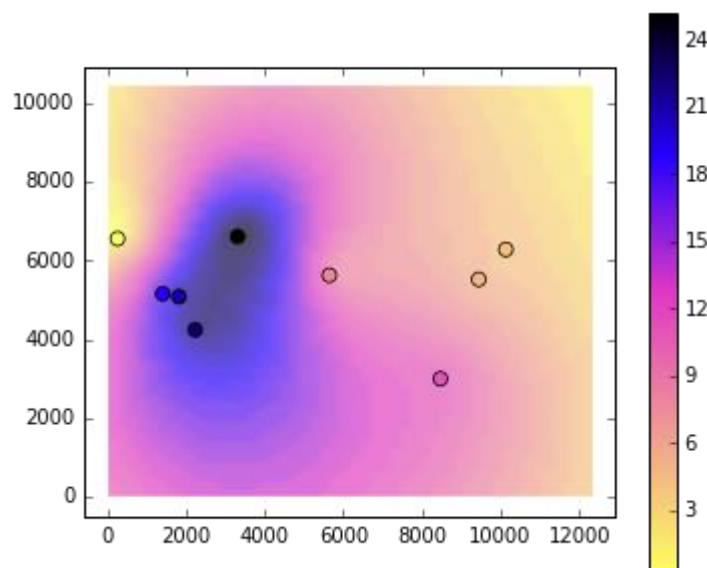
²⁹ Kriging <http://connor-johnson.com/2014/03/20/simple-kriging-in-python/> Accessed 13 May. 2018.

The image is still blocky to a degree, and we want to further smooth it out. To achieve this we could increase the resolution to an extreme amount which would take a very long time to calculate. Or we can use an inbuilt image interpolation from pylab, meaning that it smoothes out our resulted image when rendering it. This achieves a near identical result while being very time efficient.



Result with pylab image interpolation, data for PM10

The image extent is the size and area that is rendered. So far we have set the extent to be the min and max XY of all the points. To get an image covering the part of Oslo we are focused on, we need a larger extent. By setting the extent and resolution aspect ratio to reflect the area we mentioned in the mapping area section, we generated the following image.



Result after setting up the correct display area, data for PM10

Deciding the programming language

To program our kriging interpolation we need to pick a programming language. In the start of the development we were planning to use Java since we had the most experience in that language, as it was what we had been using in most of our programming classes. Looking into examples for kriging with Java yielded few results, with all of being focused on the mathematical aspect of kriging. Looking into kriging and interpolation using Python led us to examples complete with graphical renders of the data, which is exactly what we need for this project.

We decided that Python would be a good choice for our project. Looking at the options IBM Cloud gave us, it led us to use Python flask.

This was the first time we used Python, but IBM Cloud had a step by step process for getting started with Python in IBM Cloud and we quickly had a working hello world page running on the cloud.

NILU API

The NILU API provided us with a json object containing measurements from the stations in Oslo, it updates every 3 hours with current air measurements. To use the data we reformatted it into a Python DataFrame, which makes it easier to access the data we need in the calculations. DataFrame has a handy automated function to normalize json data. This meant that going from json to a DataFrame was seamless.

Testing the NILU API

```
@app.route('/data')
def data():
    # fetches data from nilu
    data = urlopen('https://api.nilu.no/aq/utd.json?areas=Oslo').read()
    # turns it into a json object
    jdata = json.loads(data)
    # put it into a dataframe
    fdata = json_normalize(jdata)
    Return fdata.to_html()
```

Code from testing phase

We tested the NILU API with the code block above. Using dataframes `to_html()` we get a HTML table we can display.

	area	color	component	eo	fromTime	index	latitude	longitude	municipality	station	toTime	unit	value	zone
0	Oslo	6ee86e	CO	NO0011A	2018-03-19T17:00:00+01:00	1	59.932330	10.724470	Oslo	Kirkeveien	2018-03-19T18:00:00+01:00	mg/m³	0.460810	Stor-Oslo
1	Oslo	6ee86e	SO2	NO0088A	2018-03-19T16:00:00+01:00	1	59.914870	10.763000	Oslo	Gronland	2018-03-19T17:00:00+01:00	µg/m³	2.500000	Stor-Oslo
2	Oslo	6ee86e	PM10	None	2018-03-19T17:00:00+01:00	1	59.921100	10.833630	Oslo	Breivoll	2018-03-19T18:00:00+01:00	µg/m³	5.633910	Stor-Oslo
3	Oslo	6ee86e	PM10	NO0011A	2018-03-19T17:00:00+01:00	1	59.932330	10.724470	Oslo	Kirkeveien	2018-03-19T18:00:00+01:00	µg/m³	6.454900	Stor-Oslo
4	Oslo	6ee86e	PM10	NO0073A	2018-03-19T17:00:00+01:00	1	59.922950	10.765730	Oslo	Sofienbergparken	2018-03-19T18:00:00+01:00	µg/m³	2.900000	Stor-Oslo
5	Oslo	6ee86e	PM10	NO0095A	2018-03-19T17:00:00+01:00	1	59.932550	10.669840	Oslo	Smestad	2018-03-19T18:00:00+01:00	µg/m³	13.434577	Stor-Oslo
6	Oslo	6ee86e	PM10	NO0101A	2018-03-19T17:00:00+01:00	1	59.941030	10.798030	Oslo	Rv 4, Aker sykehus	2018-03-19T18:00:00+01:00	µg/m³	2.813022	Stor-Oslo

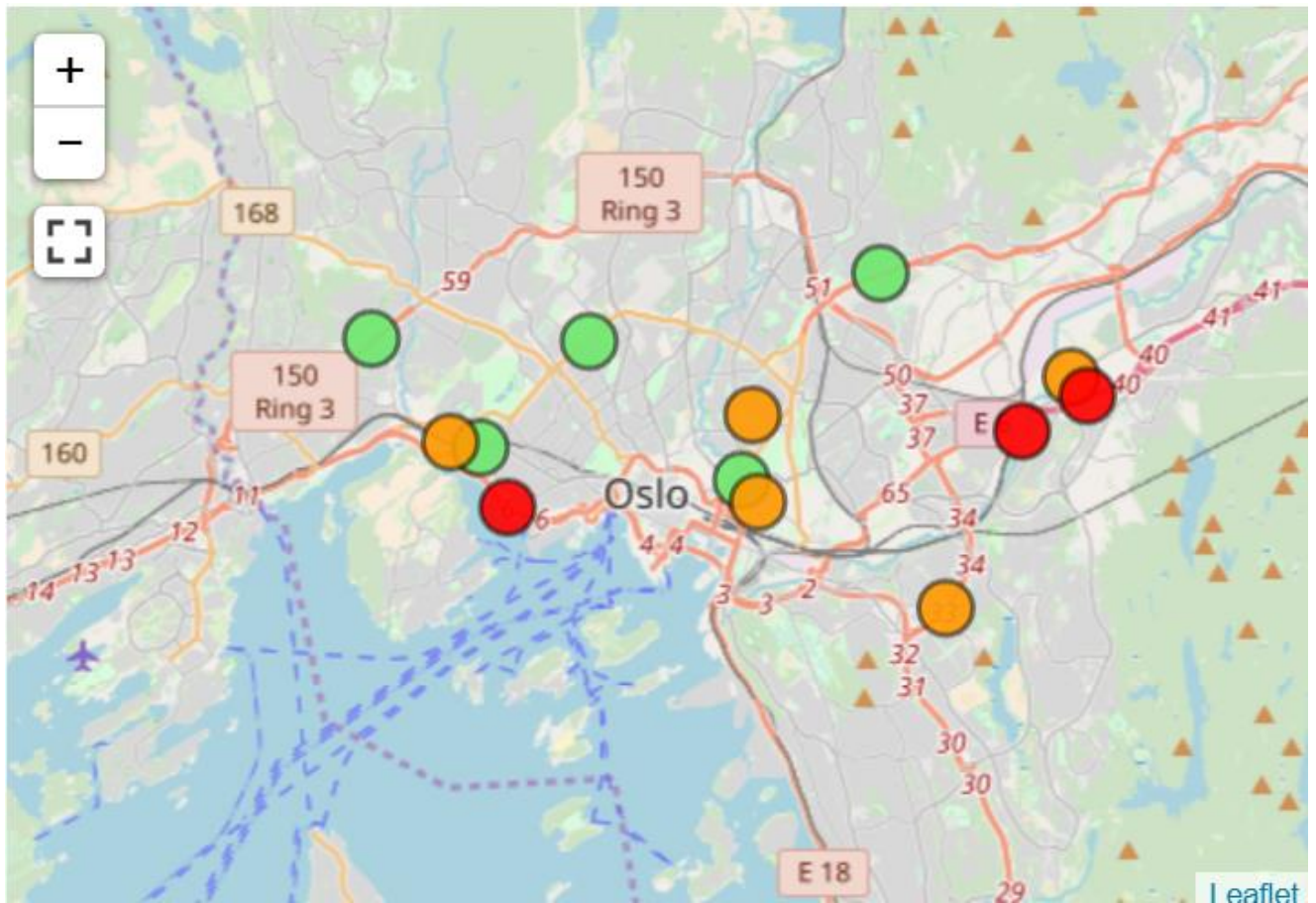
Screenshot of the first 7 rows in the output from testing code

Analysing the data

There are many things we need to consider when analyzing the data from the NILU stations and generating our map. One of the first things we needed to determine was the minimum and maximum values for the map, this refers to which values would generate the lightest areas on the map, and which would generate the darkest. The minimum and maximum values will be different depending on which gas or particulate matter we are measuring.

The way the kriging script seemed to generate the map originally was that it took the lowest input value and considered that to be the minimum value, and the highest input value to be the maximum value for the colours. This leads to some inefficient outputs for us since the colours were relative to the measurements of the day. For example if all the stations were reporting awful air quality but one was slightly better than the others, the kriging script would generate a map that looks mostly normal with some areas that you should avoid. On the other hand if it was a really good day for air quality with one station reporting just slightly bad air, it would make the output look very similar to our last example only this time it is painting a picture far worse than what is happening in reality. The solution for this was to manually set the variables called vmin and vmax. These variables control the relationship between colour and the values for our estimates. To make the colours informative we look up what levels of pollution constitutes health risks for each component. And used them as base reference to determine colours the user would be shown.

Another thing we needed to consider is the amount of measuring stations and their positions.



Screenshot from luftkvalitet.info³⁰

One factor is that there are only around 13 stations and they do not all measure the same things at all times, measuring NO₂ for example there are usually 8 stations that are measuring it. This means that there is only so much we can be sure about when we generate a map. There is not much we can do about this other than installing more measuring stations, but that would be expensive to purchase and maintain.

As we can see there are no stations at sea, and the closest one to the sea are near motorways that are constantly used by many cars. On most days this paints a picture that would suggest that if you are looking for healthy fresh air you are better off going to the heart of the city than out at sea. The same can be said for the woods north of Oslo since there are no measuring stations either.

There are multiple ways to go about creating a more realistic output. The way we decided to solve this is to put some extra fictive stations where we set measurements manually. At sea and in the forest for

³⁰ "Luftkvalitet.info." <http://www.luftkvalitet.info/home.aspx?type=Area&id=%7B48fd69aa-76f7-4883-8bbb-5ba79c3879ea%7D>. Accessed 22 May. 2018.

example we are fairly certain that the air quality will be better than in the city. We set a few fictive stations at strategic points and set their value to 40% lower than the lowest point in the city. This means that if there is really bad air quality one day, our outputs are not completely misleading.

Summary

To summarize; we calculate a picture of the air quality using kriging as our interpolation technique, to estimate the quantity of a component between the measurement stations. We make an educated guess for a couple of fictive points we place in strategic positions to improve realism of our picture. To program our kriging we use Python, and implement our solution in Python Flask as it is compatible with IBM Cloud. To get our data we will fetch it every 3 hours from NILU's open api.

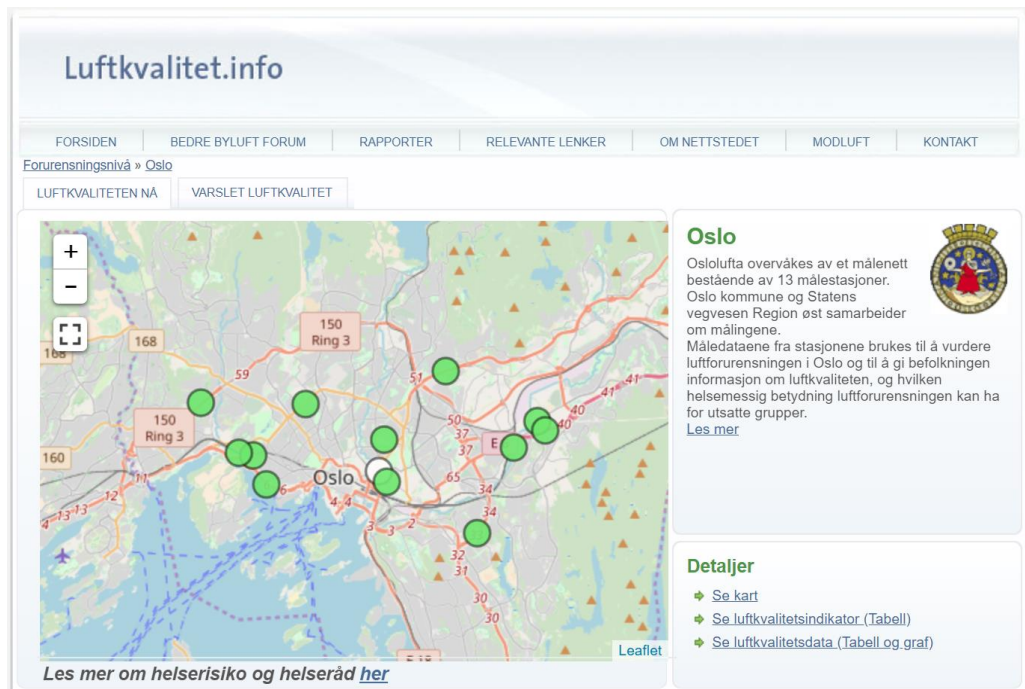
4.5 Displaying our results

Having calculated the air quality in Oslo we need to format our data in a way which would make it useful for our web app users.

Mapping area

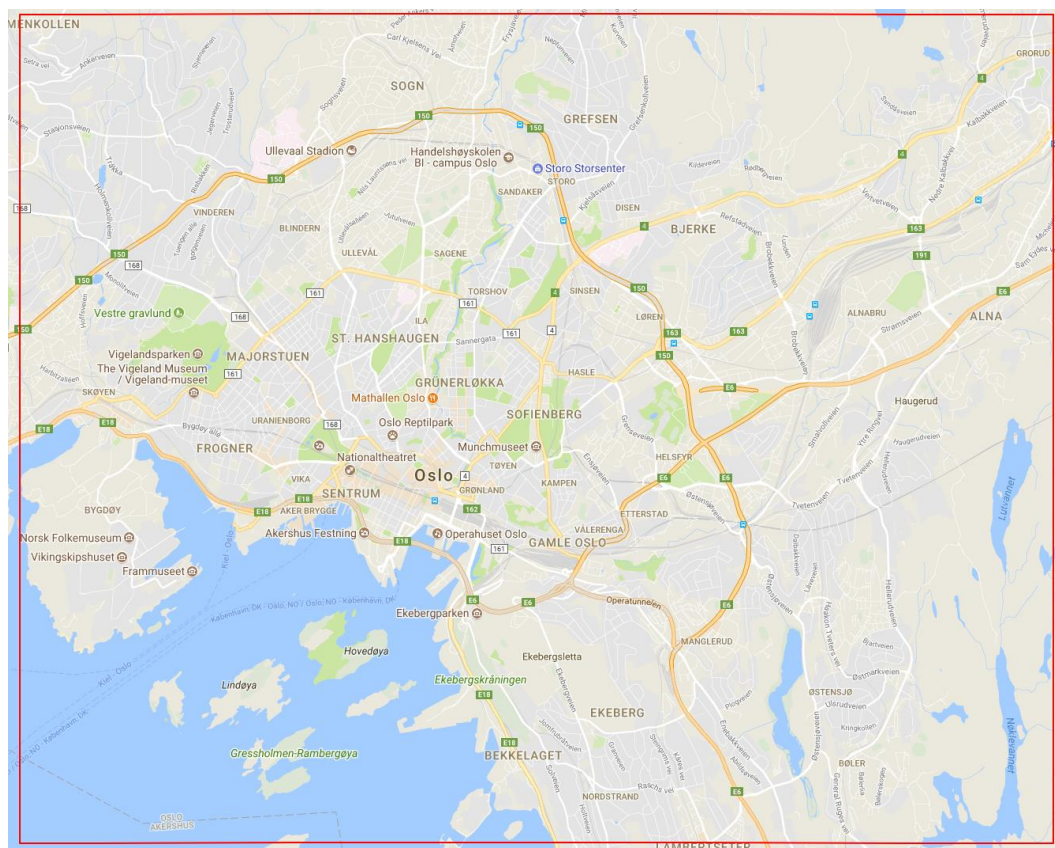
In this project we will be estimating what the air quality in Oslo is. In order to do this we need to decide on a map area that we would support and use to test. To decide on how big of an area and where to support, we begin with we taking a look at what the luftkvalitet website³¹. We used that to get an idea for where the measurement stations are. We needed an area that consists of Oslo, while at the same time having a measurement station close to it so that we could estimate the air quality with reasonable precision.

³¹ <http://www.luftkvalitet.info/home.aspx?type=Area&id=%7B48fd69aa-76f7-4883-8bbb-5ba79c3879ea%7D> Accessed 11 May. 2018.



Screenshot from luftkvalitet.info using the Oslo filter³²

Based on the stations locations we decided on using the following 10 km by 13 km area:



Screenshot of modified an official google maps API example³³

The coordinates marking the edges of the map are:

³² "Luftkvalitet.info." <http://luftkvalitet.info/>. Accessed 14 May. 2018.

³³ <https://developers.google.com/maps/documentation/javascript/examples/polyline-simple> Accessed 11 May. 2018.

Top Left 59.963886 N, 10.662291 E

Top Right 59.963886 N, 10.887139 E

Bottom Right 59.873800 N, 10.887139 E

Bottom Left 59.873800 N, 10.662291 E

The way we decided the coordinates was to start with deciding the top right and bottom left corners and then switching the latitude and longitude around to get the other two corners.

In order to generate an accurate visual representation of the area we modified an official google maps API example³⁴ making it draw four lines connecting our four corners, generating the image above.

GPS coordinates and utm projection

To do calculations on the data we get we have to put the values in a coordinate system. The locations NILU provides for its stations are GPS coordinates, and to put them in a coordinate system we can use we have to use projection.

GPS coordinates consists of latitude and longitude. Latitude tells us how far north/south a point is, while longitude tells us how far east or west it is. Some GPS coordinates also include altitude or elevation. In this case we don't have that data, so we will treat the earth as if its a smooth marble. Now if the earth was flat and not spherical in shape, all we would have to do is plot the latitude and longitude, treating them as x and y. Sadly the earth is not flat and so we need to to use something called projection to plot them.

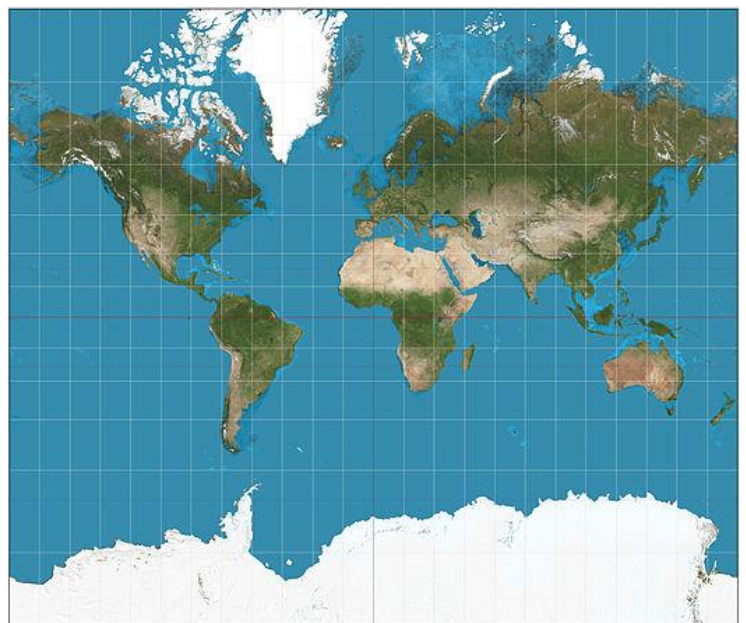


Image: Mercator projection of the world between 82°S and 82°N³⁵

³⁴ <https://developers.google.com/maps/documentation/javascript/examples/polyline-simple> Accessed 11 May. 2018.

³⁵ https://en.wikipedia.org/wiki/Mercator_projection Accessed 11 May. 2018.

Projection is when you take the surface of the earth and convert it into a plane. All maps created use some form of projection(unless its a 3d globe), because there is no way to display a sphere in two dimensions without distorting the image in some way. There are many ways to project a map, but you are always making some sort of compromise. One of the most popular/seen projections is the “Mercator” projection³⁶.

As we can see it is not the most accurate. The way this projection distorts the map is by stretching out the top and bottom. Looking at it one would think that the south pole is bigger than most of the continents combined, or that Greenland is bigger than Africa. When in fact that is not the case. To perform our projection we decided to use utm 0.4.2 which is a Python library for doing projection with a specific system called UTM-WGS84 which suits our needs. The alternative is the more standard Python library pyproj, but utm is simpler so we decided to stick with it.



Picture: UTM projection map³⁷

³⁶ https://en.wikipedia.org/wiki/Mercator_projection Accessed 11 May. 2018.

³⁷ <https://no.wikipedia.org/wiki/UTM-koordinater#/media/File:LA2-Europe-UTM-zones.png> Accessed 11 May. 2018.

Basemap

Basemap is a matplotlib tool for plotting geographical data. Here is a couple of results we got using Basemap.

component: PM10, date: 2018-03-03 08:01 component: PM10, date: 2018-03-03 23:57

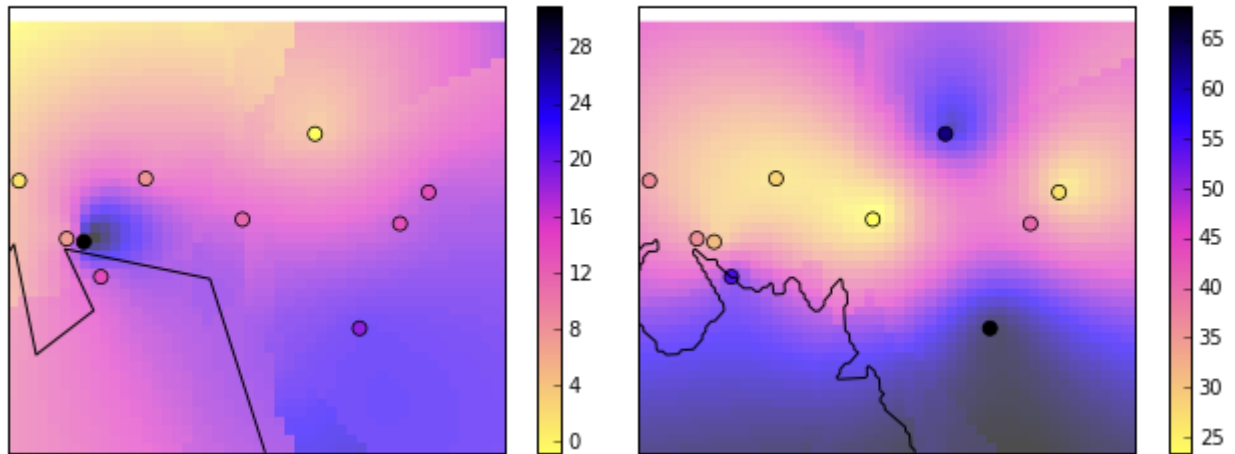


Image: basemap

Image: basemap high resolution

The image on the left is basemap at standard resolution, and the image on the right is basemap with the highest resolution available. The trade off for getting the high resolution is a big jump in time where basemap required half a minute to finalize while higher resolutions could take over 5 minutes.

As shown above basemap turned out to be quite limiting in its usefulness for us. Given the examples³⁸ we found, we expected to be able to get more detail on a zoomed in city level. Sadly basemap is not suited for our project since we are working on a small scale and not with bigger areas like the example shown.

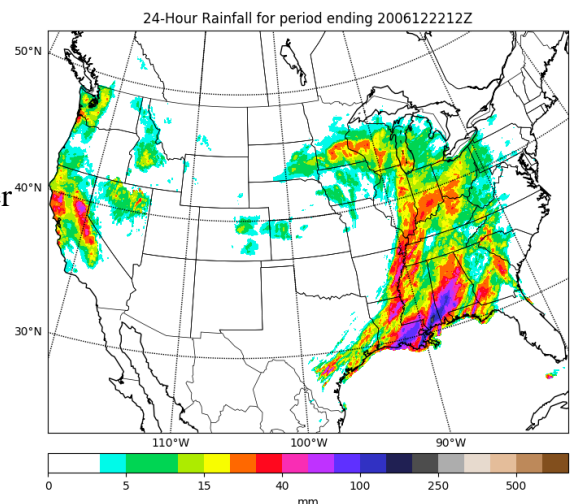


Image: Plot precip with filled contours³⁹

³⁸ <https://matplotlib.org/basemap/users/examples.html> Accessed 11 May. 2018.

³⁹ <https://matplotlib.org/basemap/users/examples.html> Accessed 11 May. 2018.

Mplleaflet package

When looking for other alternatives for displaying the map area we discovered Mplleaflet. Mplleaflet is a very useful tool for mapping points or routers. It is also the tool luftkvalitet.info⁴⁰, NILU's website are using. Bellow is an example of us using Mplleaflet to map the location of the stations and their readings for a specific gas type.

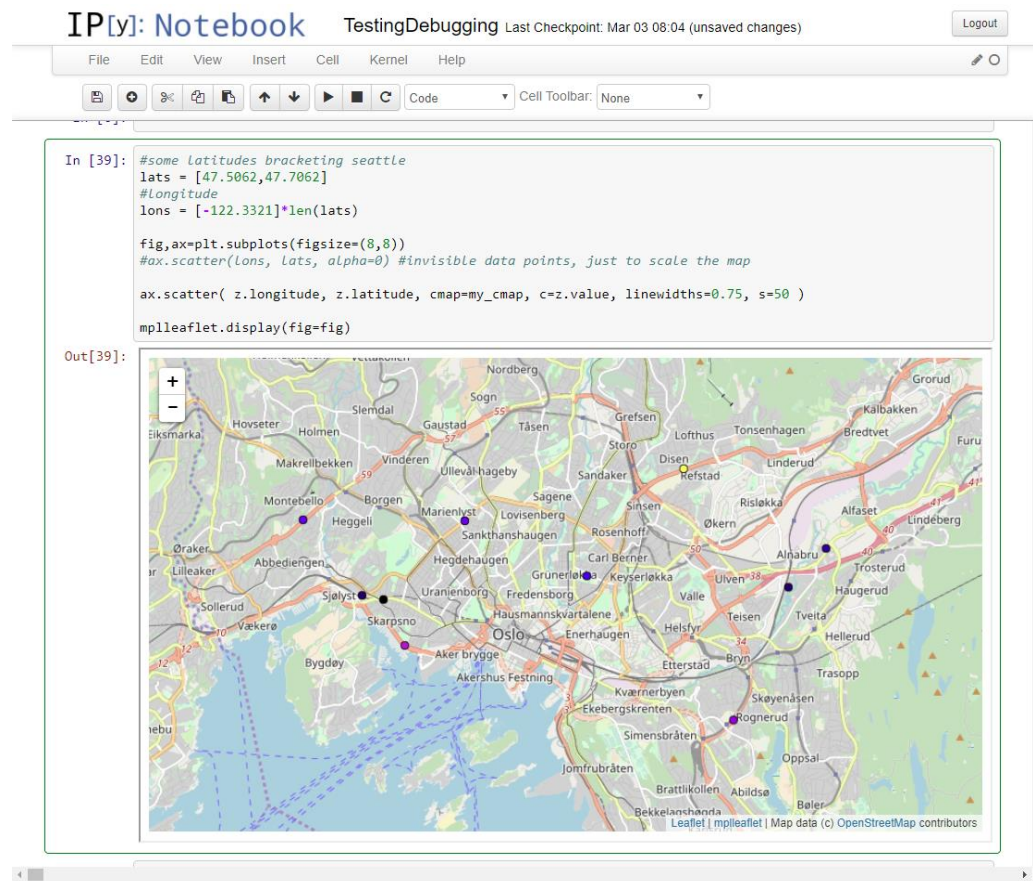


Image: Screenshot of plotting stations in Mplleaflet

Next is an example from their github⁴¹ where they plotted and mapped a route with points along the way.

⁴⁰ "Luftkvalitet.info." <http://luftkvalitet.info/>. Accessed 16 May. 2018.

⁴¹ <https://github.com/jwass/mplleaflet> Accessed 11 May. 2018.

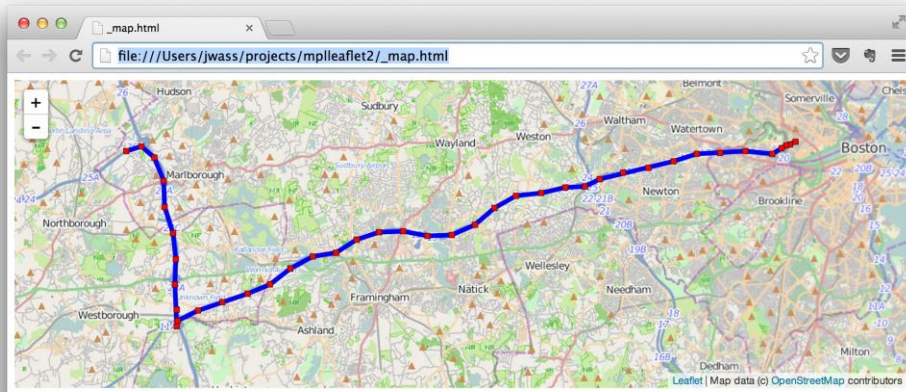


Image: from Mplleaflets github page⁴²

What we see is that Mplleaflet is good at points and lines, but it is suited for our project where we want to overlay a whole image over an area on the map. In february 2015 there was a request⁴³ to add the ability to overlay images like we require but it is still an open issue on github.

OpenLayers

After being disappointed with basemap we started looking for an alternative mapping tool. In one of our meetings, our advisor mentioned OpenStreetMap⁴⁴, an open source map initiative. That sounded promising so we started looking into the possibility of using it for our project. While looking into it we stumbled upon OpenLayers⁴⁵ which seemed to suit our needs perfectly.

OpenLayers is an open source web map initiative, that may sound identical to OpenStreetMap, but there are two main differences.

- The latest version of the OpenStreetMap is v0.9 from April 2009, while OpenLayers is in active development as of 2018.
- OpenLayers has over 160 great API examples for different functions and use cases, while OpenStreetMap has virtually none, in fact if you search for examples you end up getting linked to OpenLayers which is how we found it.

⁴² "GitHub - jwass/mplleaflet: Easily convert matplotlib plots from Python" <https://github.com/jwass/mplleaflet>. Accessed 16 May. 2018.

⁴³ <https://github.com/jwass/mplleaflet/issues/11> Accessed 11 May. 2018.

⁴⁴ "OpenStreetMap." <https://www.openstreetmap.org/>. Accessed 15 May. 2018.

⁴⁵ "OpenLayers." <https://OpenLayers.org/>. Accessed 11 May. 2018.

As the name suggests OpenLayers lets you generate maps with many layers of different information. The way we use it is having our interpolated data on top of the map followed by a layer for the contour lines if the user wants to view them, and finally a layer for plotting the measurement stations.

While OpenLayers has been great to work with, having great documentation and many examples there have been a few issues. The biggest issue we encountered was poor support for the SVG image format in image layers, we will explain more about why we used SVG images later. When using a SVG file in the image layer it does not scale properly. Instead of a smooth image it uses the lowest possible resolution making it extremely blurry and useless if text or sharp lines are involved. After working on it for a couple of days we found that there was an open issue with the same problem from over a year ago. Asking on Stack Overflow and posting an issue on github went mostly ignored with the only responses being “There is no good solution”. If we had more time we could have hosted our own version of OpenLayers where we patch this issue, but with our limited time and the complexity that it would involve we had to work around it instead. We opted for using other image file formats like PNG instead.

One of the OpenLayer examples we used was one which let us create a popup window when clicking on a map element⁴⁶. We wanted to use this feature to interact with the NILU measurement stations we would display on the map. This example utilizes bootstrap popover⁴⁷. Bootstrap popover and the popup example are very well documented so we managed to implement it into our project relatively quick. After working on the popups for a few days we noticed an issue. Clicking on a map element would normally bring up a popup. Clicking again, on a different element, would then trigger the following bug. With the second click, a second popup would appear and the first one would close. However, after a short time, the second popup would also disappear. This bug would only appear in the situation when clicking an element with an open popup.

Troubleshooting and scouting the web lead us to a stackoverflow thread⁴⁸. The most upvoted solution claimed that the reason for the issue was that popover requires at least 200ms between closing a popup and creating a new one. We implemented a delay like suggested but it did not completely fix our issue. Now instead of showing the new popup for a split second it required two attempts to bring

⁴⁶ Popup example in openlayers <https://openlayers.org/en/latest/examples/icon.html?q=popup> Accessed 11 May. 2018.

⁴⁷ <https://getbootstrap.com/docs/4.0/components/popovers/> Accessed 11 May. 2018.

⁴⁸ <https://stackoverflow.com/questions/27238938/bootstrap-popover-destroy-recreate-works-only-every-second-time> Accessed 11 May. 2018.

up a new popup. Looking more into this issue and possible solutions we concluded that it must be something that had been fixed in newer versions of Bootstrap. The OpenLayers example uses Bootstrap version 3.3.6 while the newest version available as of writing is 4.1.0. Attempting to update to the newest version of Bootstrap proved complicated as the syntax for generating a popup had changed. The syntax OpenLayers uses for Bootstrap differed from anything else we found online. In addition we could not find any examples of other Bootstrap versions being used with OpenLayers. Being quite frustrated we looked over the OpenLayers examples again. To our surprise we found another example which used the Bootstrap popover⁴⁹. The example created multiple popups without any delay or any of the issues we were facing. Comparing our implementation to the new example spotted one line which the original example does not use.

```
'animation': false,
```

Adding this to our implementation finally fixed our issue. We should note that we did not set animation to true, instead we had not specified it making it the default state. We could not find any other documentations of animation causing such issues so we documented our findings on stackoverflow⁵⁰.

OpenLayers extensions

OpenLayers is a big project used by many in projects that differ greatly. This has lead to many people making extensions for OpenLayers. These are often things that would not fit well with the core product, but very useful in niche circumstances.

One of the core features of most digital maps is the ability to search for a physical address on the map. Surprisingly OpenLayers does not support this functionality natively. Looking through the over 160 examples for any relating to “search” or ”searching” only gives us one result “Reprojection with EPSG.io Search”⁵¹ which lets you search for different map projections and displays them. You can also search for a major city/country/continent and it will try to give you a reprojection containing it. Searching Oslo will give you “(27391) NGO 1948 (Oslo) / NGO zone I”

⁴⁹ <https://openlayers.org/en/latest/examples/overlay.html?q=popover> Accessed 11 May. 2018.

⁵⁰ <https://stackoverflow.com/a/50361556/5383729> Accessed 16 May. 2018.

⁵¹ <https://openlayers.org/en/latest/examples/reprojection-by-code.html?q=Reprojection+with+EPSG.io+Search> Accessed 16 May. 2018.



Image: “(27391) NGO 1948 (Oslo) / NGO zone I” from OpenLayers Examples⁵²

As you can tell it does not contain Oslo, since it only shows the west coast of southern Norway. The best solution to this problem was ol-geocoder⁵³.

ol-geocoder is a geocoding extension for OpenLayers. Geocoding is the technical term for converting addresses, like street addresses into geographic coordinates. The extension lets you set up a number of options. The main one we looked into was geocoding providers, such as Bing, MapQuest and a few others, plus the ability to use a custom provider so that you can use any geocoding provider. These providers act as phone books for geocoding, so we wanted to find out which one worked best for us. Since Oslo is in Norway we tested out many different addresses and places in Norway using the different providers. In the end we decided to use OpenStreetMap. One of the main advantages of using it was that it supported the ability to lock the search results to a country or selection of countries. This helped us avoid situations like getting “Oslo Road” in England when searching for Oslo. Another advantage of OpenStreetMap is that it is one of the only options that does not require us to use an API key. An API key, also known as an application programming interface key is a way to identify who is using your service in order to restrict or grant access based on requirement. Most other

⁵² <https://openlayers.org/en/latest/examples/reprojection-by-code.html?q=Reprojection+with+EPSG.io+Search> Accessed 16 May. 2018.

⁵³ "ol-geocoder" <https://github.com/jonataswalker/ol-geocoder>. Accessed 11 May. 2018.

services require one and are limited to a certain amount of operations unless you pay for premium access. Not being limited by an API key grants us flexibility.

At the end of the project we wanted to add a sidebar or footer where we would have relevant text about our website. We also wanted this sidebar or footer to be retractable, so that we maximise the screen real estate for our web app. There are many ways to make retractable windows using scripts and CSS, but they all caused the same problem. If we extended/retracted and subsequently adjusted the size of the OpenLayers window, the OpenLayers window would not redraw the interface. This left us with either controls which were outside the window, or hiding under the extended sidebar or footer. This lead us to search for OpenLayers extensions that could give us this functionality. We found sidebar-v2⁵⁴ an extension which adds slidebars inside the map window of many map solutions.

Responsive design

We use Python flask templates⁵⁵ to insert data from our server into our webpage. Some of the data we send from our server includes the interpolated pollution in the form of a base64 string. To pull data from the server we would send HTTP GET requests to our server.

This was not very responsive or user friendly, because you had to load a new webpage every time you wanted to look at different data. This made us look for alternative solutions. We found that Python Flask supports javascript AJAX with JQuery requests⁵⁶.

AJAX with jQuery

Using the AJAX with jQuery was straight forward, but it interacted strangely with loops. If we looped through a range 0 to 5 and tried to use a jQuery “.get” request using the iteration variable it would use only the first value of it for all the requests the loop made. To fix this we had to “wrap” the “.get” request in a function call.

Colour maps

In order to decide the colour scheme of the interpolation output you use a colormap. A colour map also referred to as a colour table or a palette, is an array of colours used to map pixel data to actual

⁵⁴ "sidebar-v2" <https://github.com/Turbo87/sidebar-v2>. Accessed 11 May. 2018.

⁵⁵ "Templates — Flask 1.0.2 documentation." <http://flask.pocoo.org/docs/1.0/templating/>. Accessed 15 May. 2018.

⁵⁶ "AJAX with jQuery — Flask 0.12.4 documentation." <http://flask.pocoo.org/docs/0.12/patterns/jquery/>. Accessed 15 May. 2018.

colour values. The matplotlib library uses colour maps when generating images. Matplotlib also lets you make your own custom colormaps as they describe in their documentation⁵⁷.

The kriging example used the following colour map to generate the yellow- red - blue - black colour scheme.

```
cdict = {'red': ((0.0, 1.0, 1.0),
                (0.5, 225/255., 225/255. ),
                (0.75, 0.141, 0.141 ),
                (1.0, 0.0, 0.0)),
         'green': ((0.0, 1.0, 1.0),
                  (0.5, 57/255., 57/255. ),
                  (0.75, 0.0, 0.0 ),
                  (1.0, 0.0, 0.0)),
         'blue': ((0.0, 0.376, 0.376),
                  (0.5, 198/255., 198/255. ),
                  (0.75, 1.0, 1.0 ),
                  (1.0, 0.0, 0.0)) }
```

The first thing a user thinks when they look at this colour map might not be that some areas on that image might have higher quality air. Instead they might think that it all looked negative. To remedy this we read up on colour association to find a colour scheme better suited for our project.

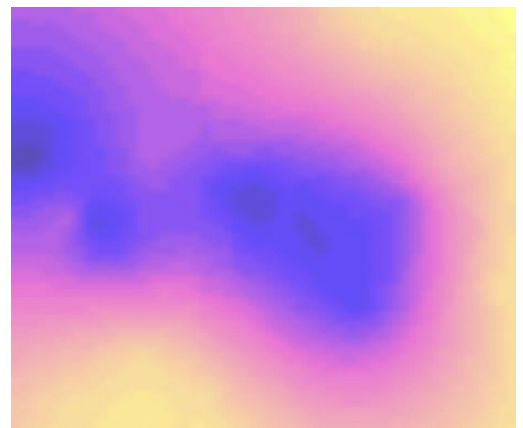


Image: Original colour map

Green is a colour associated with health and safety, this is used many places like for example traffic lights where green means that it is safe to go. Red is associated with danger and alarms, looking at the traffic light example again red indicates stop. To reflect this our colour scheme goes from green indicating safe air, to red indicating a potential health risk, with yellow as a middle ground.

⁵⁷ https://matplotlib.org/api/_as_gen/matplotlib.colors.LinearSegmentedColormap.html Accessed 11 May. 2018.

```

cdict = {'red': ((0.0, 0.0, 0.0),
                (0.35, 1.0, 1.0),
                (0.4, 1.0, 1.0),
                (0.8, 1.0, 1.0),
                (1.0, 2.0, 0.0)),

         'green': ((0.0, 1.0, 1.0),
                  (0.5, 1.0, 1.0),
                  (1.0, 0.0, 0.0)),

         'blue': ((0.0, 0.0, 0.0),
                 (0.1, 0.0, 0.0),
                 (0.4, 0.0, 0.0),
                 (1.0, 0.0, 0.0))
        }

```

Making this change has increased the intuitiveness and user friendliness, allowing users to understand what is happening at just a glance.

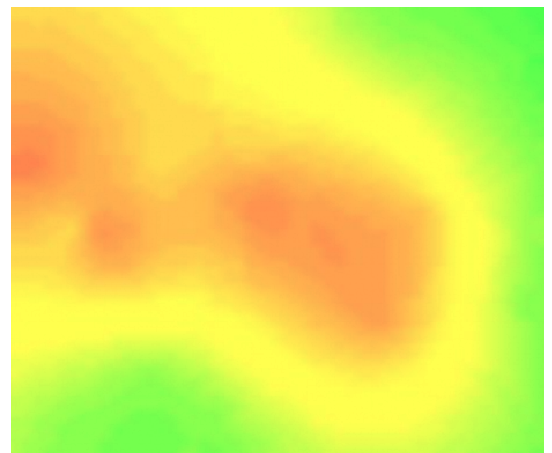


Image: Final colour map.

Summary

To summarize, we are going to cover a 10 km by 13 km rectangle containing Oslo. To put the measurement stations in a coordinate system we will use UTM projection, converted their GPS location to X, Y coordinates. The result we get by interpolating pollution is then displayed on top of a map using OpenLayers in our web app. The web app is made responsive with the use of javascript

AJAX and jQuery. To convey the level of air quality better we have made a colormap that goes from green to red.

4.6 Managing data

To address the requirements relating to historical data, we need to store some data and serve it to the web app.

Storing data in IBM Cloudant NoSQL

We need to store data about the interpolated pollution to display it to a user on a webpage. Recall that we start off with pollution data at specific measurement stations, and make an interpolation for the city of Oslo using kriging. Databases are often used to store such data in a structured manner, and there are classically two types of databases, so called relational and non-relational databases. Today the most used type is relational, it stores data in tables and manipulates the data with SQL. Non-relational also called NoSQL databases are a newer type, it is used where storing data in tables is not optimal. NoSQL has been developed with the needs of companies such as Facebook, Google and Amazon in mind, and is focused around bigger websolutions.

For Trial and Lite users IBM offers two database resources. One relational and one non-relational, called Db2 and Cloudant NoSQL. But for our region we only have access to Cloudant NoSQL. It uses a non-relational model with a document-oriented focus which stores JSON (JavaScript Object Notation) documents. This works out for us as we can benefit from the more flexible and web focused format.

As Trial users the restrictions we have on the use for our database are 20 lookups, 10 writes and 5 queries per-second, and a maximum size of 1 GB of data storage. It is important for us to keep these restrictions in mind when we design our application and backend, especially because the data size of our interpolation may potentially become very large.

Initially we stored image data in addition to metadata such as pollution component, and date. Component is the pollution type measured by the measurement stations, for example PM10 meaning we plotted data of particles with a size of 10 micrometers in Oslo. The date in the metadata signifies the time the kriging interpolation was performed. This date is typically very close to the actual measurement date. The image data consists of two separate images, one showing the interpolated

pollution, and one showing the interpolation in addition to the measurement stations, descriptive title, and colorbar. Below is an example of two such images.

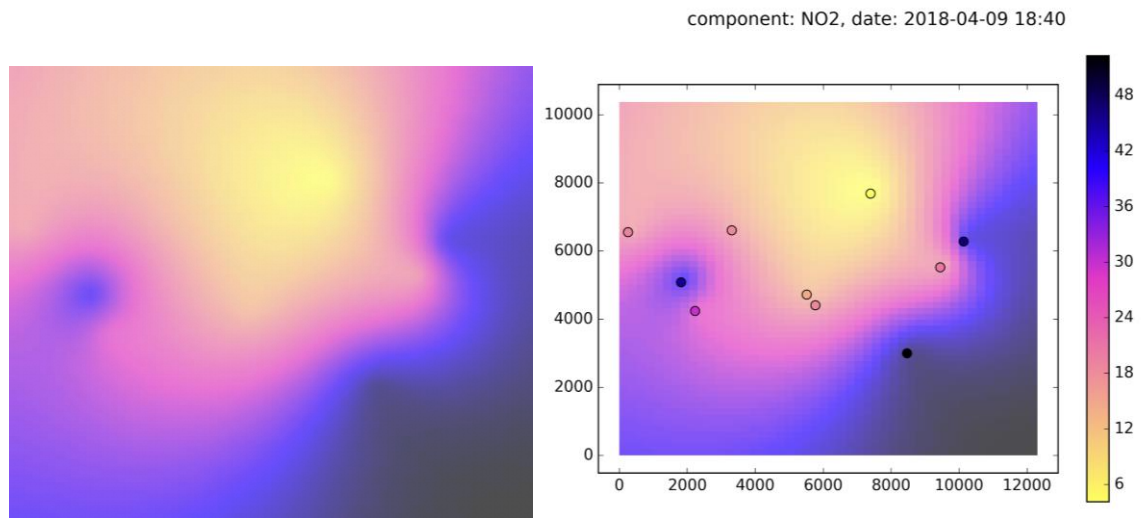


Image: Rendered without frame

Image: Rendered with descriptively information

Because we are storing JSON documents in our database, and JSON does not support images, we need to convert the PNG images we generated to text representation. Images can be stored as text if we convert them to a base64 representation. Base64 is a type of encoding that converts the binary image data to an ASCII string by using multiple characters to represent a bit code. This way we can store the images as base64 string representations in the JSON document.

An example of a JSON file that we stored in the database is shown below.

```
{
  "date": "2018-04-11 03:02",
  "component": "PM10",
  "_attachments": {
    "PM10_info": {
      "data": "iVBORw0KGgoAAAANSUhEUgAACUwAAAf..."
    },
    "PM10_img": {
      "data": "iVBORw0KGgoAAAANSUhEUgAACUwAAAf..."
    }
  }
}
```

JSON document from old database entry

Here, we see that the metadata are the “date”, and “component” sections, and the image data is stored as “PM10_info” and “PM10_img”. The actual image data are the strings starting with “iVBORw...”, which have been truncated here. Notice that all of the image data is in ASCII text. The base64 string “iVBORw0KGgoAAAANS” actually corresponds to the hexadecimal data 89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 48, which is the start of the PNG image⁵⁸.

Later we ran into some problems and this way of operating our database was limiting what we could do on the web app in terms of features we wanted to implement. To fix these issues we changed what we put into the database, instead of making the images and encoding them in base64 then storing them in the JSON document we would store the initial data and the numpy array with the kriging calculations we make with the kriging script. Then we can plot the data into the image we need on the fly as it gets requested.

To store the data frame we have our data and the numpy array with the kriged data. We used simplejson’s dump function to easily parse it into JSON we could just plug into the document. Then to get the data back we would use simplejson’s load function.

This is a JSON document stored into the database using the new method:

```
{
  "date": "2018-04-10 09:37",
  "data": "{\\"21\\":{\\"latitude\\":59.9211,\\"longitude\\":10.83363,\n    \\"value\\":41.1020888617,\\"unit\\":\\"\\u00b5g\\/\\"m\\\\"\\u00b3\\",\n    \\"component\\":\\"N02\\",\n    \\"x\\":9445.1961470791,\\"y\\":5519.4264017474}}, ...",
  "component": "N02",
  "krige_data": "[[1483.0, 1476.0, ... 1424.0, 1466.0, 1506.0]]"
}
```

Here is an example of what we can get out of one such document. The first line is the auto generated id. The following line is the component, followed by the date. The table is the data used in the kriging

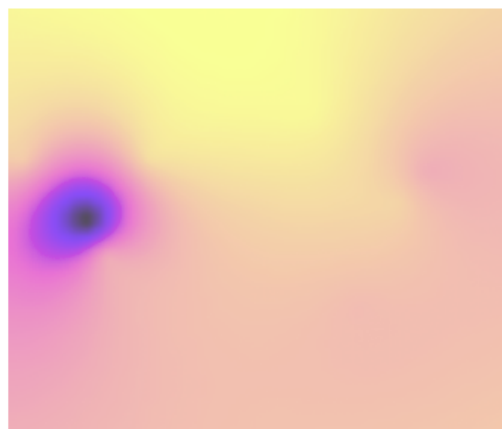
⁵⁸ Fun fact, all PNG images starts with the ASCII text PNG and the hex 50 4e 47 we got from the start of the hexadecimal data equivalent of our base64 string translates to PNG

and the picture is generated from the numpy array named kriging_data in the json document containing the kriging calculations with pylab.

```
19c30918088080b38d98932ab3789975
PM2.5
2018-04-10 12:14
```

	component	latitude	longitude	unit	value	x	y
12	PM2.5	59.92110	10.83363	µg/m³	3.791667	9445.196147	5519.426402
13	PM2.5	59.91132	10.70407	µg/m³	8.100000	2231.188986	4236.971741
14	PM2.5	59.93233	10.72447	µg/m³	2.504004	3310.899194	6605.790624
15	PM2.5	59.92295	10.76573	µg/m³	3.600000	5644.313939	5622.211051
16	PM2.5	59.93255	10.66984	µg/m³	4.588646	257.592415	6552.007552
17	PM2.5	59.94103	10.79803	µg/m³	0.301349	7394.997730	7683.827281
18	PM2.5	59.92773	10.84633	µg/m³	8.100000	10134.484938	6277.328925
19	PM2.5	59.89869	10.81495	µg/m³	6.400000	8469.460905	2995.554277
20	PM2.5	59.91898	10.69707	µg/m³	30.000000	1817.921401	5079.796788

```
<<cloudant.result.Result object at 0x7fe1aa5b1210>
```



Screenshot of output from Python notebook

So far we have stored the date and time when we gathered the most recent data and interpolated them, but we wished to store the exact time of the measurements. Reviewing the data from the measurement stations shows that the measurement stations listed two entries “fromTime” and “toTime”. We assume this is because the measurement is done in the interval between fromTime and toTime. Since toTime is the time and date when the measurement is complete we will use this as the time and date of the measurement.

After having a correct time and date for each entry to the database we could make a function to clean up the database of duplicate entries. This way it would be possible for multiple instances of the server to be running without pushing the same data to the database multiple times.

While on cleaning up the databases we also need to delete data that is old to make space for new data. We decided to keep 50 days worth of data, with each day being 400 kB. This is because with 50 days worth of data we can generate sufficiently accurate average numbers without risking going over the limits set by our Trial IBM Cloud accounts.

Web app portal

When we send data to the web app we want to send as much of our data as possible without slowing down the web app. With templates we insert the tables containing everything but the kriging data to the web app. For example a table containing all our data on PM10. We could use arrays here but we had a really good experience using Python pandas dataframe⁵⁹. This prompted us to look for a JavaScript equivalent we could use to structure our data in the web app. We found Dataframe-js which looked promising.

Dataframe-js

Dataframe-js⁶⁰ is a JavaScript library which gives you an immutable⁶¹ data structure to store your data in. It allows for extracting mathematical statistics about the data inside. We use the flask template to loop through our data and insert it into a Dataframe-js. Dataframe-js was really useful but we had some struggles.

Not being used to Immutable objects, caused us to stumble a lot. Where you would with an usual object change it with functions the dataframe-js being an immutable object returns a new dataframe instead. This meant that we needed to update our reference to the new dataframe our function returned.

Dataframe-js had little flexibility in the options it gave for changing a cell value with a given row and column. To combat this we made our own helper function that did operations on the dataframe and returned a new one we could update our old reference with. Dataframe-js provided more options for changing a column value opposed to a row value. This meant that when we wanted to change a cell we decided to turn the dataframe 90 degrees to utilize the column options. Then finding the cell and inserting a value.

⁵⁹ "pandas.DataFrame — pandas 0.22.0 documentation." <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html>. Accessed 16 May. 2018.

⁶⁰ "GitHub - Gmousse/dataframe-js: A javascript library providing a new" <https://github.com/Gmousse/dataframe-js>. Accessed 16 May. 2018.

⁶¹ Immutable meaning that you can't change the objects data after defining it.

Error handling

When Dataframe-js encountered a problem it would throw a JavaScript runtime error stopping the rest of our JavaScript. To address this we used try catch on some critical areas. These areas were where the data contained inside the dataframe would vary and sometimes be undefined. In JavaScript when an object is undefined, meaning the object is created but does not have any value. If we caught an error we would write the error message in the console with JavaScript `console.error()` function and use an empty dataframe instead. We would also later adopt lots of “ternary if”⁶² tests to our variables. If we found them to be undefined, meaning something went wrong we would update the reference with an empty dataframe. We made a helper function for supplying an empty dataframe and an empty dataframe row where we needed it.

Data compression

Our application serves images of the air pollution over an internet connection to a user. A user will typically look at several time instances, and perhaps several different pollution components. This results in a large number of files. We did not consider until late into the project the size of the data we were using and presenting. For example, the images of the interpolated data we were using to visualise air pollution were around 1.5 to 3 MB to begin with. This meant that it took a noticeable amount of time to load them in, significantly degrading the user experience. This was especially noticeable if we were showing a time lapse slider/animation, which would have been more than 15 MB in image files alone for a modest 10 frames. This is an important point for mobile users, as visiting this website would actually be a significant download, straining their data plan. It is also very important for us to consider because our lite user account only has 1 GB of storage.

While prototyping and working on the project in general we did not realise that this could be an issue, as we were both working with high speed internet connections. The first improvement we made was to stop saving the rendered images to our database, instead saving the interpolated “DataFrame” that is used to generate the images. The DataFrame contains the actual interpolated pollution in addition to metadata such as component, date and each of NILU’s measured values used to interpolate. This means that it takes a little longer for the client to receive the image, but in exchange the server needs to have a lot less data stored. We chose to go this route because the time it took to generate an image (100ms) to be negligible, and the fact that we have a 1GB limit on our Trial IBM Cloud accounts. The change from storing image data to storing the DataFrame resulted in a 20kB file instead of the

⁶² "Ternary conditional operator - ?: - Wikipedia." <https://en.wikipedia.org/wiki/%3F%3A>. Accessed 17 May. 2018.

previous 1.5 to 3 MB per timestep. In addition, this change enable a lot of flexibility, for example the ability to generate multiple different visualizations of the same data. This could be contour plots and different colormaps, as well as different resolutions and different file formats.

Changing the internal storage in our database did not solve the issue with a large download for users. To combat this we started looking into Python image compression and optimization options. The simplest way to compress the image size was for us to look at the PNG compression algorithm we were using. The PNG compression in Python has an optimize feature that can decrease file size. For our PNG images the change was miniscule, and the best result we managed to get was a 200 kB decrease in file size.

Because our initial compression experiment showed only a modest compression ration, we started looking at other alternatives. The main reasons for the large size of our files are:

1. The resolution/DPI of our images.
2. The PNG format.

We can reduce the DPI/resolution but that means that the image quality degrades significantly. We have therefore decided to address the compression format instead. PNG was created as a lossless image format, meaning it is supposed to exactly preserve all details of an image, even minuscule ones that are hardly noticeable. Since we are dealing with images that do not have many sharp edges we do not suffer to much from losing some detail, as our human perception of such loss is miniscule.

An popular alternative to PNG compression is to use JPG. JPG is a lossy format meaning that it enables us to remove some detail in an image in exchange for significantly smaller file size. If we examine the following interpolated image in detail, we see that the JPG compression artefacts are hardly noticeable.

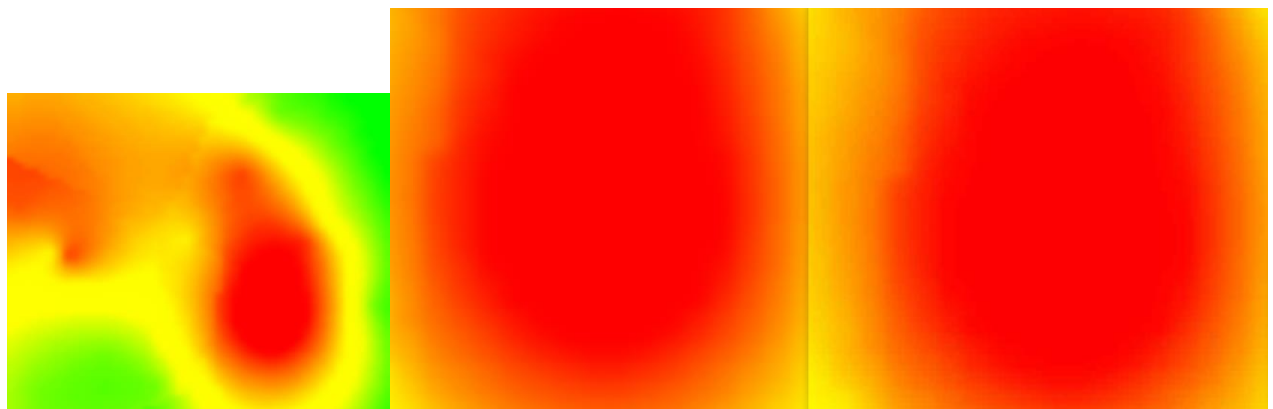


Image: Whole render

Image: 100% quality, closeup

Image: 40% quality, closeup⁶³

The left image shows the whole interpolation, whilst the center and right image show a closeup.

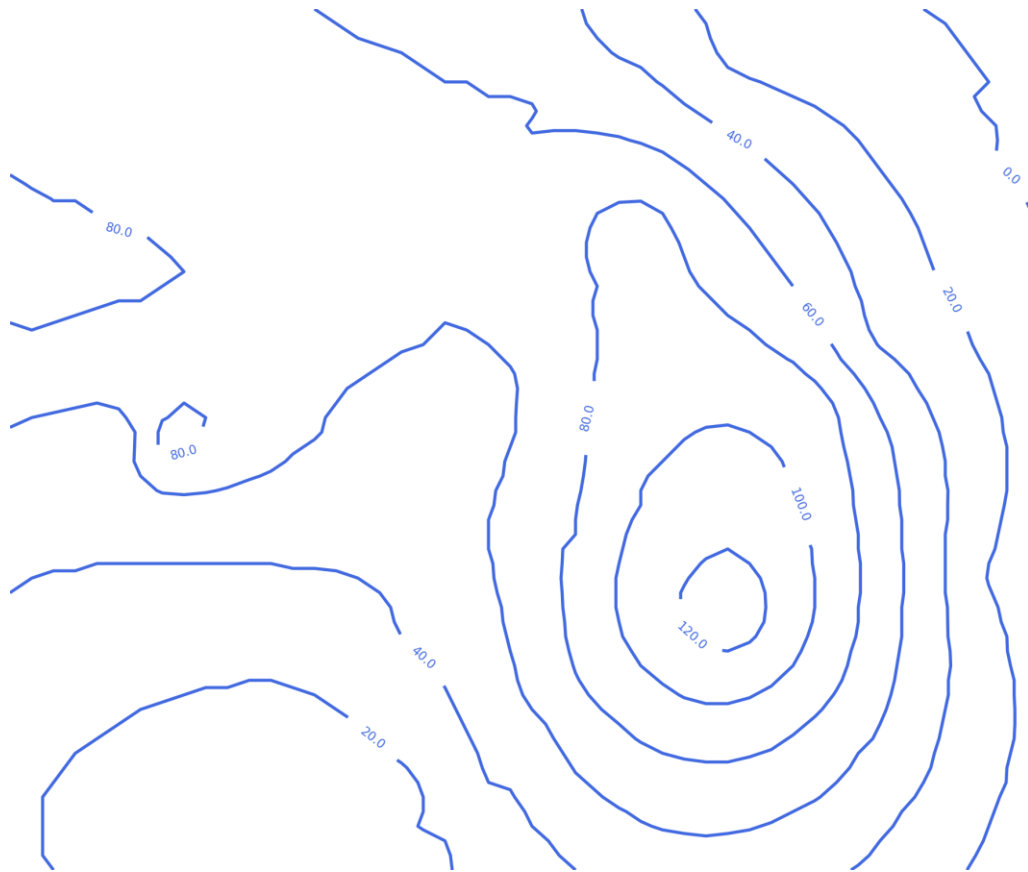
The center image is stored at 100% quality and the right at 40% quality.

As we can see the image on the left looks a lot smoother in its transition from red to orange compared to the right. In exchange for that loss in quality we ended up with a lot smaller files, 550 kB for the 100% image and 33 kB for the 40% image.

Sadly with JPG the only thing we give up on is not image quality, we also lose transparency. This is not a problem when overlaying our interpolated image of pollution such as the ones used in the example above, Because OpenLayers lets us decide transparency. However it becomes a problem for our contour lines.

When you have contour lines you want them to be sharp so that the text is legible. The way we implemented contour lines also involved having them as a separate layer in OpenLayers on top of the interpolated pollution. If we did this with JPG it would mean that the text would not be sharp, and in addition the background would be a solid colour instead of transparent, making the map underneath look washed out.

⁶³ Differences between 100% and 40% might also be due to not capturing the exact same area.



Thankfully the generated PNG contour line images are mostly transparent, this means that even though the image might be large, the file size can stay small since there is little data represented on it. Our transparent contour line images ended up being roughly 180 kB, which is acceptable. If for some reason it was not small enough the way we would have attempted to fix this would have been to either have the contour lines and map, in the same image. Another way to solve the problem would be to use vector lines and text on OpenLayers to draw the contour lines there, but that would be a lot more complex than simply overlaying them like we do now.

Later having discussed file formats we found that matplotlib supported SVG file format. SVG or Scalable Vector Graphics is a very compact file format because they do not store pixels, instead they store vectors used to generate an image. With respect to efficiency SVG proved vastly superior to JPG and PNG for render times and file size. Openlayers supported SVG image files but in newer versions of Openlayers they botched support SVG scaling. This made it impossible to store our contour plot with SVG as the text would be blurry and illegible. However our interpolated pollution does not suffer much from using SVG in OpenLayers.

We decided to use the SVG format for the Interpolated image of pollution and PNG format for contour plot.

Summary

To summarize; We are storing our calculations and metadata in Cloudant NoSQL. In our JSON documents we include the initial measurements used for the kriging, our kriging calculations, date and component. We will generate a picture of the interpolated pollution on the fly as it gets requested from our web app. We use compact formats such as SVG for the interpolated pollution as it won't suffer, but keep the PNG format where detail is important like contour plot.

5 Product documentation

In the following chapter we will present the final product that resulted from our development. We will break down the products functionality, and how it compares to the project scope. We will then go over how you can access the source code and set it up to run on IBM Cloud and locally. To sum up we will go over the different back- and front-end components with relevant code examples.

5.1 Product description

The purpose of our program is to represent the air pollution in Oslo on a colour coded map. The pollution will be divided into different components using data gathered by NILU and serves it from the cloud in a comprehensive manner. The colormap will use red as a indication of high pollution while green will be used to indicate low pollution with a set value to determine high and low points for each component. The different NILU stations will be present on the map and will display different relevant information when clicked. In addition to this the user will have available to them a view of the historical data.

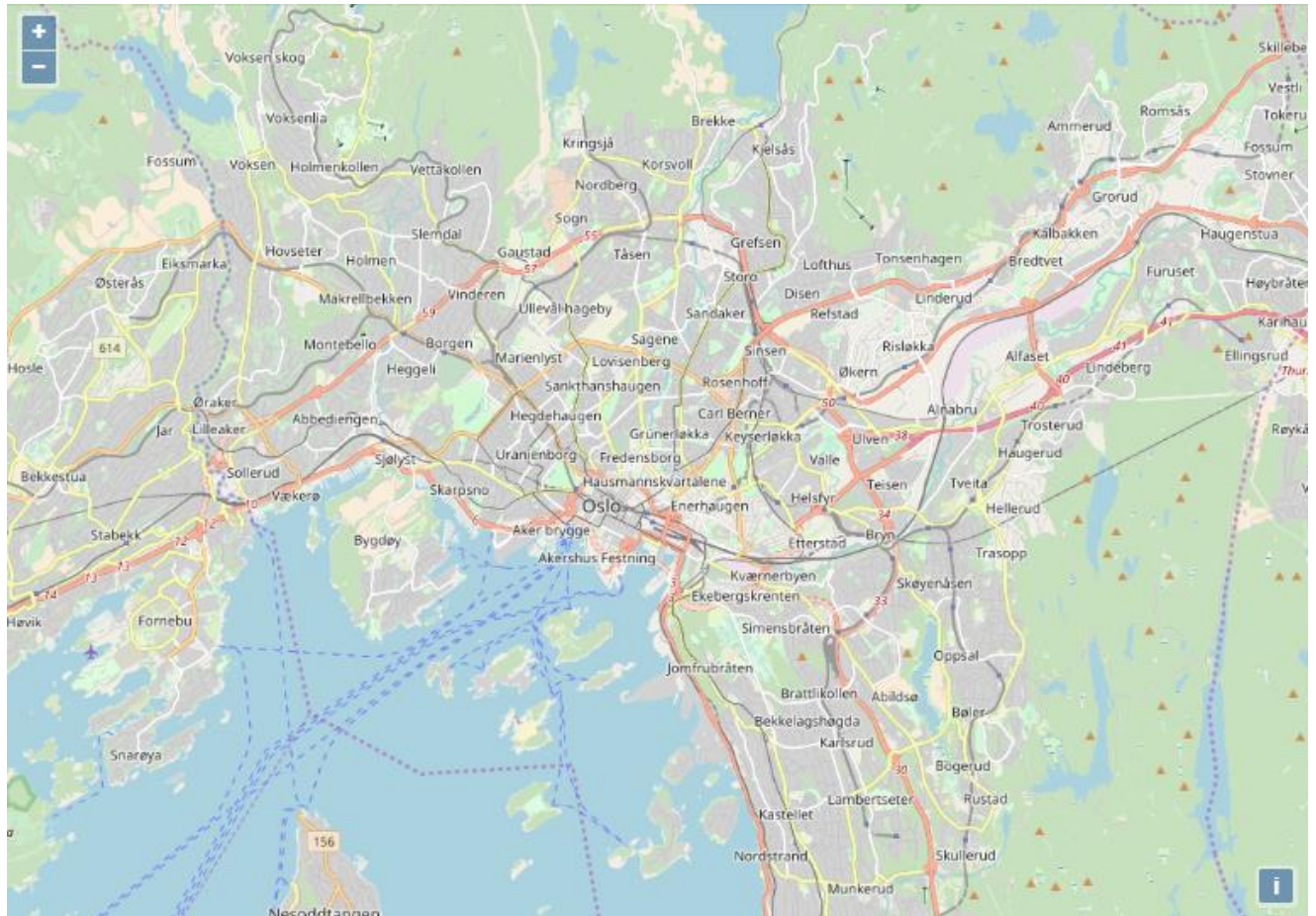
5.2 Project scope and the product

In chapter 3 we discussed the project scope and set a couple of requirements we wanted the system to fulfill, all of which as been satisfied. We also made some user stories to act as milestones and goals. We addressed the user stories A, B, E and to some degree C. User stories C and D are related to evaluating air quality along a route. In user story C you only need to evaluate one route, as such we can say that our geolocation function covers this. When geolocation is turned on your view is focused on where you are every time your location changes it updates it on the map. If a user were to ride a bike while having geolocation turned on it would update their location on the map thereby informing them of the air quality for the route they take. User story D involves comparing different routes, this is not something we implemented.

During the project period we decided to implement additional functions which we decided were good goals based on our progress at the time. This includes plotting clickable stations on the map that could display statistics, a sidebar with related information, a fullscreen button and contour plot for easier comprehension of the map values.

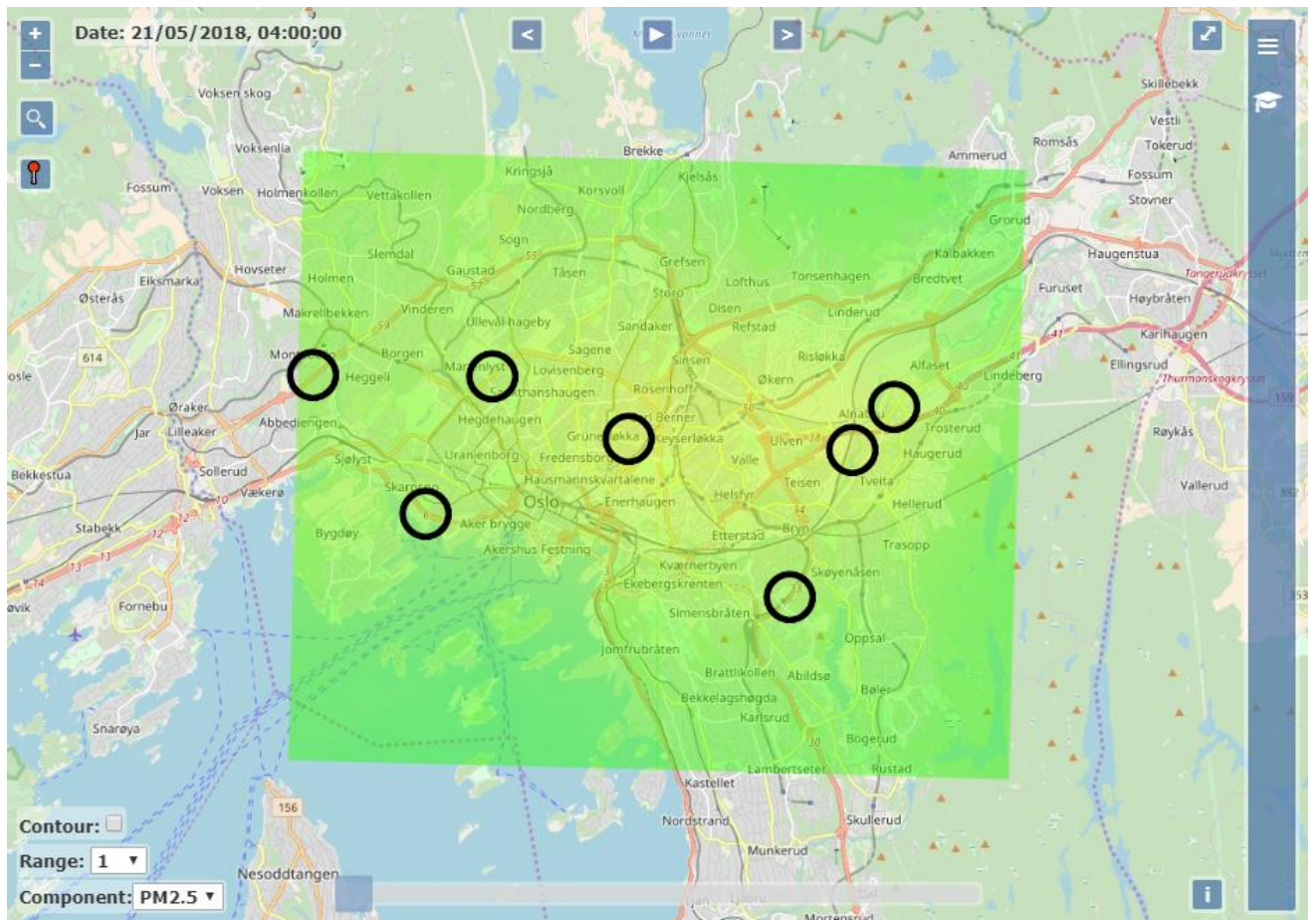
5.3 Interface

To properly display all the elements of our project and to give the user options we need to adjust the default OpenLayers interface.



Screenshot of OpenLayers interface with Oslo in focus

Above is an example of the basic default interface which OpenLayers provides. It consists of the map plane, zoom buttons and the “i” button. The “i” button is used for copyright and license notices, by default it redirects to the OpenStreetMap “Copyright and License” page.

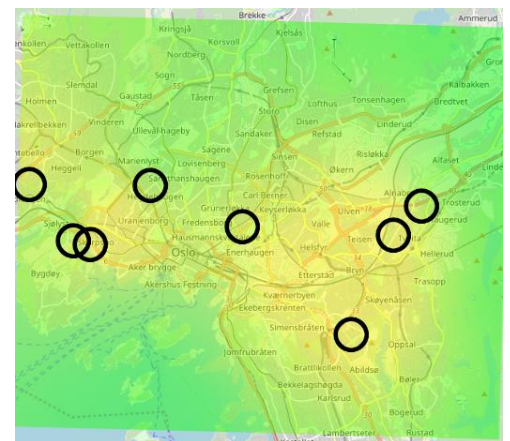


Screenshot: Our website's interface

This is the result of our interface modification. For our project a number elements are essential to the interface.

The main element a user will notice is the layer on top of the map used display our interpolated measurements. This is a image file that we reproject onto the map so that it is correctly aligned.

The black circles each represent a measurement station. The reason they are hollow circles is that it ensures that the user can see the value under keeping the map obstruction to a minimum. The hollow part of the circle actually has a less than 1% black alpha layer. The reason for that unnoticeable detail is that OpenLayers is set up to not detect clicks on transparent parts of PNG images.

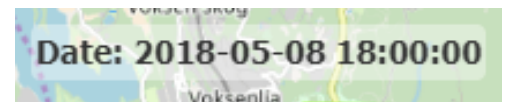


In the bottom left we have controls that establish the desired component, range and style. The contour checkbox enables and disables the contour outlines on our interpolated measurements image. Contour lines help establish where the transitions are from one value to another. Range is a variable the user

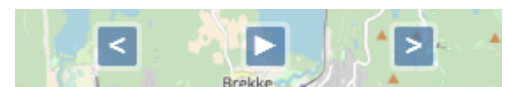
can set which decides how many renders to load in. The selection is dynamically made. It allows the user to pick any number from 1 to 60 which is divisible by 5. Component refers to which air component they want to display an interpolated image for. The component selection is also dynamically generated, so that if in the future more components start being measured they will show up in the list automatically.



The date and time are when the data for the currently displayed interpolation was measured.



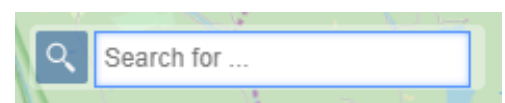
The buttons at the top middle of the window control the currently displayed interpolated image. If the range selection is set to 1 these buttons do not function as there is no other renders to display. When the range is set to a number higher than 1 the “<” button changes the interpolation to a more recent one up to the newest. The “>” button changes the interpolation to an older interpolation up to the range max set. The middle button is a play/pause button that automatically cycles through the interpolations up to the last one at the speed of 500 ms per frame.



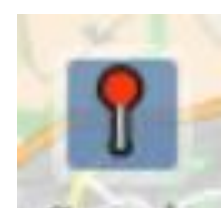
At the bottom center of the window we have a slider, this slider changes position to show where you are in the range of interpolated images. It can also be used to set which of the interpolations is displayed, based on the position. The number of points on the slider is changed dynamically as the range is changed.



The magnifying glass button is generated by the geocoding extension mentioned in the [OpenLayers chapter](#). Clicking it brings up a search field where you can search for your address or location. The search results are limited to 5, and limited to locations in Norway, to increase accuracy.



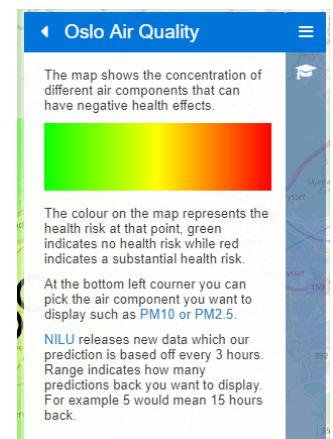
The pin button is a button used to turn on locations services. Using location services we are able to zoom in on where the user is, and mark it.



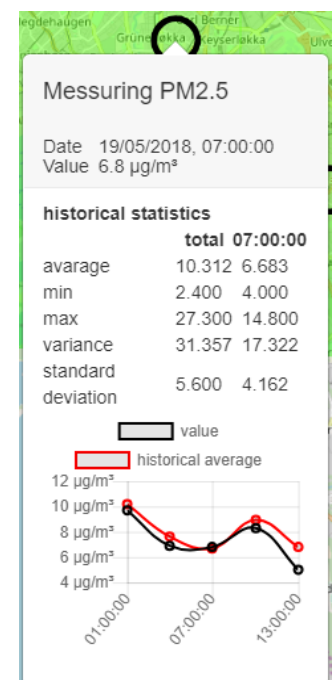
The loading text appears on the map when the user changes range, while the website downloads all the interpolated images from the server.



The sidebar is where we inform the user of the sites functionality and state our sources. Clicking on the student hat brings up another tab in which we go into more detail about NILU and explain that this is a bachelor project in cooperation with IBM.



When we click on a station the following popup appears. It shows Current and historical information about the measurements at that station. The graph on the bottom of the window displays historical data, middle of the graph will always display the value for the selected measurement. The graph extends six hours forward and backwards in time from that point, using three hour increments. Clicking on another station will close the first popup and create a new one at the station selected. Clicking another place on the map will close the popup.



5.4 Program setup

During our development we used Linux based operating systems. From our experience it was a substantially better experience over working in Microsoft Windows. If this project is to be expanded upon or recreated we would recommend using a Linux OS. The following instructions will be for a Ubuntu Linux OS.

Running the project locally

The first step to setting up our project is to clone or download the project source files from the github repository⁶⁴. The requirements for running our project include Python 2.7⁶⁵, PyPa pip⁶⁶, and the requirements listed in the “requirements.txt” file.

To install Python 2.7 and pip run the following commands in the command prompt.

```
$ sudo apt update
$ sudo apt dist-upgrade
$ sudo apt install python2.7 python-pip
```

After the installations are complete you can verify that you have the correct version of Python by running the following line.

```
$ python -V
```

The output should be

```
python 2.7.14
```

To install the rest of the requirements navigate the terminal into our project folder and run the following command.

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

If you successfully install the required packages, you can simply start the server with the following command from inside our project folder:

```
$ python main.py
```

⁶⁴ “Project github repository”<https://github.com/skjalg-gustav/LuftKvalitet> Accessed 18 May. 2018.

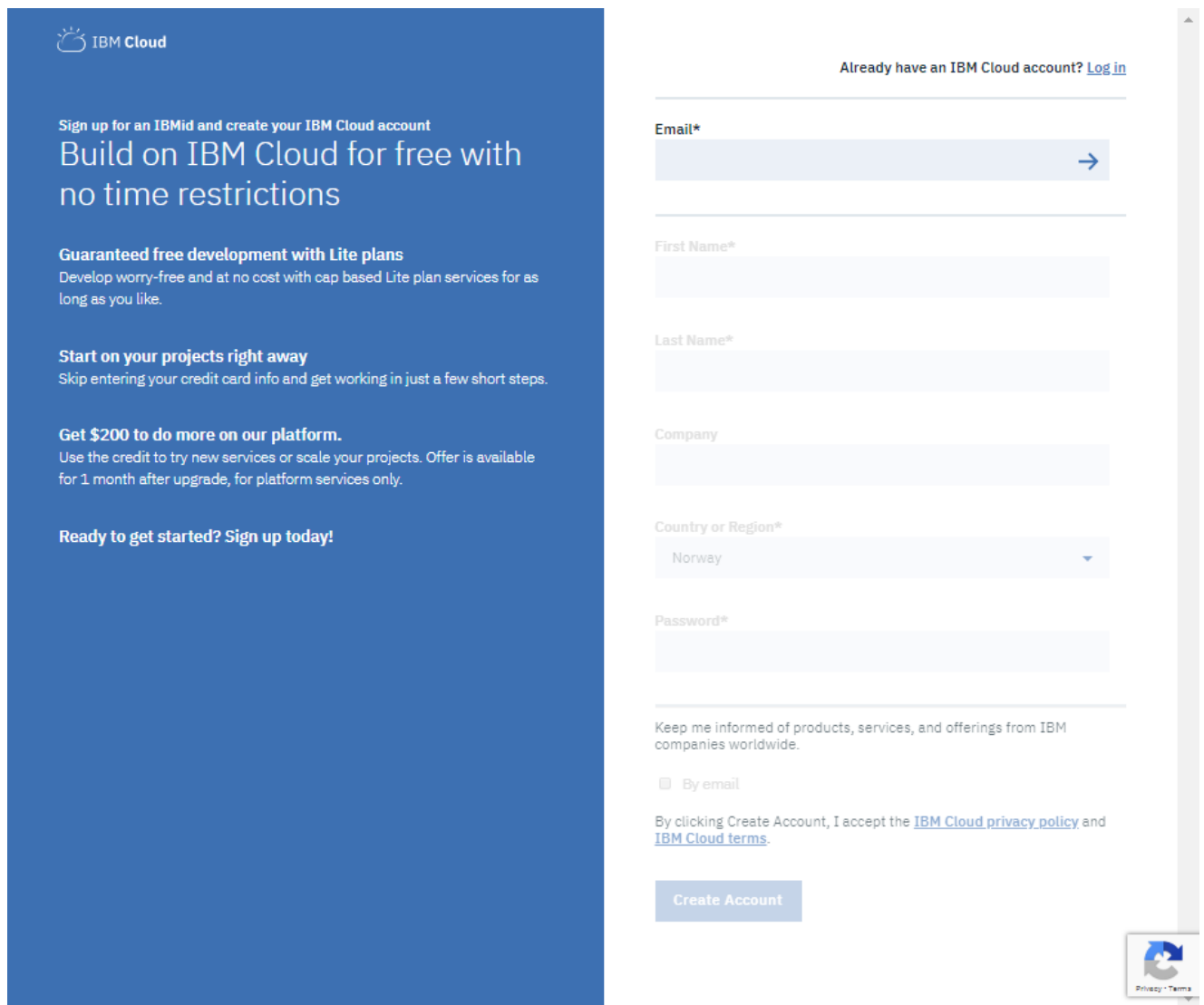
⁶⁵ “Python 2.7.0 Release | Python.org.” <https://www.python.org/download/releases/2.7/>. Accessed 19 May. 2018.

⁶⁶ “pip · PyPI.” <https://pypi.org/project/pip/>. Accessed 19 May. 2018.

If you want the web app to connect to a database you will need to first complete instructions for setting up IBM Cloud Cloudant NoSQL and making a vcap-local.json file described in the backend chapter about main.py.

IBM Cloud

The first step when working with IBM Cloud is making an account. You can sign up at the following link: <https://console.bluemix.net/registration/>⁶⁷.



IBM Cloud

Sign up for an IBMid and create your IBM Cloud account

Build on IBM Cloud for free with no time restrictions

Guaranteed free development with Lite plans
Develop worry-free and at no cost with cap based Lite plan services for as long as you like.

Start on your projects right away
Skip entering your credit card info and get working in just a few short steps.

Get \$200 to do more on our platform.
Use the credit to try new services or scale your projects. Offer is available for 1 month after upgrade, for platform services only.

Ready to get started? Sign up today!

Already have an IBM Cloud account? [Log in](#)

Email*

First Name*

Last Name*

Company

Country or Region*

Password*

Keep me informed of products, services, and offerings from IBM companies worldwide.

☐ By email

By clicking Create Account, I accept the [IBM Cloud privacy policy](#) and [IBM Cloud terms](#).

Create Account

[Privacy - Terms](#)

Screenshot: IBM Cloud Sign up screen

⁶⁷ IBM Cloud was earlier named IBM Bluemix so many of the links and commands still use the Bluemix name.

IBM Cloud CLI

The IBM® API Connect Command Line Interface (CLI)⁶⁸ is used to manage your IBM account with all its resources and apps.

Installation on linux is done by running the following command⁶⁹:

```
$ curl -sL https://ibm.biz/idt-installer | bash
```

Uploading a project to the cloud is done by the following command:

```
$ bx cf push projectname
```

Other basic commands for getting a list of apps, stopping and starting your project are:

```
$ bx cf apps
```

```
$ bx cf start projectname
```

```
$ bx cf stop projectname
```

IBM Cloud related files

IBM Cloud has a couple of setup files that are used to define some basic settings for our application when we are uploading it to the cloud. These files are “manifest.yml”⁷⁰, “Procfile”⁷¹ and “requirements.txt”⁷².

applications:

```
- name: luftkvalitet
  disk_quota: 1G
  instances: 1
  memory: 2G
  buildpack: python_buildpack
  routes:
  - route: luftkvalitet.eu-gb.mybluemix.net
  stack: cflinuxfs2
```

⁶⁸ "The Command Line Interface - IBM."

https://www.ibm.com/support/knowledgecenter/en/SSMNED_5.0.0/com.ibm.apic.overview.doc/ref_cli_about_cli.html. Accessed 11 May. 2018.

⁶⁹ "IBM Cloud Developer Tools" <https://console.bluemix.net/docs/cli/index.html#installation> Accessed 11 May. 2018.

⁷⁰ "The manifest.yml file" https://console.bluemix.net/docs/services/Cloudant/tutorials/create_bmxapp_appenv.html#the-manifest-yml-file. Accessed 17 May. 2018.

⁷¹ "The 'Procfile' file" https://console.bluemix.net/docs/services/Cloudant/tutorials/create_bmxapp_appenv.html#the-procfile-file. Accessed 17 May. 2018.

⁷² "The requirements.txt file" https://console.bluemix.net/docs/services/Cloudant/tutorials/create_bmxapp_appenv.html#the-requirements-txt-file. Accessed 17 May. 2018.

File: manifest.yml from our project

This is the luftkvalitet_manifest file that specifies to IBM Cloud what we are uploading.

web: python main.py

File: Procfile from our project

The Procfile is a file telling IBM Cloud what kind of application we have and what file to use to start our app.

```
Flask==0.12.2
cf-deployment-tracker==1.0.4
cloudfant==2.4.0
urllib3==1.22
numpy==1.14.0
pandas==0.22.0
matplotlib==1.5.1
scipy==1.0.0
Utm==0.4.2
APScheduler==2.1.2
simplejson==3.13.2
```

File: requirements.txt from our project

The requirements.txt contains all the other Python packages we need for our project to run.

5.5 Running on IBM Cloud

Once you have finished testing the project locally the next step is running it on IBM Cloud. To get the app running on IBM Cloud we will use the CLI. Open the linux terminal in the project folder and run the following commands:

```
$ bx api api.eu-gb.bluemix.net
$ bx login
```

Then fill in your login credentials, then:

```
$ bx target -o 'your organisation73' -s dev
```

If you only have access to a IBM Lite Cloud account use the “manifest-LITE-USER.yml”, delete “manifest.yml” and replace it with the Lite user equivalent. Before the next step make sure to change the start of “- route” from “luftkvalitet.eu-gb.mybluemix.net” to something unique like “my-copy-of-luftkvalitet.eu-gb.mybluemix.net” inside the “manifest.yml” file.

Then finally run:

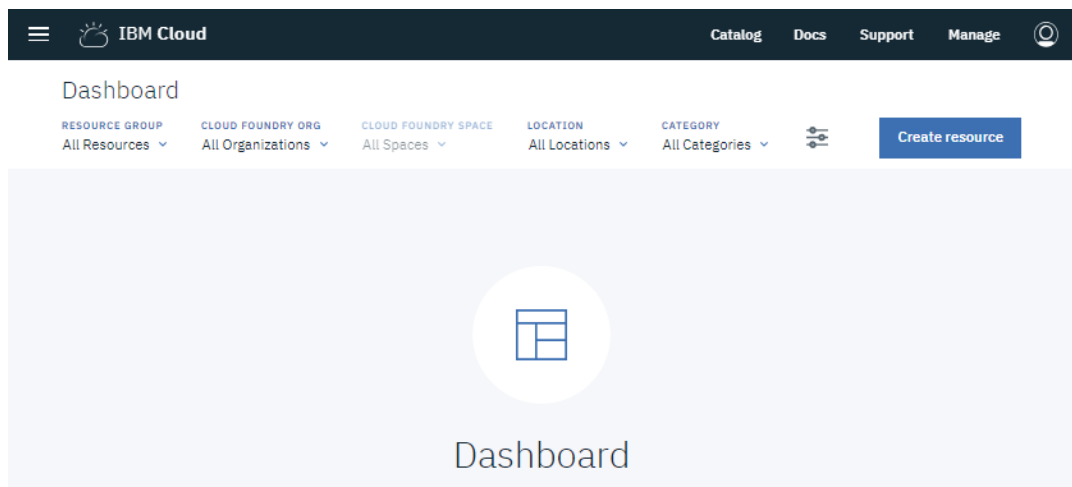
⁷³ By default the organisation is set to the email used for signing up.

```
$ bx cf push
```

After a few minutes your copy of the project will be on IBM Cloud but it will crash until you create and connect a database as explained in the next chapter.

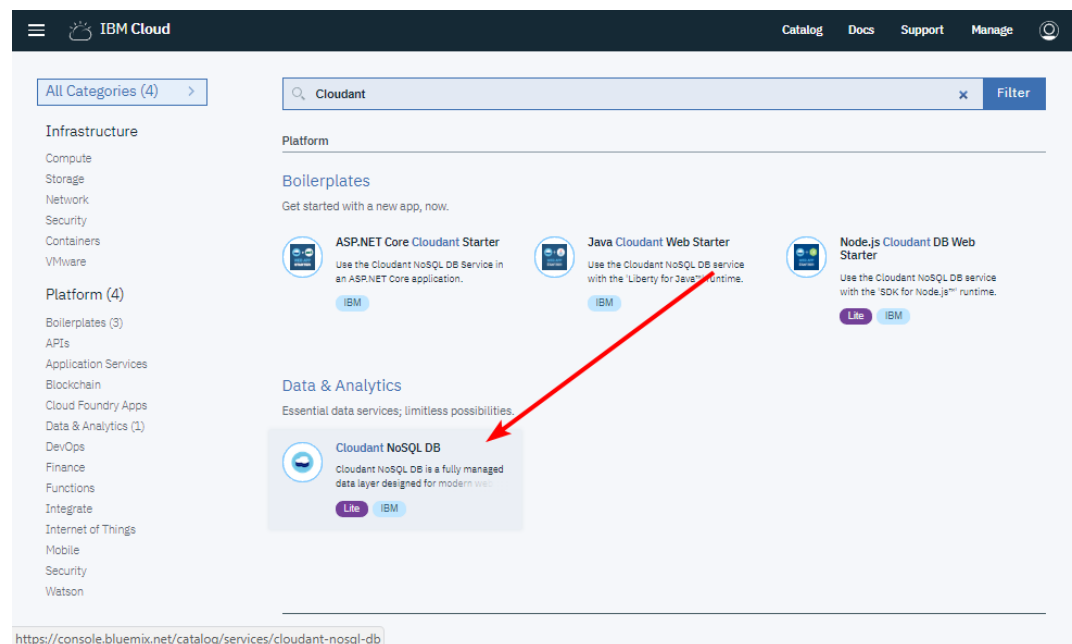
5.6 Cloudant noSQL database

To save the interpolations for our project we need a database. In our project we use a Cloudant NoSQL DB⁷⁴, which we create manually using the IBM Cloud Dashboard. Go to the IBM Cloud Dashboard by going to the following link <https://console.bluemix.net/dashboard/apps>.



Screenshot: IBM Cloud Dashboard home-screen

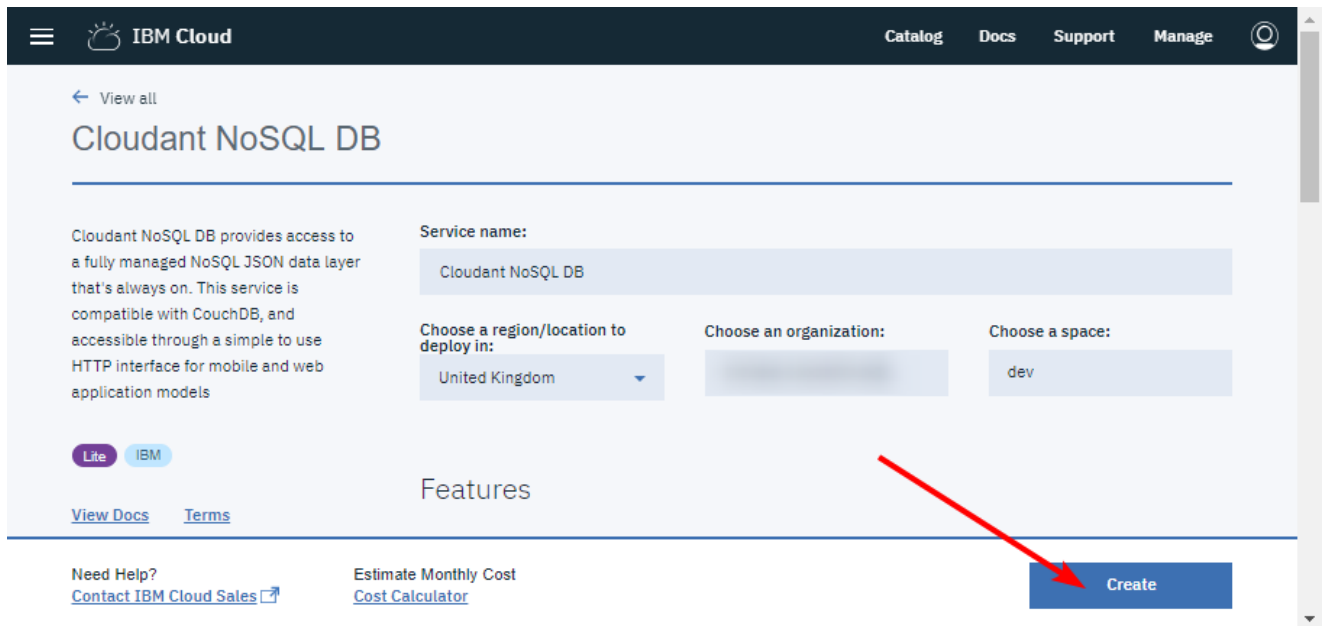
When on you have opened the dashboard press the “Create resource” button and then search for “Cloudant”.



⁷⁴ https://console.bluemix.net/docs/runtimes/python/getting-started.html#add_database

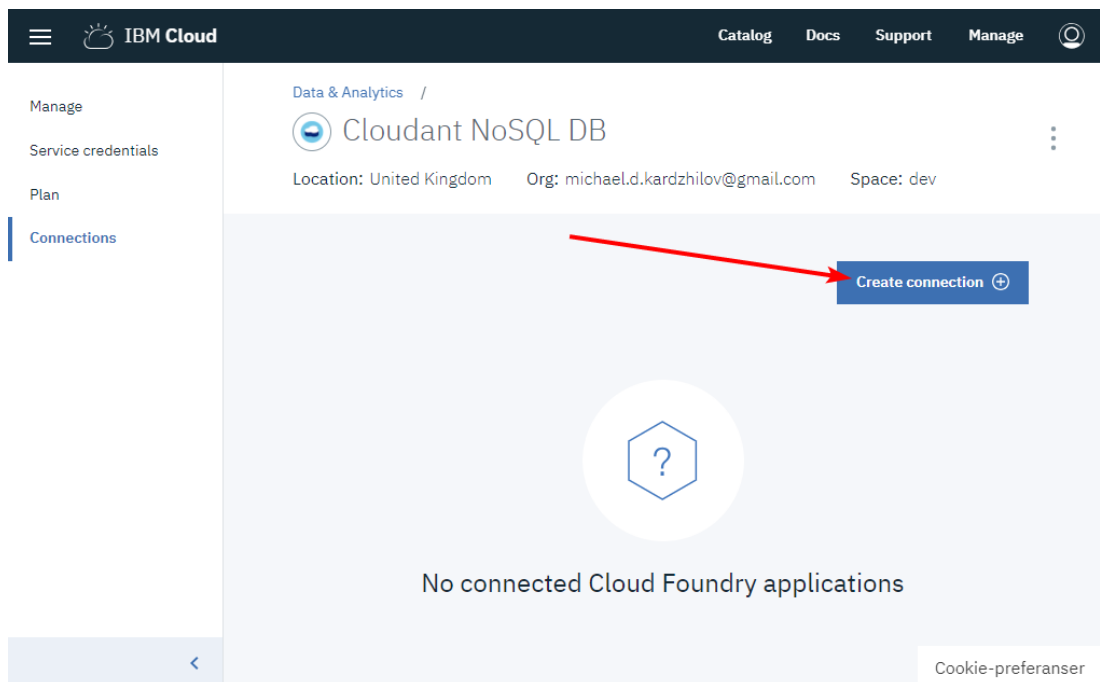
Screenshot: 'Cloudant' search in IBM Cloud

Select the “Cloudant NoSQL DB” option from the “Data & Analytics” section and create the service.



Screenshot: Cloudant NoSQL DB Creation screen⁷⁵

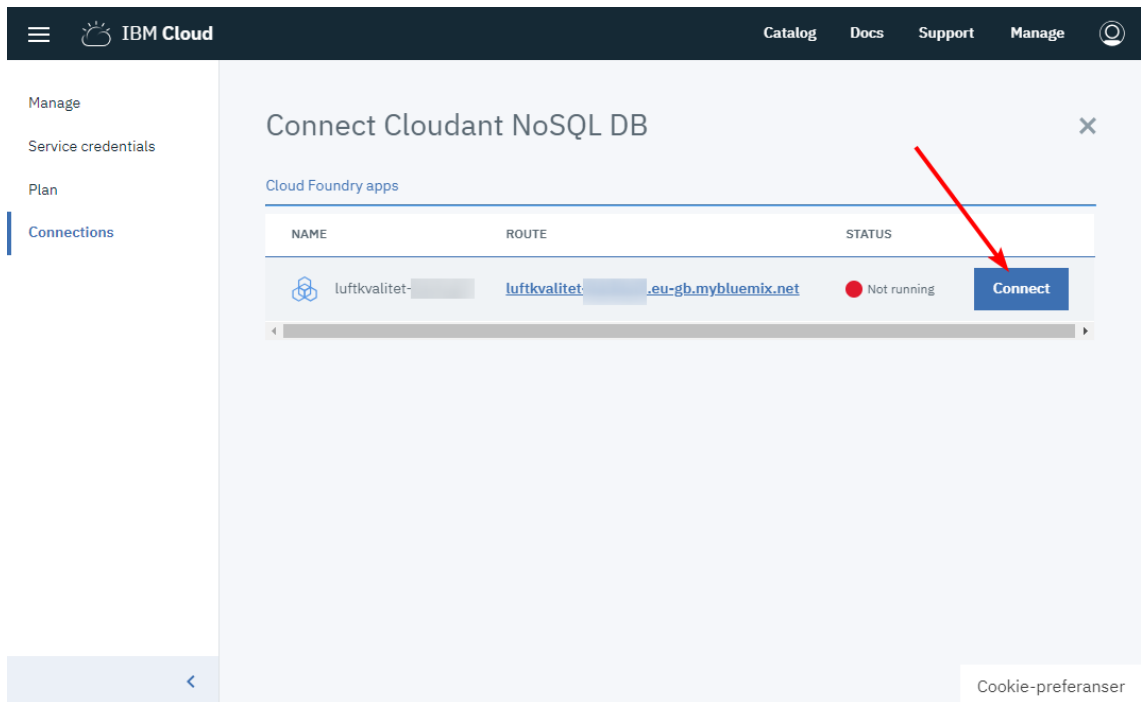
Once created we need to connect our database to our project. We do this by clicking on the cloudant database in our dashboard, going to the connections tab, and pressing “Create connection”.



Screenshot: Cloudant NoSQL DB Connection screen

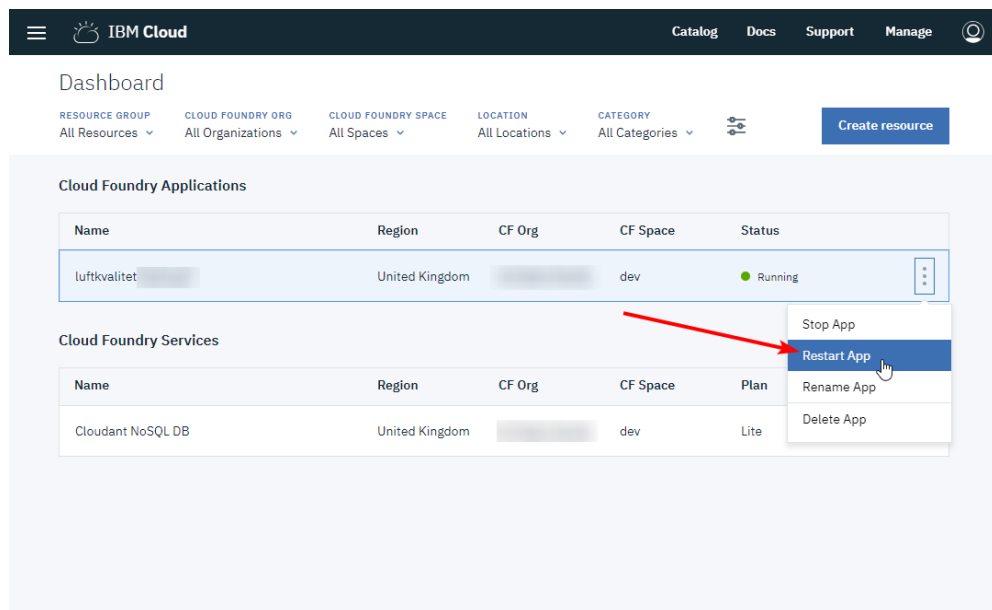
Once on the connection selection screen select the project you uploaded earlier and then press “Restage” when prompted.

⁷⁵ The IBM Cloud organisation is usually represented by the e-mail address used to sign up.



Screenshot: Cloudant NoSQL DB Connection selection screen

Once the connection has been established you need to restart the project app from the IBM Cloud Dashboard.



Screenshot: IBM Cloud Dashboard

Once this step is completed⁷⁶ you should have a working copy of our project which will generate new interpolations every 3 hours. You will need to wait 3 hours before the site is operational as it does not work while the database is empty.

⁷⁶ If it does not work follow the last step from 5.5 again.

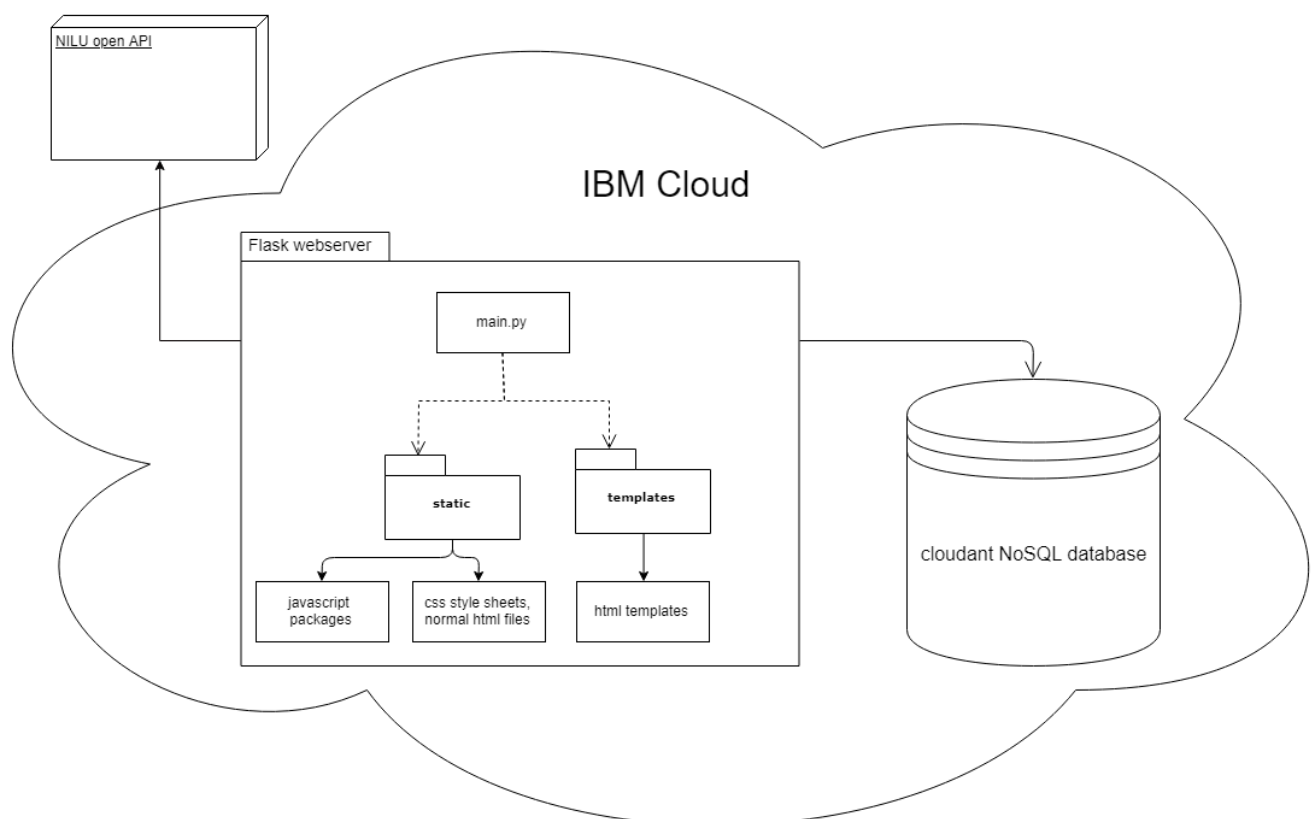
5.7 Program components

We have split our technical program documentation into two different sections, backend and frontend. Backend is where we document all the components on the server side and frontend documents everything happening on the web app that you can see as the interface.

Backend

The backend consists of a flask server and the files it uses to serve the web app and all its functions.

Python Flask



Illustration, flask structure⁷⁷

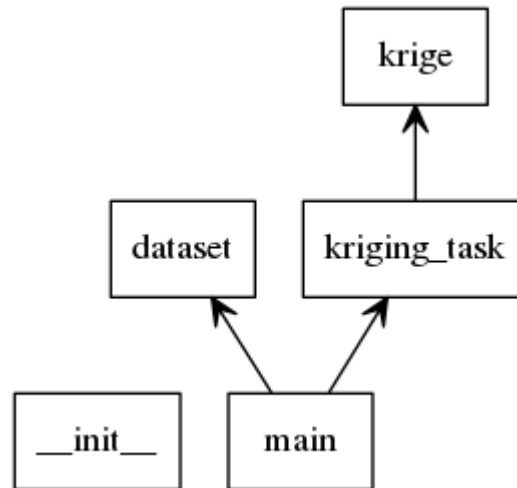
The Flask⁷⁸ server runs from the main.py file. The folders static and templates is part of the flask framework and has different uses. In static, flask expects to find static files it can serve the user. Here we store for example stylesheet or javascript packages. Templates are used to store HTML templates, we will expand on templates later.

⁷⁷Illustration made using <https://www.draw.io/>

⁷⁸"Welcome | Flask (A Python Microframework)." <http://flask.pocoo.org/>. Accessed 11 May. 2018.

The two arrows pointing out of the Flask Server point to the open NILU API and our Cloudant NoSQL database. They will be described in detail in the next segment.

Python files



Illustration, packages dependency, generated using pyreverse⁷⁹

We have programmed in a functional style and as such have no classes. We have split our code into a number of files each with its purpose. The diagram above describes file dependencies⁸⁰.

dataset.py

The purpose of dataset is to gather data from NILU's open API, it has only one function, data.

“data” extracts data from the NILU API and formats it into a dataframe. Each measurement that it extracts will be given an x, y value using the Python package UTM. Lastly we will remove the offset, using the bottom left corner of the area we map.

```
# gps coordinates for map
# TOP LEFT      59.963886, 10.662291
# TOP RIGHT     59.963886, 10.887139
# BOTTOM RIGHT  59.873800, 10.887139
# BOTTOM LEFT   59.873800, 10.662291

# bottom left corner in GPS Latitude and Longitude
latitude_left_corner = 59.873800
longitude_left_corner = 10.662291
```

⁷⁹ "Pyreverse : UML Diagrams for Python (Logilab.org)." 23 Dec. 2008, <https://www.logilab.org/blogentry/6883>. Accessed 14 May. 2018.

⁸⁰ The `__init__` file is a empty file, which was necessary for pyreverse to work, as it looked for a `__init__` file to consider the folder a python project.

In the code snippet above we establish the position of the bottom left corner so that we can use it later as the 0,0 position in our coordinate system.

```
def data():
    # fetches data from nilu
    data = urlopen('https://api.nilu.no/aq/utd.json?areas=Oslo').read()
    # turns it into a json object
    jdata = json.loads(data)
    # puts it in a dataframe
    fdata = json_normalize(jdata)
    # picks out the wanted values
    fdata = DataFrame(fdata, columns=['latitude', 'longitude', 'value',
    'unit', 'component', 'toTime'] )
```

The code snippet above is where we retrieve the raw data from NILU, and filter out the information we want.

```
# add x, y columns to fdata
fdata['x'] = 0
fdata['y'] = 0
```

In order to place the stations into a local coordinate system we establish x and y columns in our dataframe.

```
for i, row in fdata.iterrows():
    # add calculated x, y to dataframe and remove the offset (bottom left
    corner) to make a local coordinate x,y system
    fdata.loc[i, 'x'] = utm.from_latlon( row['latitude'],
    row['longitude'] )[0] - utm.from_latlon(latitude_left_corner,
    longitude_left_corner)[0];
    fdata.loc[i, 'y'] = utm.from_latlon( row['latitude'],
    row['longitude'] )[1] - utm.from_latlon(latitude_left_corner,
    longitude_left_corner)[1];

return(fdata)
```

In the last segment we add the x, and y value using the UTM package. Using the bottom left corner we defined at the top we remove the offset from that point. Then return the dataframe with all our data.

krige.py

krige.py contains the mathematical functions used for krige_task.py. Inside it we implement all of Connor's kriging⁸¹ related functions and credit him at the start using his MIT licence⁸².

krige_task.py

Krige_task is the file containing the routine for making an interpolated picture, it is used by main.

```
from krige import SVh, SV, C, spherical, opt, cvmodel, krige
```

Code snippet from krige_task.py, imports from krige.py Line 24

We use a number of imports, from our krige.py file. These are all the functions we need to interpolate pollution with kriging.

```
# gps coordinates for map
# TOP LEFT      59.963886, 10.662291
# TOP RIGHT     59.963886, 10.887139
# BOTTOM RIGHT  59.873800, 10.887139
# BOTTOM LEFT   59.873800, 10.662291

# map corners in utm
LEFT      = 593065.1648494017; # X0 for the map in utm
BOTTOM    = 6638524.509011956; # Y0 for the map in utm
RIGHT     = 605365.439142052;  # X1 for the map in utm
TOP       = 6648891.652304975; # Y1 for the map in utm
```

Code snippet from krige_task.py line 35-45

We set a couple of important variables related to what area we map, they are used to delete the offset so we can start from a 0, 0 coordinate when interpolating.

```
def krige_task(data):
    # dataframe containg all the measurements from dataset
    z = data
```

Code snippet from krige_task.py line 47-49

To increase the accuracy of our model we add extra points to the dataframe to act as fictive stations. These points are based on the lowest value in our dataframe. We do this because all the NILU stations are in the city of Oslo often close to roads. This makes it often look like the outskirts of town and the ocean are highly polluted when we are fairly certain they are not.

```
#costume points to improve our analysis
```

⁸¹ "Simple Kriging in Python | Connor Johnson." 20 Mar. 2014, <http://connor-johnson.com/2014/03/20/simple-kriging-in-python/>. Accessed 20 May. 2018.

⁸² "The MIT License | Open Source Initiative." <https://opensource.org/licenses/MIT>. Accessed 20 May. 2018.

```

num_points = len(z.index)
unit_type = z.loc[1, 'unit']
comp_type = z.loc[1, 'component']

lowest_point = 900000 # just some large number so that we can find the
lowest in the table
#lowest value
for i, row in z.iterrows():
    if lowest_point > z.loc[i, 'value']:
        lowest_point = z.loc[i, 'value']

# make the costume points to be 40% of the lowest point
new_low = lowest_point*0.4

```

Code snippet from kriging_taskline line 48-60

In order to create extra points we start off by retrieving the component, unit and number of points in the dataframe. We then loop through all of the values for the components in the dataframe to find the lowest one. Finally we set the value we will use for our points to be 40% of the lowest value.

```

# all gps points of costume points
# 59.878971, 10.678320
# 59.890065, 10.717549
# 59.880442, 10.746924
# 59.874472, 10.871463
# 59.957640, 10.820121
# 59.902286, 10.871526

# array of the points
lant_arr = [59.878971, 59.890065, 59.880442, 59.874472, 59.957640,
59.902286]
lat_arg = [10.678320, 10.717549, 10.746924, 10.871463, 10.820121,
10.871526]

t = 0; # Loop index
for i in lant_arr:
    x = utm.from_latlon( lant_arr[t], lat_arg[t] )[0] -
utm.from_latlon(lat1, long1)[0];
    y = utm.from_latlon( lant_arr[t], lat_arg[t] )[1] -
utm.from_latlon(lat1, long1)[1];
    new_df = [lant_arr[t],lat_arg[t], new_low, unit_type, comp_type, x,
y]
    t = t+1
    z.loc[len(z)] = new_df

```

The code snippet above adds new entries to the database using the latitude array, longitude array and the values from the last code snippet. The way we picked the locations for these extra points was by looking on our mapped area for locations which we expected to have less pollution, like for example at sea or in forests.

Once our dataframe is ready it is time to interpolate it.

```
# part of our data set recording pollution
P = np.array( z[['x','y','value']] )
# bandwidth
bw = 15500
# kriging distances
hs = np.arange(0,20000, bw)

X0, X1 = 0, RIGHT-LEFT
Y0, Y1 = 0, TOP-BOTTOM

# resolution of the interpolated picture, indexes for the numpy array
nx = 48
ny = 60
```

The first step to interpolating the values is creating a numpy array “**P**” used to store the results. We then set the bandwidth, Connor did not document the **bandwidth** variable well, but from testing out different values we can tell a few things. Setting it to a high value, up to the max kriging distances -1 smoothes out the interpolated result and increases the kriging calculation time. Setting a low value makes the result blocky and speeds up the kriging calculation time. Kriging **distances** establishes how far away points should influence each other. The **X0, X1** and **Y0, Y1** set the bounds of our local coordinate system. **nx** and **ny** are the X and Y resolution of the rendered image, it is important that their aspect ratio correlates to the aspect ratio of the area you want to map.

```
# consider all the points available, but no more than 10.
num_points = len(z.index)
if (num_points > 10):
    num_points = 10;
```

The variable num_points is the variable we will use to decide how many points that will be taken into consideration for each point we interpolate. To limit the time and resources needed to complete the calculation we make sure the num_points is 10 or less.

```
# make an numpy array
Z = np.zeros((ny,nx))

# divide the range of the map width, height with our resolution
dx, dy = (X1-X0)/float(nx), (Y1-Y0)/float(ny)
```

Then we make an numpy array the size of nx, ny. The variables dx, dy will split our coordinate system into dx by dy boxes.

```
# calculate each point with kriging
for i in range(nx):
    print (i), # print i for showing progress, goes till it reaches nx
    for j in range(ny):
        # get the x, y address of the cell for the numpy array
        x = X0 + i * dx
        y = Y0 + j * dy
        # preform kriging for the cell
        Z[ j, i ] = krige( P, spherical, hs, bw, (x, y), num_points )
```

Using for-loops we go through our numpy array and perform kriging for each point. Printing out the i variable gives us an idea how far our progress is, when i reaches nx the kriging is complete. Lastly we call the krige function which gives us an interpolated value for our given point.

```
# round the calculations, to increase the contrast
H = np.zeros_like( Z )
for i in range( Z.shape[0] ):
    for j in range( Z.shape[1] ):
        H[i,j] = np.round( Z[i,j] )
```

```
return H;
```

This segment loops through the numpy array containing the interpolated values and rounds them.

Conner had this in his tutorial and after testing it out we found that it increases contrast of the image as the steps between each point next to each become bigger as the rounded values don't have all the decimals. This also helps keep our file size down as it removes unnecessary information about each point.

main.py

main.py is where we define our flask server.


```
app = Flask(__name__)
```

We define the variable app as our Flask server. This makes it so that we can use the flask framework through this variable.

```
# On Bluemix, get the port number from the environment variable PORT  
# When running this app on the local machine, default the port to 8000  
port = int(os.getenv('PORT', 8000))
```

The port variable is used to set the port to 8000 when the web app is not started in IBM Cloud. This means you would have to go to localhost:8000 or ip-address:8000 if you're running it on a separate host.

```
if __name__ == '__main__':  
    app.run(host='0.0.0.0', port=port, debug=True)
```

This line is at the bottom of the main.py, it starts the flask server with the port.

```
# Scheduler for krige_task  
cron = Scheduler(daemon=True)  
cron.start()
```

Code snippet from main.py, get_img

The cron variable is defined as a Scheduler. A Scheduler is a Python package for scheduling tasks, daemon=true essentially means it is going to construct a daemon and work standalone by itself.

```
#set the time interval to run kriging and clean_db  
@cron.interval_schedule(hours=3)  
def job_function():  
    kriging_plot()  
    clean_db()
```

Code snippet from main.py

Here we use our scheduler cron to schedule the job_function() function every 3 hours, we use this to run kriging routinely.

```
# Shutdown the cron thread if the web process is stopped (used to  
shutdown thread cycling the krige_task)  
atexit.register(lambda: cron.shutdown(wait=False))
```

Code snippet form main.py

Atexit is a Python package that schedules things to happen upon exit. Meaning when the server shuts down its gonna run the registered commands. We set our scheduler cron to shutdown when our server goes down.

```
# database variables
db_name = 'luftkvalitet_db'
client = None
db = None

# test if you're on IBM Cloud or running locally
if 'VCAP_SERVICES' in os.environ:
    # get environment variables for database from your IBM Cloud account
    vcap = json.loads(os.getenv('VCAP_SERVICES'))
    print('Found VCAP_SERVICES')
    if 'cloudantNoSQLDB' in vcap:
        creds = vcap['cloudantNoSQLDB'][0]['credentials']
        user = creds['username']
        password = creds['password']
        url = 'https://' + creds['host']
        client = Cloudant(user, password, url=url, connect=True)
        db = client.create_database(db_name, throw_on_exists=False)
elif os.path.isfile('vcap-local.json'):
    # get environment variables for database from a local vcap file
    with open('vcap-local.json') as f:
        vcap = json.load(f)
        print('Found local VCAP_SERVICES')
        creds = vcap['services']['cloudantNoSQLDB'][0]['credentials']
        user = creds['username']
        password = creds['password']
        url = 'https://' + creds['host']
        client = Cloudant(user, password, url=url, connect=True)
        db = client.create_database(db_name, throw_on_exists=False)
```

Code snippet from main.py

Creating the database variables used to communicate with our NoSQL Database we first define our variables. We will test if we are inside IBM Cloud or not, if you are inside IBM Cloud you set the database variables to the environment variables inside your Cloud instance. Otherwise it will look for a local vcap-local.json file.

```
{
  "services": {
```

```

"cloudantNoSQLDB": [
  {
    "credentials": {
      "username": "CLOUDANT_DATABASE_USERNAME",
      "password": "CLOUDANT_DATABASE_PASSWORD",
      "host": "CLOUDANT_DATABASE_HOST"
    },
    "label": "cloudantNoSQLDB"
  }
]
}
}

```

vcap-local.json file example <https://github.com/IBM-Cloud/get-started-python/>

The vcap-local.json file can be made using the example above, just by replacing the placeholder text for username, password and host. As explained in the github tutorial these variables can be found with the following steps:

Back in the IBM Cloud UI, select your App -> Connections -> Cloudant -> View Credentials

#Green, yellow, orange, red; color map

```

cdict = {'red': ((0.0, 0.0, 0.0),
                (0.35, 1.0, 1.0),
                (0.4, 1.0, 1.0),
                (0.8, 1.0, 1.0),
                (1.0, 2.0, 0.0)),

         'green': ((0.0, 1.0, 1.0),
                  (0.5, 1.0, 1.0),
                  (1.0, 0.0, 0.0)),

         'blue': ((0.0, 0.0, 0.0),
                 (0.1, 0.0, 0.0),
                 (0.4, 0.0, 0.0),
                 (1.0, 0.0, 0.0))

        }

my_cmap = matplotlib.colors.LinearSegmentedColormap('my_colormap', cdict,
256)

```

Code snippet from main.py

We also define our colormap variable that's going to be used in some of the functions in main.py.

```

@app.route('/img/<_id>')
def get_img(_id):
    #connect to the db
    client = Cloudant(user, password, url=url, connect=True)
    db = client.create_database(db_name, throw_on_exists=False)
    #get the document from the db
    doc = db[ _id ]

```

Code snippet from main.py, get_img

This is the top of our get_img function. Over the function declaration it says

‘@app.route('/img/<_id>')’ here, app is our Flask server. When using app.route we are giving a url or a ‘route’ linked to the function. The string parameter is the url path it would need behind our server ip to call this function. In this path the <> indicate a variable, here we use it to pass _id from the url to the get_img function. In these functions anything returned with the return statement will be the server response.

Our get_img takes a _id parameter which should correspond to a id we have in our database. It will use the id to fetch the data associated with it.

```

#load the krige data
H = simplejson.loads(doc['krige_data'])
buffr = io.BytesIO()

```

Code snippet from main.py, get_img

We use simplejson to extract the data from the json document we fetched from our server and define a variable “buffr” that is a byte object we will use to write image data to. Using a byte object lets us create files in memory, without having to save them on the disk before serving them to the user.

```

#min max values for the colour
maxvalue = 100;
minvalue = 0;

#set the min max values according to whats unhealthy
#max will be unhealthy value, and in turn be red on the generated image
#we set values based on https://uk-air.defra.gov.uk/air-
pollution/daqi?view=more-info&pollutant=no2 , Accessed 11.05.2018
if (doc['component'] == 'PM2.5'):
    maxvalue = 60;
    minvalue = 0;
if (doc['component'] == 'PM10'):

```

```

    maxvalue = 85;
    minvalue = 0;
    if(doc['component'] == 'NO2'):
        maxvalue = 420;
        minvalue = 0;

```

Code snippet from main.py, get_img

The vmin and vmax variable is used to control the relationship between the colour in our interpolated image and the actual value at a given point. As of now the only components we have enough data on to make an interpolated image is PM2.5, PM10 and NO2. Having looked into what constitutes a health risk we have set a min max value for the vmin and vmax.

```

#plot the figure
    fig, ax = subplots()
    ax.imshow(H, cmap=my_cmap, vmin=minvalue, vmax=maxvalue,
origin='lower', interpolation='gaussian', alpha=0.7, extent=[X0, X1, Y0,
Y1])
    ax.axis('off')

#set proper image size and extent to plot
    fig.set_size_inches(5.95, 5)
    imgextent =
ax.get_window_extent().transformed(fig.dpi_scale_trans.inverted())
    fig.savefig(buffr, format='svg', bbox_inches=imgextent,
transparent=True, pad_inches=0, frameon=None)

```

Code snippet from main.py, get_img

After setting the correct vmin and vmax we start plotting the interpolated picture with pylab and matplotlib. Since this is the image we are going to overlay over the map we turn off the axis's so they will not be in the result. We set the fig.set_size_inches to a number that has the aspect ratio 1.19:1 which is the aspect ratio of the map area we cover. Further we set the image extent to be the the size of the interpolated image, so that we do not have borders with empty space around it. We then we call fig.savefig to save the image of the interpolated pollution, we save it inside the byte object buffr from the ByteIO Python pack. We use the svg format which is one of the most resource efficient.

```

#go to start of file
    buffr.seek(0)
#disconnect from db
    client.disconnect()

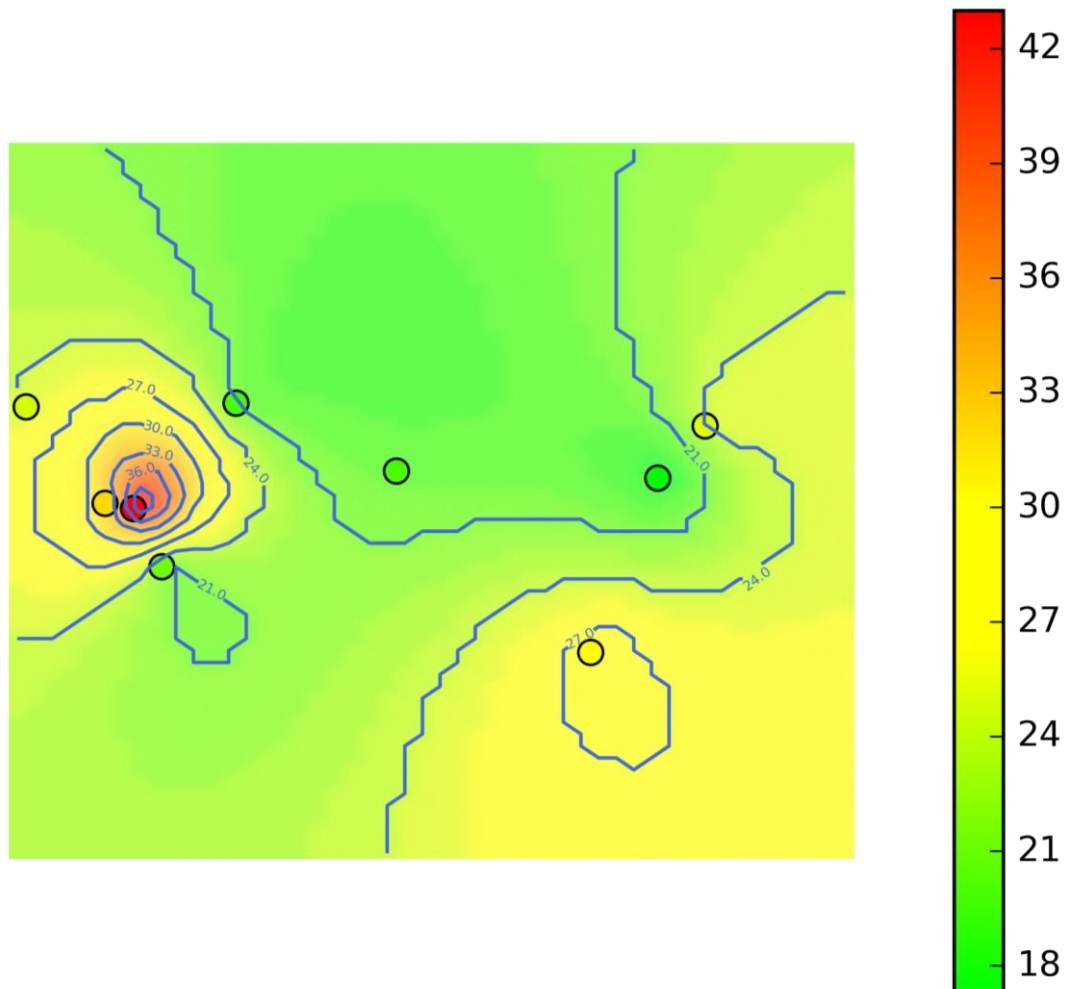
```

```
return send_file(buffr, mimetype='image/svg+xml')
```

Code snippet from main.py, get_img

We make sure the byte object “buffr” is pointing at the start of the bytes with seek(0). We disconnect from the database so we do not keep the connection alive. Then we send a server response with the “buffr” byte object as an image/svg+xml file type. We have two other functions which are very similar to this one. They each generate another type of image with the same data.

One generates a Contour line for the interpolated pollution. When you have a colour hue, like our interpolated image it can be hard to see where exactly the transition between different values occurs. A contour line is a line or a curve outlining a constant value across the map. That means following the line you will have a constant value. Here is an example of the interpolated image where we have added contour lines.



The other function generates an interpolated picture much like the first function generates a interpolated picture, the difference being that it keeps the axis's and has extra information printed onto it. This information includes the colorbar like shown in the picture above, measurement points, a

title with date and component. These images are meant for testing purposes so they are not included in the main web app, but are still accessible at /all_entries on our server.

```
@app.route('/all_entries')
def all_entries():
    #connect to the db
    client = Cloudant(user, password, url=url, connect=True)
    db = client.create_database(db_name, throw_on_exists=False)
```

Code snippet from all_entries function in main.py

The first thing we do in all_entries is make a connection to the database.

```
    #get all docs
    docs = list(map(lambda doc: doc, db) )
    #put them into a dataframe
    fdocs = json_normalize(docs);
    fdocs = DataFrame(fdocs, columns=['date', 'component', 'data', '_id'])
    fdocs['date'] = to_datetime(fdocs['date'])
    fdocs = fdocs.reset_index(drop=True)
    fdocs.sort_values(['date', 'component'])
    #get the components
    components = fdocs['component'].unique().tolist();
```

Code snippet from all_entries function in main.py

Here we gather in all the JSON documents that's stored in our database and put them into a dataframe with Dataframes json_normalize. In the next line we specify which columns we want.

The next function we use is to_datetime, this function is also from DataFrame, it will turn a JSON time to a Python datetime object. We do this to make the date readable when we print it out for the user. We reset the index for the dataframe to make sure it continuously goes from 0 and up.

We set the components variable to be a list containing 1 of the components in our dataframe.

```
data = [None] * len(fdocs)
for i, row in fdocs.iterrows():
    tmp = read_json(fdocs.loc[i, 'data'], orient='index')
    tmp = tmp.reset_index()
    data[i] = tmp
    fdocs.loc[i, 'data'] = i;
```

Code snippet from all_entries function in main.py

In this segment we extract a dataframe out of the data column we got from the database. It contains the NILU pollution data. We put each of the Dataframes containing one measurement into the data variable.

```
#make a list of same size as components
complist = [None]* len(components)
for i in range(len(components)):
    #drop everything but relevant info
    tmp = fdocs.drop(fdocs[fdocs.component != components[i]].index)
    #drop duplicates
    tmp = tmp.drop_duplicates(subset=['date'], keep='first',
inplace=False);
    #sort them
    tmp = tmp.sort_values(['date'], ascending=False)
    #re index the dataframe
    tmp = tmp.reset_index(drop=True)
    #put the dataframe into the list
    complist[i] = tmp;
#disconnect from db
client.disconnect()
return render_template('entries.html', entries = complist, data =
data);
```

Code snippet from all_entries function in main.py

Then we make a variable called complist to store our data on each component separately. After splitting and sorting our data we send a server response containing the complist and data as variables in the template entries.html.

The last important function with a route in main.py is show_data, it is the function that is used to serve the web app. The function is nearly identical to all_entries but the main difference is that it does not make the rows in the date column into Python datetime objects.

Templates

We use Templates for our web app. Using templates the server will generate parts of the HTML document when sending it to the user, The flask templates uses the Jinja2⁸³. We define what template we want and what variables we send it in the server response.

```
return render_template('index.html', entries = complist, data = data);
```

⁸³ "Jinja2 Documentation." <http://jinja.pocoo.org/docs/2.10/>. Accessed 11 May. 2018.

Code snippet from our main.py

When the user requests the website from the web server it runs the flask command 'render_template'. In the code snippet above we specify that we want to use the template 'index.html', and we declare a number of variables to be used in the template. These variables will be available inside the template before they are sent to the user.

```
{% for df in entries: %}
    dropdown.appendChild(new Option("{{df.loc[0, 'component']}}",
    "{{loop.index0}}"));
{% endfor %}
```

Code snippet from our index.html

We use normal Python syntax inside the HTML document encapsulated by {% ... %} and {{ ... }}. These two has two different uses, {% ... %} is used for logic that does not directly write something in the document. {{ ... }} is used when writing something to the document. In the code snippet above we generate JavaScript code for adding a child element to our dropdown menu. Each child element we generate will have the value of the first row in the component column. We also make use of a special Jinja2 template variable, loop.index0. It can be used inside a loop block and will give us the current iteration we are on. It comes in two forms loop.index and loop.index0, with the difference being loop.index0 starts at 0.

We use templates to generate more elements like this with similar logic.

```
<script type="text/javascript" src="{{ url_for('static',
filename='jquery.js') }}"></script>
```

Code snippet from our index.html

Inside the templates we use "url_for" to generate a url for our static files like shown in the snippet. Url_for can also be used to call functions such as get_img.

Frontend

The frontend of our web app is built up using a number of JavaScript packages.

OpenLayers

In order to utilize OpenLayers we need to start by including their JavaScript library and stylesheet.

```
<link rel="stylesheet" href="https://openlayers.org/en/v4.6.5/css/ol.css"
type="text/css">
```

```
<script src="https://openlayers.org/en/v4.6.5/build/ol.js"></script>
```

Snippet from our index.html

The lines above is how we include OpenLayers in our project. Alternatively we could self host it, this would allow us to adjust the functionality available to better suit our needs. We decided against this because it adds a potential point of failure for our project, and we are not sure if we would be able to host it on IBM Cloud with our restrictions.

When setting up an OpenLayers map you need assign a <div> to contain the map, so we do that in the following way.

```
<div id="map" class="map sidebar-map">
  <div id="popup"></div>
</div>
```

Snippet from our index.html

The map div contains a div for the popups which we will go into detail about later.

The next step is to setup your map in JavaScript.

```
var map = new ol.Map({
  controls: ol.control.defaults().extend([
    new ol.control.FullScreen(),
    new app.DropDown(),
    new app.PositionButton(),
    new app.DateText(),
    new app.HistoricSlider(),
    new app.PreviousButton(),
    new app.NextButton(),
    new app.PlayPauseButton(),
    new app.Range(),
    new app.Contour(),
    new app.Loading()
  ]),
  interactions: ol.interaction.defaults().extend([
    new ol.interaction.DragRotateAndZoom()
  ]),
  layers: [
    new ol.layer.Tile({
      source: new ol.source.OSM()
    }),
    map_base_img,
    map_contour,
    vectorLayer
  ]
});
```

```

    ],
    target: "map",
    loadTilesWhileAnimating: true,
    view: view
  });

```

Snippet from our index.html

Our map variable is more complex than most examples you would find on the OpenLayers, this is because of the amount of layers and controls we add to it. At the top we start by extending all the our **controls** for our project. These include controls such as buttons and sliders. Under **Interactions** we enable the ability to rotate the map using touch controls or clicking and holding CTRL. We then establish the **layers** and their order. The layer on the top of the list is the one that will be furthest in the back while the layer on the bottom of the list will be on top. Our first layer is the tile layer, this is what would be considered the geographic map layer where the map displays a normal geographic map. The source we use for our tile layer is OpenStreetMap as they are open source and require no license or setup, but if we wished we could have set it up to use Bing or Google maps for example. The second layer is the “map_base_img” this is an image layer where we overlay the interpolated coloured representation of air pollution. The third layer is “map_contour” this is an image layer where we overlay the contour lines for our interpolation. The final layer is the “vectorLayer” it is a vector layer which is used for plotting the NILU measurement stations on the map. **Target** is used to assign the map to a specific <div>. **View** is used to determine the maps center, rotation and potentially limit the resolution and zoom factor.

In order to add controls we first to establish them.

```

app.PreviousButton = function(opt_options) {

  var options = opt_options || {};

  var button = document.createElement('button');
  button.innerHTML = '<';
  button.addEventListener("click", function() {
    previous();
  });
  var element = document.createElement('div');
  element.className = 'previous-button ol-unselectable ol-control';
  element.appendChild(button);

  ol.control.Control.call(this, {

```

```

        element: element,
        target: options.target
    });

    };
    ol.inherits(app.PreviousButton, ol.control.Control);

```

Snippet from our index.html

The first two and last six lines of snippet above are the syntax for creating a custom control for OpenLayers. The lines in the middle are where we establish that it is a button, and give it classes.

```

.previous-button{
    top: 1.5em;
    left: 40%;
    margin-right: -40%;
    transform: translate(-50%, -50%);
}

```

Snippet from stylesheet.css

Once the control element is established in JavaScript we need to use CSS to declare its position and other estetic settings. The snippet above shows how how we place our “<” button on the top of the screen to the left of the play pause button.

```

var map_base_img = new ol.layer.Image({
    source: new ol.source.ImageStatic({
        url: "",
        projection: "EPSG:32632",
        imageExtent: imageExtent
    }),
    opacity: 0.65
});

```

Snippet from our index.html

The snippet above is is how we define an image layer. **url** is the variable used to establish the image source for the layer, this is something we set programmatically as the page loads in. **projection** refers to the spherical reference system⁸⁴ used. In our case we use “EPSG:32632” which is a projection where Oslo is included inside of it. **imageExtent** sets the extent the image, in our case we use it to project our image over the area we established in the start of chapter 4.5. **opacity** decides the opacity

⁸⁴ "Spatial reference system - Wikipedia." https://en.wikipedia.org/wiki/Spatial_reference_system. Accessed 20 May. 2018.

of the layer. This means that even if you use an image format like JPG which does not have transparency and can set the layer to have transparency, so that you can still see the map underneath.

```
var imageExtent = [  
    593065.1648494017,  
    6638524.509011956,  
    605365.439142052,  
    6648891.652304975  
];
```

Snippet from our index.html

In order to set our imageExtent we made a variable for it. The variable consists of the coordinates of our four corners of the mapped area projected to UTM.

```
var vectorSource = new ol.source.Vector(),  
    vectorLayer = new ol.layer.Vector({  
        source: vectorSource  
    });
```

Snippet from our index.html

The snippet above is how we define a vector layer.

```
var view = new ol.View({  
    center: ol.proj.transform(  
        ol.extent.getCenter(imageExtent),  
        "EPSG:32632",  
        "EPSG:3857"  
    ),  
    zoom: 11,  
    minZoom: 11.7,  
    maxZoom: 20  
});
```

Snippet from our index.html

When establishing our view the first thing we set is the center. This is where the map will be focused when we start up. In our case we center it based on our imageExtent variable which was based on the four corners of our mapped area. The zoom variable sets how zoomed in the map will be when it starts up. Min and max zoom establish the limits for zooming. In openlayers a zoom of 1 is the highest you can zoom out, while a high number zooms you in.

When plotting the measurement stations on the map we plot them as map “features” on the vector layer. The following is the function we use to do add them.

```

function addFeature (latitude, longitude, title, content, data,
historical_AVG, chart_label, unit){
    var pos = ol.proj.fromLonLat([longitude, latitude]);
    var feature = new ol.Feature({
        geometry: new ol.geom.Point(pos),
        name: title,
        content: content,
        data: data,
        historical_AVG: historical_AVG,
        chart_label: chart_label,
        unit: unit
    });
    feature.setStyle(iconStyle);
    vectorSource.addFeature(feature);
}

```

Snippet from our index.html

Each feature has a fixed position which we set using its longitude and latitude coordinates. The features also get filled with a number of self defined variables, from name to unit. These variables are later used when we want to get information about the specific station on the map.

Setting the style of the feature establishes how the feature will be visually represented. The settings we use are as follows.

```

var iconStyle = new ol.style.Style({
    image: new ol.style.Icon({
        opacity: 1,
        scale: 0.3,
        src: '{{ url_for('static', filename='./icon.png')}}'
    })
});

```

Snippet from our index.html

We set opacity to 1 so that our markers are completely solid. The scale is set to 0.3 because we made the image source bigger than necessary so that it did not look pixelated on larger screens. src points to the location of our icon file for the stations.

When we want to update the map with new information we need to remove all the stations. This is because the new dataset might use fewer stations, and the stations will need to be assigned new information. The following is the function we use to clear the stations from the map.

```

function clearFeatures() {

```

```

    var features = vectorLayer.getSource().getFeatures();
    features.forEach((feature) => {
        vectorLayer.getSource().removeFeature(feature);
    });
}

```

The function retrieves all the features from the vector layer, then loops through all of them and deletes them.

Geolocation is the identification or estimation of location. We use it to place the users location on the map if they wish it. We accomplish it using the following functions.

```

var custom_button = function() {
    geolocation.setTracking(true);
    view.setCenter( geolocation.getPosition() );
};

```

Snippet from our index.html

When the geolocation button is pressed the functions above get triggered. **setTracking** prompts the browser to share the location information it has with the script. The next line focuses the map on the detected geolocation.

```

var geolocation = new ol.Geolocation({
    projection: view.getProjection()
});

// update the HTML page when the position changes.//
geolocation.on('change', function() {
    updateView();
});

```

Snippet from our index.html

OpenLayers has a geolocation class which we use to set our geolocation variable. Geolocation's **on** method takes two arguments, one is the trigger 'change' which lets us update the map to focus on where the user is when his geolocation updates.

```

var accuracyFeature = new ol.Feature();
geolocation.on('change:accuracyGeometry', function() {
    accuracyFeature.setGeometry(geolocation.getAccuracyGeometry());
});

var positionFeature = new ol.Feature();

```

```

positionFeature.setStyle(new ol.style.Style({
  image: new ol.style.Circle({
    radius: 6,
    fill: new ol.style.Fill({
      color: '#3399CC'
    }),
    stroke: new ol.style.Stroke({
      color: '#fff',
      width: 2
    })
  })
}));

geolocation.on('change:position', function() {
  var coordinates = geolocation.getPosition();
  positionFeature.setGeometry(coordinates ?
    new ol.geom.Point(coordinates) : null);
});

new ol.layer.Vector({
  map: map,
  source: new ol.source.Vector({
    features: [accuracyFeature, positionFeature]
  })
});

```

Snippet from our index.html

The **accuracyFeature** variable is used to determine how accurate the geolocation is. When geolocating it is never 100% exact. It is always within some margin of error, like for example 100 meters. **positionFeature** is used to determine the styling used to visualise the accuracy and position. On detected geolocation change the map updates your position using the “geolocation.on(‘change:position’” function. To display the geolocation information on the map we draw the positionFeature and accuracyFeature information on the **vector layer**.

ol-geocoder

This extension for OpenLayers adds geocoding functionality. Geocoding means that you can search for a place or address and get that location on the map. In order to utilize the geocoding extension we need to start by importing the required JavaScript library and stylesheet.

<link href="https://cdn.jsdelivr.net/npm/ol-geocoder@latest/dist/ol-


```
geocoder.min.css" rel="stylesheet">
<script src="https://cdn.jsdelivr.net/npm/ol-geocoder"></script>
```

Snippet from our index.html

After importing the required files we need to instantiate with some options and add the control.

```
var geocoder = new Geocoder('nominatim', {
  provider: 'osm',
  lang: 'en',
  countrycodes : 'no',
  placeholder: 'Search for ...',
  featureStyle: '',
  limit: 5,
  debug: false,
  autoComplete: true,
  keepOpen: false
});
map.addControl(geocoder);
```

Snippet from our index.html

provider refers to provider for the geocoding information. There are a number of supported providers such as bing and mapquest. We decided to use OpenStreetMap as our provider because it offered us a lot of flexibility as mentioned in chapter 4.5. **lang** determines the preferable language used for the geocoding interface. **countrycodes** allows us to limit the geocoding results to results in Norway. **placeholder** is the text displayed in the geolocation search field when you first open it up. **featureStyle** is used to style the pin which is used to mark the geocoding result. By default if you leave the feature style blank the style used is a small transparent circle, we liked this more than the pin used in the example, so we left it blank. **limit** limits the number of results you display to the number specified, so in our case we will never display more than 5 results. **debug** if true logs the provider's response. **autoComplete** if false searches only for exactly what you type. **keepOpen** if true will keep the results window open after picking your result. The last line in the code snippet is used to add the geocoding search button to the interface.

Sidebar-v2

Sidebar-v2 is an extension which allows for creation dynamic sidebars inside map windows. In order to implement sidebar-v2 we needed to download the javascript library and stylesheet and import them.

```
<link rel="stylesheet" href="{ url_for('static', filename='./sidebar-v2-master/css/ol3-sidebar.css') }" />
<script src="{ url_for('static', filename='./sidebar-v2-master/js/ol3-
```

```

sidebar.js'}}"></script>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-
awesome/4.1.0/css/font-awesome.min.css">

```

Snippet from our index.html

```

<div id="sidebar" class="sidebar sidebar-right collapsed">
  <!-- Nav tabs -->
  <div class="sidebar-tabs">
    <ul role="tablist">
      <li><a href="#home" role="tab"><i class="fa fa-
bars"></i></a></li>
      <li><a href="#academic" role="tab"><i class="fa fa-
graduation-cap"></i></a></li>
    </ul>
  </div>

```

Snippet from our index.html

As shown above the tab buttons are set up as list items with href's referring to different divs that come later. The icons for the tabs are acquired using font awesome library⁸⁵.

```

<!-- Tab panes -->
<div class="sidebar-content">
  <div class="sidebar-pane" id="home">
    <h1 class="sidebar-header">
      Oslo Air Quality
    <span class="sidebar-close"><i class="fa fa-caret-
left"></i></span>
    </h1>
    <br>
    <p>The map shows the concentration of different air components
that can have negative health effects.</p>
    
    <br><br>
    <p>The colour on the map represents the health risk at that
point, green indicates no health risk while red indicates a substantial
health risk.</p>

    <p>At the bottom left corner you can pick the air component

```

⁸⁵ <https://fontawesome.com/how-to-use/svg-with-js> Accessed 11 May. 2018.

you want to display such as `PM10 or PM2.5.</p>`

```
<p><a href="https://www.nilu.no/">NILU</a> releases new data
which our prediction is based off every 3 hours. Range indicates how many
predictions back you want to display. For example 5 would mean 15 hours
back. </p>
</div>
```

Snippet from our index.html

Above is an example of how we format the main tab in our sidebar.

```
var sidebar = new ol.control.Sidebar({ element: 'sidebar', position:
'right' });
map.addControl(sidebar);
```

Snippet from our index.html

The lines above is how we make a connection between the page elements and the JavaScript library and append them to the map the same way as on screen controls.

Bootstrap Popover

In order to create popup windows when we click on specific map elements we use popover. In order to utilize the bootstraps popover we need to start by importing the required JavaScript libraries and stylesheet.

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.c
ss">
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"
></script>
<script src="https://code.jquery.com/jquery-2.2.3.min.js"></script>
```

Snippet from our index.html

```
map.on('click', function(evt) {
  var feature = map.forEachFeatureAtPixel(evt.pixel,
    function(feature) {
      return feature;
    });
  if (feature) {
    $(element).popover('destroy');
    overlay.setPosition(undefined);
    closer.blur();
  }
});
```

```

var coordinates = feature.getGeometry().getCoordinates();
popup.setPosition(coordinates);
$(element).popover({
  'title': feature.get('name'),
  'placement': 'bottom',
  'animation': false,
  'html': true,
  'content': feature.get('content')
});
$(element).popover('show');

```

Snippet from our index.html

The snippet above is triggered when we click on map elements. If the map element is a feature we destroy any existing popovers and generate one for the station we have clicked on. The **title** we set is what will appear at the top of the popover. **Placement** refers to how the popover the popover will appear in relation to its origin. **Animation** if true applies a CSS fade transition to the popover. It is important to keep animation false for our use case since if it is true it causes problems when creating new popovers. **HTML** true lets us insert HTML into the popover from inside the content, if false it all HTML in the content would be displayed as ASCII text. **Content** is the information displayed in the popover other then the title. We retrieve the content from the feature we click on.

Dataframe-js

Using templates described at the end of the backend chapter, we turn the Python dataframe we sendt with the template into a dataframe-js objects. This way we can do operations on our data in the dataframe-js object on the client side. This gives us a lot of flexibility, and allows us for example to make charts. To utilize the dataframe-js we made a couple of functions.

```

function getDataframeIndex(){
  return document.getElementById("component").value;
}

```

In main.py we explained that we split up the data concerning each component into different dataframes. The function above returns the index for a component's dataframe.

```

function getDataframe(){
  return datafrm[getDataframeIndex()];
}

```

Using the first function `getDataframe()` returns the relevant dataframe.

```
function getRow(i){
  return datafrm[getDataframeIndex()].find(row => row.get('index') == i);
}
```

`getRow()` will return the row for the index passed as a parameter.

```
function changeRowValue (index, column, value){
  var new_row = datafrm[getDataframeIndex()].find(row =>
row.get('index') == index);
  new_row = new_row.set(column, value);

  datafrm[getDataframeIndex()] =
datafrm[getDataframeIndex()].transpose();

  for(var i = 0; i <
datafrm[getDataframeIndex()].listColumns().length; i++){
    if (i == index){
      datafrm[getDataframeIndex()] =
datafrm[getDataframeIndex()].drop(String(i));
    }
  }
  datafrm[getDataframeIndex()] =
datafrm[getDataframeIndex()].transpose();
  datafrm[getDataframeIndex()] =
datafrm[getDataframeIndex()].renameAll(['date', 'component', 'data',
'_id', 'index', 'img', 'contour']);
  datafrm[getDataframeIndex()] =
datafrm[getDataframeIndex()].push(new_row);
  datafrm[getDataframeIndex()] =
datafrm[getDataframeIndex()].sortBy('date', true);
}
```

Code snippet from index.html

To change a cell we have made this helper function. It first gets the row with `dataframe-js`'s “find” function and then we insert the data with row's `set` function.

To delete our old row we first use the “transpose” function turning the dataframe 90 degrees, this turns the rows into columns. Then we for-loop through the columns till we find the old row and drop it.

To end off the function we transpose it again, turning the columns back to rows and rename all the columns to their previous names. Then we use dataframes “push” function to add our new row.

In the chapter about OpenLayers we talked about placing the measurement stations as points on the map. To place each station on the map every time we update the map with new information we use the dataframe-js’s “chain” function. “chain” goes through all of the dataframe’s rows and does operations on each of them. Each row will correspond with a measurement station on the map. We call our “addFeature” function to make a feature with that row’s data.

Preloading with AJAX JQuery

To load the interpolated pollution images we overlay on the map, we use JQuery to make an request to the server. While we load we turn on a loading text that will tell the user we are downloading from the server.

```
// loaded functions
var loaded = 0;
function incrementLoaded(){
    loaded++;
    if (loaded == getRange()*2){
        updateMap();
        setLoadingTextVisible(false);
    }
}
function resetLoaded(){
    loaded = 0;
}
```

Snippet from our index.html

To keep track of what is loaded we have the loaded variable that we increment using the increment function. getRange returns the amount of images the user has requested to load, and once we reach that number we update the map and turn off the loading text.

```
function load(id){
    $.get( '/img/' + id , function(data, status) {
    }).done(function() {
```

```

        //when the request has finished:

        var img = new Image();
        img.src = '/img/' + id;
        img.title = id;

        changeRowValue_id(id, 'img', img);
        incrementLoaded();
    }).fail(function() {
        //when the request has failed:
        setLoadingTextFailure();
        console.error( id, "loading contour failed for", id);
    });

$.get( '/contour/' + id , function(data, status) {
}).done(function() {
    //when the request has finished:
    var contour = new Image();
    contour.src = '/contour/' + id;
    contour.title = id;

    changeRowValue_id(id, 'contour', contour);
    incrementLoaded();
}).fail(function() {
    //when the request has failed:
    setLoadingTextFailure();
    console.error( "loading contour failed for", id);
});
}

```

Snippet from our index.html

The load function takes id as an argument, this id will correspond with a entry in the database. We use the “\$.get” function on the address /img/<id> this is the function we described at the main.py in the backend chapter. Once our request is completed we increment our loaded variable that keeps track of our progress. We store an image object in dataframe-js with our helper function changeRowValue.

If it fails we use the setLoadingTextFailure to inform the user that something went wrong.

Chart.js

We use chart.js^{86 87} to make a visual representation of our statistics. We generate these charts when a popover is made. All the data we use to make the chart is fetched from the feature.

```
var chr = document.getElementById('chart');
var chrs = new Chart(chr, {
  type: 'line',
```

Snippet from our index.html

We start out with getting the chart HTML element we have in our popup box. We use the this reference when making a new instance of chart.js chart class.

```
data: {
  labels: feature.get('chart_label'),
  datasets: [
    {
      label: 'value',
      data: feature.get('data'),
      borderWidth: 2,
      fill: false,
      borderColor: 'black'
    },
    {
      label: 'historical average',
      data: feature.get('historical_AVG'),
      borderWidth: 2,
      fill: false,
      borderColor: 'red'
    }
  ]
},
```

Snippet from our index.html

Data declares what data to be used in the chart, inside it we have labels and datasets. The datasets consists of an array containing the fields label, data, borderWidth, fill and borderColor. We make two lines in our chart one with historical average and one with the current values stored in the feature. The rest of the fields in each are used to determine the visuals of each line, like for example how thick they are.

```
options: {
  scales: {
```

⁸⁶ "Chart.js." <https://www.chartjs.org/>. Accessed 21 May. 2018.

⁸⁷ "Chart.js · GitHub." <https://github.com/chartjs>. Accessed 21 May. 2018.


```

xAxes: [{
  type: 'time',
  ticks: {
    source: 'labels'
  },
  time: {
    format: 'MM/DD/YYYY HH:mm',
    tooltipFormat: 'HH:mm:ss',
    unit: 'minute'
  }
}],

```

Snippet from our index.html

Options is used to set some properties of the chart. In the code snippet above we set the scale of the xAxis to be time. In the last 6 lines we further declare what format the time should be.

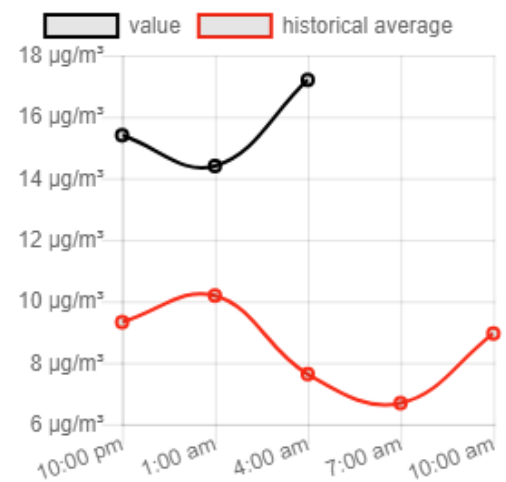
```

yAxes: [{
  ticks: {
    callback: function(value, index, values) {
      return value + " "+feature.get('unit');
    }
  }
}]
}
});

```

Snippet from our index.html

In the yAxes options we change the properties of the yAxes ticks. We use chart.js's callback to manually say how we want the ticks to be written. This lets us add the unit behind the value, as illustrated.



Screenshot of chart from our web app

5.8 Further development

If we had more time to develop this project there are a number of things we would improve or add. One of the main things we would like to add is an overall estimation of the air quality. This would involve combining the different interpolated components we generate. When combining the data it is important to know how the different components interact and affect each other. We could easily have combined the data we have in a way where we keep the highest values for each component and generated results based on that. The problem with that is that it not a true result, meaning that we would spread misinformation.

Let us imagine that a point has the following values PM10: 70, PM2.5: 55, NO2: 230. This would mean that the PM10 air quality index⁸⁸ is 7, the PM2.5 air quality index⁸⁹ is 7 and the NO2 air quality index⁹⁰ is 4. This means that the PM10 and PM2.5 are at a similar health risk level, and that NO2 has a lower health risk comparatively. Knowing all this what should the health risk value at that point be? Since PM10 and PM2.5 are the highest values maybe the overall health risk should be based on them? This might be correct, but it does not take into account how the different air components interact with each other, and how different compositions affect health. These are all questions that need to be researched and answered before making a conclusive overall estimation.

⁸⁸PM10 air quality index <https://uk-air.defra.gov.uk/air-pollution/daq?view=more-info&pollutant=pm10#pollutant> Accessed 11 May. 2018

⁸⁹PM2.5 air quality index <https://uk-air.defra.gov.uk/air-pollution/daq?view=more-info&pollutant=pm25#pollutant> Accessed 11 May. 2018

⁹⁰NO2 air quality index <https://uk-air.defra.gov.uk/air-pollution/daq?view=more-info&pollutant=no2#pollutant> Accessed 11 May. 2018

A smaller change we would make to the website is using a date picker to pick dates, instead of our current range solution. Another thing we would have added to our webapp is popup information anywhere. This would mean clicking anywhere on the map and getting a popup that displayed the estimated value closest to that point. This popup should feature the value, a colour representation of that value and could also show this information for all components at that point. Also similarly if you use the geocoding feature, you should get a popup for the location you searched for. If you use the location services on the map you could have a small window in the corner of the display showing the current value, which updates as you move around. This would help inform the user of the exact value for a location.

If we had this service running for over a year it would be interesting to use the data from the year prior to compare the results to the current year. This would let us see how much change there is in air quality year to year, during the same dates and times.

We had little time to bug test the features we implemented towards the end of the development period. The charts were the last thing we implemented and they turned out to have some complications such as NILU not being consistent on the 3 hour update they specified in the open API. The stations also do not always participate in the measurements. These inconsistencies make our rigid chart implementation not always work optimally. If we had more time we would make a less rigid chart implementation using the time scale option for chart-js to make all the points within a 6 hour difference turn up on the chart.

6 Summary

This project's goal has been to develop a web portal for visualizing air quality data. We have used Python Flask, Kriging interpolation, data from NILU, JavaScript packages and OpenLayers mapping to develop our application. The solution works on both desktop computers as planned, in addition it works on smartphones and tablets however, with limited optimization. We have outlined a set of possible extensions to our solution in further development. This includes overall estimation of air quality and other things we did not have time for.

We have learned a lot during this project period, especially the use of OpenLayers, Python, Python Flask, IBM Cloud, and executing a larger development project. It has been exciting to program in Python for the first time, and programming with the various tools we discovered. IBM Cloud have been a really good experience with many useful tutorials and good documentation.

It has been a fun experience to work on such a project, and we are happy to have been able to create our final product. If we were to start this project over again, we would have spent more time documenting the progress and choices we made during our development. It has been a very useful learning experience to discover how much work goes into documenting a project of this scale.

The communication and meetings we had with IBM have been an invaluable experience. We are happy to hear that they are excited to use what we have made. It will be shown off at their Oslo offices once we deliver the final version.