Hochschule Ulm

Masterproject

Data Science

Analytics and Prediction for Rental Bike Usage in London

Contributors:

Anass Khaldi, Guillaume Goni, Kathi Rodi, Pascal Riedel

Expert: Prof. Dr. von Schwerin

Expert: Prof. Dr. Herbort

Expert: Prof. Dr. Goldstein

Friday 21st June, 2019

Contents

Chapter 1

Project Presentation and Organization

1 Introduction

No matter in which area you move there is a huge chance that data science will be there too. It is one of the fast moving trends during the last years which is caused by the unimaginable amount of data we produce everyday. We could gain a lot of knowledge out of this data, but therefore we must analyze it. That is when data science comes into play. But with the help of data science we can not just analyze the past we can also predict the future.

Let's say we have a bike rental station which provides to rent bikes for a certain amount of time. We track the duration of the rental, the time of rental, the routes the bikes rode and lot more information. By processing all this data it is possible to predict the future usage of the bikes and the most frequented routes to specific times. That is what the main goal of this project represents. This project report deals with tasks which can be summarized by *preparatory steps* to reach this final goal. The project duration is divided into two semesters. The tasks of the first part will be discussed in this paper.

2 Task Description (Semester 1)

The first part can be roughly divided into two parts:

1. Data Engineering

- Data Profiling
- identify potential difficulties
- store data set

2. Virtual Infrastructure

   - install and configure Nominatim
   - install and evaluate different Hadoop distributions

The project should be implemented in an agile manner, therefore we will follow the development process of **Scrum**. This allows flexibility during the project in terms of specific tasks and topics.

To gain a better understanding of the project and its goals we created a „Big Picture“of the applied technologies and the relations among themselves which can be seen in figure 2.1.

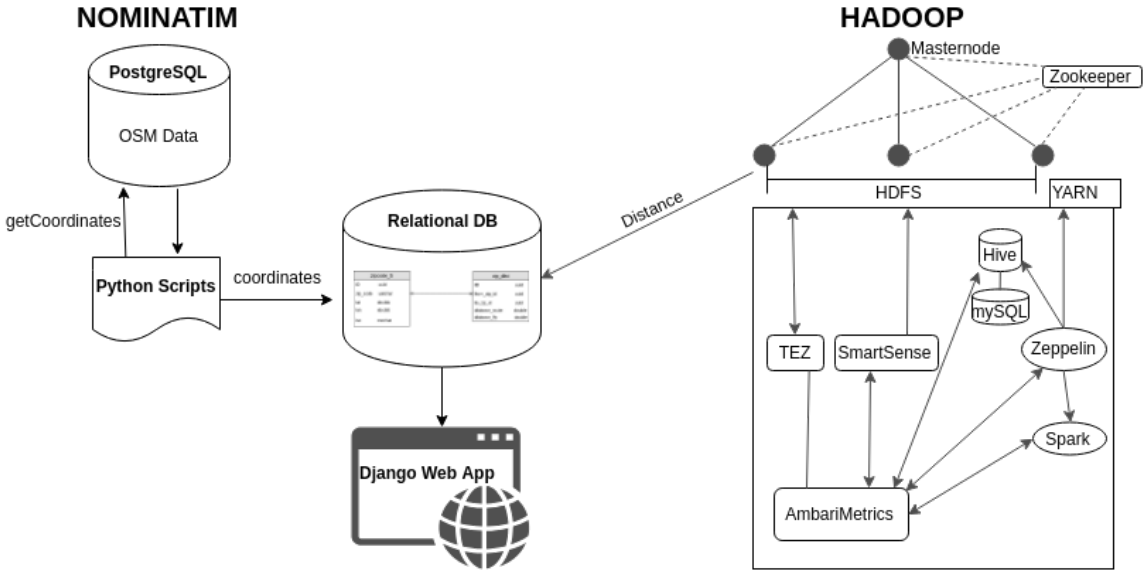

Figure 2.1: Overview of project architecture

According to this picture the project can be classified more detailed into four parts:

1. Nominatim

2. Hadoop
3. Database
4. WebApp

The implementation of those four parts will be documented in the following chapters.

3 Task Description (Semester 2)

4 Collaboration Technologies

Since we want to ensure efficiency which includes that everyone has access to required data, can exchange data and that data can be versioned, we decided to work with several technologies which will be explained further in the following sections.

4.1 IceScrum

As mentioned in the chapter before the development process should be done by using Scrum methods. Those methods includes *Sprint Plannings, Daily Scrums, Retrospectives* and *Sprint Reviews*. To apply the methods properly it was very helpful to use *IceScrum* which is a web application in order to use Scrum. It provides storyboards on which tasks can be accepted. Furthermore it helps to keep track of progress and remaining tasks by burndowncharts as well as reports. We used IceScrum mainly for the sprint planning. To be precise for writing, estimating, prioritizing tasks, maintain our backlog and to distribute the incurred tasks as well as keeping track of our remaining story points and our progress.
IceScrum offers furthermore different roles which should be satisfied in Scrum. There is the *Team* which is responsible to deliver the committed delivery in time and with the defined quality. In our case the Team consists of Anass Khaldi, Guillaume Goni and Pascal Riedel. Within the team there is one *Scrum Master*, this role was taken by Kathi Rodi. The Scrum Master is responsible to ensure the team keeps to the values and practices of Scrum, sort of like a coach. He removes impediments, facilitates meetings and work with product owners as well. This leads us to the next role in Scrum the

Product Owner which is the project‘s key stakeholder. The Product Owner does not get to determine how much work happens in the sprint cycles, or alter the goals for that sprint. Product Owners must be available to the team, and engage actively with it. In this project the Product Owners are Prof. Dr. von Schwerin, Prof. Dr. Herbort and Prof. Dr. Goldstein.

4.2 Box

Besides IceScrum we also used *Box* for collaborating. Our Box includes for example the documentation of our daily scrums. We created a template which is used to log every daily meeting. According burndowncharts and reports will be stored there as well. Furthermore we used the box for recording our working hours and to exchange documents.

4.3 GitHub

To group our code base and have versioned on it, we created a GitHub account which is available at https://github.com/dataBikeHsUlm. For the sake of clarity we created three repositories ordered by the main topics:

- WebApp: for the web server
- NominatimLibrary: for the Jupyter notebooks
- MapReduce: for the scripts for Hadoop

4.4 Skype

We carried out our daily scrums every monday at 4 pm as well as every thursday at 1.30 pm. On thursdays we have lectures together the whole day, hence the daily scrums on thursdays takes place at the university. Since it wasn’t‘t possible for the whole team to meet mondays as well, we decided to hold those daily scrums via Skype.

5 Responsibilities

Within Scrum the team is self-organizing and cross-functional. This means that every team member can choose tasks fitting his knowledge base as well

as his preferences and interests. Hence everyone was free to suggest and accept the tasks he liked most.
It turned out that the team leads a very good self-organization because everyone of us has a specific interest and knowledge according to the tasks of the project.
Therefore Anass Khaldi was responsible to install the Nominatim server and it's components. In the second sprint his main part was to research Django and how we can display maps on our web application. In the third sprint it was his responsibility to install graphhopper and to fix the according issues as well as issues of hadoop.
The main part of the Jupyter notebook to research nominatim functions like compute centroids, compute distances by route and doing all in bulk was done by Guillaume Goni. In the second sprint he created all GitHub repositories for storing our work. He wrote the script to initialize the database as well as the one to fill the database. Therefore he researched for best ways to collect zip codes and find the solution in the PostgresDB. Moreover he prepared a tutorial about Django for the rest of the team members.
As mentioned before Kathi Rodi took over the role of the Scrum Master, hence it was her responsibility to ensure that the scrum artifacts were kept as well as correspond with the Product Owners. Apart from her tasks as a Scrum Master she was responsible to provide a box for collaborating with all needed documents like templates for daily scrums, sprint plannings, reviews, retrospectives and tracking the working hours. Furthermore she created the presentation for every sprint. Moreover she solved several tasks like create Jupyter notebooks for geocode, reverse geocoding and compute distances as the crow flies, create the ER model, install the MySQL database and to research for Graphhopper. In the end it was her responsibility to write a project report and add the task descriptions of the team members to it.
Pascal Riedel was responsible for research, document and compare the different Hadoop distributions. With the help of his weightened comparison table we could take the decision for one specific distribution. It was also his part to install the HDP as well as to implement a MapReduce job. During the first project phase he solved several issues according to HDP. Furthermore he was responsible for the data profiling part and to create associated Zepellin notebooks.
For more details please have a look at the responsibility table which could be found in attachment 2.

Chapter 2

Postal Code Database (Nominatim and Graphhopper)

1 Nominatim

Nominatim is a tool to search OpenStreetMap (OSM) data by name and address and to generate synthetic addresses of OSM points (reverse geocoding). Nominatim provides geocoding based on OpenStreetMap data. It uses a PostgreSQL database as a backend for storing the data. There are three basic parts to Nominatim's architecture: the data import, the address computation and the search frontend. The data import stage reads the raw OSM data and extracts all information that is useful for geocoding. This part is done by **osm2pgsql**, the same tool that can also be used to import a rendering database. It uses the special gazetteer output plugin in **osm2pgsql/output-gazetter.[ch]pp**. The result of the import can be found in the database table **place**. The address computation or indexing stage takes the data from **place** and adds additional information needed for geocoding. It ranks the places by importance, links objects that belong together and computes addresses and the search index. Most of this work is done in Procedural Language/PostgreSQL (PL/pgSQL) via database triggers and can be found in the file **sql/ functions.sql**.
The search frontend implements the actual API. It takes queries for search and reverse geocoding queries from the user, looks up the data and returns the results in the requested format. This part is written in php and can be found in the **lib/** and **website/** directories.

When we received the Nominatim server, we created the non-administrator user account for our use and we added the SSH public keys of all the team members.

1.1 Installation

These instructions expect that you have a freshly installed Ubuntu 18.04. Make sure all packages are are up-to-date by running:

```
sudo apt-get update -qq
```

Now you can install all packages needed for Nominatim:

```
sudo apt-get install -y build-essential cmake g++
    libboost-dev libboost-system-dev \
libboost-filesystem-dev libexpat1-dev zlib1g-dev
    libxml2-dev\
libbz2-dev libpq-dev libproj-dev \
postgresql-server-dev-10 postgresql-10-postgis-2.4 \
postgresql-contrib-10 \
apache2 php php-pgsql libapache2-mod-php php-pear php-
    db \
php-intl git
```

If you want to run the test suite, you need to install the following additional packages:

```
sudo apt-get install -y python3-setuptools python3-dev
     python3-pip \
python3-psycopg2 python3-tidylib phpunit php-cgi
pip3 install --user behave nose
sudo pear install PHP_CodeSniffer
```

System Configuration

The following steps are meant to configure a fresh Ubuntu installation for use with Nominatim. You may skip some of the steps if you have your OS already configured. Nominatim will run as a global service on your machine. It is therefore best to install it under its own separate user account. In the following we assume this user is called **nominatim** and the installation will be in **/srv/nominatim**. To create the user and directory run:

```
sudo useradd -d /srv/nominatim -s /bin/bash -m
   nominatim
```

You may find a more suitable location if you wish. To be able to copy and paste instructions from this manual, export user name and home directory now like this:

```
export USERNAME=nominatim
export USERHOME=/srv/nominatim
```

Never, ever run the installation as a root user. Make sure that system servers can read from the home directory:

```
chmod a+x $USERHOME
```

Setting up PostgreSQL

Tune the PostgreSQL configuration, which is located in **/etc/postgresql/9.5/main/postgresql.conf**. You might want to tune your PostgreSQL installation so that the later steps make best use of your hardware. You should tune the following parameters in your **postgresql.conf** file.

```
shared_buffers (2GB)
maintenance_work_mem (10GB)
work_mem (50MB)
effective_cache_size (24GB)
synchronous_commit = off
checkpoint_segments = 100 # only for postgresql <= 9.4
checkpoint_timeout = 10min
checkpoint_completion_target = 0.9
```

The numbers in brackets behind some parameters seem to work fine for 32GB RAM machine. Adjust to your setup. For the initial import, you should also set:

```
fsync = off
full_page_writes = off
```

Don't forget to reenable them after the initial import or you risk database corruption. **Autovacuum** must not be switched off because it ensures that the tables are frequently analyzed.

Setting up the Webserver

The **website** directory in the **build** directory contains the configured website. Include the directory into your web browser to serve php files from there.
Make sure your Apache configuration contains the required permissions for the directory and create an alias:

```
<Directory "/srv/nominatim/build/website">
Options FollowSymLinks MultiViews
AddType text/html
.php
DirectoryIndex search.php
Require all granted
</Directory>
Alias /nominatim /srv/nominatim/build/website
```

The path of the build directory (**/srv/nominatim/build**) should be replaced with the location of your build directory. After making changes in the apache config you need to restart apache. The website should now be available on http://localhost/nominatim. Restart the PostgreSQL service after updating this config file.

```
sudo systemctl restart postgresql
```

Finally, we need to add two PostgreSQL users: one for the user that does the import and another for the webserver which should access the database for reading only:

```
sudo -u postgres createuser -s $USERNAME
sudo -u postgres createuser www-data
```

You need to create an alias to the website directory in your apache configuration. Add a separate nominatim configuration to your webserver:

```
sudo tee /etc/apache2/conf-available/nominatim.conf <<
    EOFAPACHECONF
<Directory "$USERHOME/Nominatim/build/website">
Options FollowSymLinks MultiViews
AddType text/html
.php
DirectoryIndex search.php
```

```
    Require all granted</Directory>
    Alias /nominatim $USERHOME/Nominatim/build/website
EOFAPACHECONF
```

Then enable the configuration and restart apache:

```
sudo a2enconf nominatim
sudo systemctl restart apache2
```

Building and Configuration

Get the source code for the release and change into the source directory:

```
cd $USERHOME
wget https://nominatim.org/release/Nominatim-3.2.0.tar
    .bz2
tar xf Nominatim-3.2.0.tar.bz2
cd Nominatim-3.2.0
```

The code must be built in a separate directory. Create this directory, then configure and build Nominatim in there:

```
mkdir build
cd build
cmake ..
make
```

You need to create a minimal configuration file that tells nominatim where it is located on the webserver:

```
tee settings/local.php << EOF
<?php
@define('CONST_Website_BaseURL', '/nominatim/');
EOF
```

After those steps Nominatim is ready to use.

1.2 Database import

The following instructions explain how to create a Nominatim database from an OSM planet file and how to keep the database up to date. It is assumed that the Nominatim software itself is already successfully installed.

Flatnode files

If you plan to import a large dataset (e.g. Europe, North America, planet), you should also enable flatnode storage of node locations. With this setting enabled, node coordinates are stored in a simple file instead of the database. This will save you import time and disk storage. Add to your settings/local.php:

```
@define('CONST_Osm2pgsql_Flatnode_File', '/path/to/
    flatnode.file');
```

Replace the second part with a suitable path on your system and make sure the directory exists. There should be at least 40GB of free space.

Initial import of the data

Important: first try the import with a small excerpt, for example from **Geofabrik**. Download the data to import and load the data with the following command:

```
./utils/setup.php --osm-file <data file> --all [--
    osm2pgsql-cache 28000] 2>&1 | tee setup.log
```

The –**osm2pgsql-cache** parameter is optional but strongly recommended for planet imports. It sets the node cache size for the **osm2pgsql** import part (see -C parameter in osm2pgsql help). As a rule of thumb, this should be about the same size as the file you are importing but never more than 2/3 of RAM available. If your machine starts swapping reduce the size. Computing word frequency for search terms can improve the performance of forward geocoding in particular under high load as it helps PostgreSQL' query planner to make the right decisions. To recompute word counts run:

```
./utils/update.php --recompute-word-counts
```

This will take a couple of hours for a full planet installation. You can also defer that step to a later point in time when you realise that performance becomes an issue. Just make sure that updates are stopped before running this function.

Updating the data

There are many different possibilities to update the Nominatim database. The following section describes how to keep it up-to-date with **Pyosmium**.

For a list of other methods see the output of **./utils/update.php –help**. It is recommended to install Pyosmium via pip. Run (as the same user who will later run the updates):

```
pip install --user osmium
```

Nominatim needs a tool called **pyosmium-get-updates**, which comes with Pyosmium. You need to tell Nominatim where to find it. Add the following line to your **settings/local.php**:

```
@define('CONST_Pyosmium_Binary', '/home/user/.local/
    bin/pyosmium-get-changes');
```

The path above is fine if you used the **–user** parameter with pip. Replace user with your user name.
Next the update needs to be initialised. By default Nominatim is configured to update using the global minutely diffs. If you want a different update source you will need to add some settings to **settings/local.php**. For example, to use the daily country extracts diffs for Ireland from geofabrik add the following:

```
// base URL of the replication service
@define('CONST_Replication_Url', 'https://download.
    geofabrik.de/europe/ireland-and-northern-
ireland-updates');
// How often upstream publishes diffs
@define('CONST_Replication_Update_Interval', '86400');
// How long to sleep if no update found yet
@define('CONST_Replication_Recheck_Interval', '900');
```

To set up the update process now run the following command:

```
./utils/update.php --init-updates
```

It outputs the date where updates will start. Recheck that this date is what you expect. The **–init-updates** command needs to be rerun whenever the replication service is changed.
The following command will keep your database constantly up to date:

```
./utils/update.php --import-osmosis-all
```

Note that even though the old name „import-osmosis-all“has been kept for compatibility reasons, Osmosis is not required to run this - it uses Pyosmium behind the scenes.
Nominatim will response with the following data formats:

- JSON
- JSONv2
- GeoJSON
- Geocode JSON
- XML

1.3 Usage

As we want to use python for working with Nominatim we use the python package **geopy** to interact with the server.
Up to now, we were using the default public Nominatim server [29]. As we want to use our own Nominatim server we researched for connecting the geopy library to it.
First, the Nominatim server needed to be configured with Apache, the configuration that was done earlier wasn't working properly so we firstly had to fix that. The server can be accessed at the following url: http://i-nominatim-01.informatik.hs-ulm.de/nominatim/.
After that HTTPS isn't enabled on the server. Of course, we can set it up later but for now we need to disable it for geopy, it is done by setting:

```
geopy.geocoders.options.default_scheme = "http"
```

Third, our server is not very powerful and being quite new, it has not cached enough queries. This can make some queries take a long time to complete and according to that the geopy query timeouts. Therefore we had to increase this limit. The final connection is shown in figure 1.1.

```
1 from geopy.geocoders import Nominatim
2
3 DOMAIN = "i-nominatim-01.informatik.hs-ulm.de/nominatim/"
4 TIMEOUT = 100000
5
6 geopy.geocoders.options.default_scheme = "http"
7 locator = Nominatim(user_agent="Test", timeout=TIMEOUT, domain=DOMAIN)
```

Figure 1.1: Connection of Nominatim with geopy

1.4 Notebooks

In attempt to gain more knowledge on how it is possible to work with Nominatim in python we created several Jupyter notebooks. The different methods we need on the Nominatim API are grouped in a python library on the project's GitHub repository which can be accessed under https://github.com/dataBikeHsUlm/NominatimLibrary. It needs **geopy** as a dependency and **osmapi** for **pyroutelib2** which can be installed via **pip**:

```
pip3 install geopy osmapi
```

Geocoding

Geocoding an adress means we input an adress and as an output we expect the corresponding coordinates. Therefore we have to extract the coordinates as follows:

```
def locate (address):
    """Gets the coordinates to a given address

        Args:
            address(str): String List of following
               arguments:house_number,road, town, city
               , county, state_district, state,
               postcode, country, country_code

        Returns:
            Associated address and coordinates
  """
    location = geolocator.geocode(address)
    print(location.address)
    print(location.latitude, location.longitude)
```

We could also extract coordinates not by searching for a whole adress but by postalcodes:

```
def locateCords(postcode):
    """Gets the coordinates to a given address

        Args:
            postalcode(str): String of postalcode
```

```
        Returns:
            Associated coordinates as latitude and
               longitude
    """
    location = geolocator.geocode(postcode)
    lat = location.latitude
    lon = location.longitude
    print"The Latitude of ",postcode," is: ",lat," the
        Longitude is ",lon
    return (lat,lon)
```

The results are the corresponding coordinates.

Reverse Geocoding

Vice versa we can apply reverse geocoding to compute the corresponding address to given coordinates.

```
def reverseLocate (coordinates):
    """Gets the address to given coordinates

        Args:
            coordinatates(str): String List of
               coordinates

        Returns:
            Associated address
    """
    location = geolocator.reverse(coordinates)
    print(location.address)
```

The result was compared with Google Maps which is shown in figure 1.2.

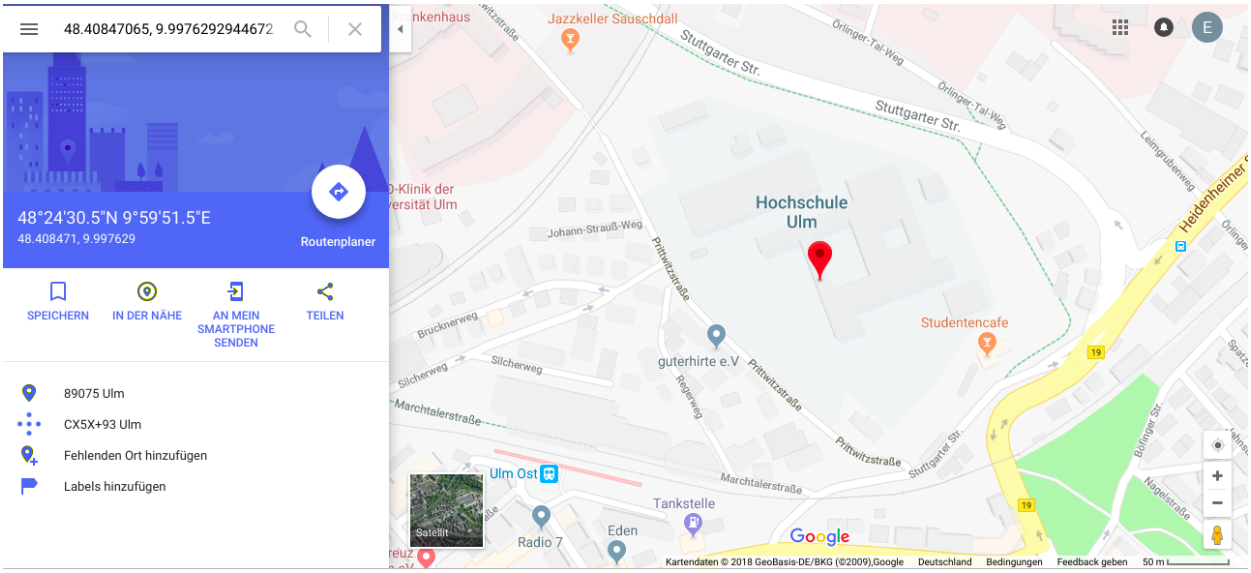

Figure 1.2: Reverse Geocoding

Querying Centroids

As already mentioned geopy offers methods to access addresses and coordinates. We used this as follows for querying centroids:

```
location = locator.geocode("Baden-Wuerttemberg,
    Deutschland")
print(location.address)
print(location.latitude, location.longitude)
```

We received the following result:

```
Baden-Wuerttemberg, Deutschland
48.6296972 9.1949534
```

We entered the given coordinates for the centroid of Baden-Wuerttemberg (**48.6296972, 9.1949534**) in Google Maps and made a route between this centroid and the one given by Google Maps as is shown in figure 1.3.

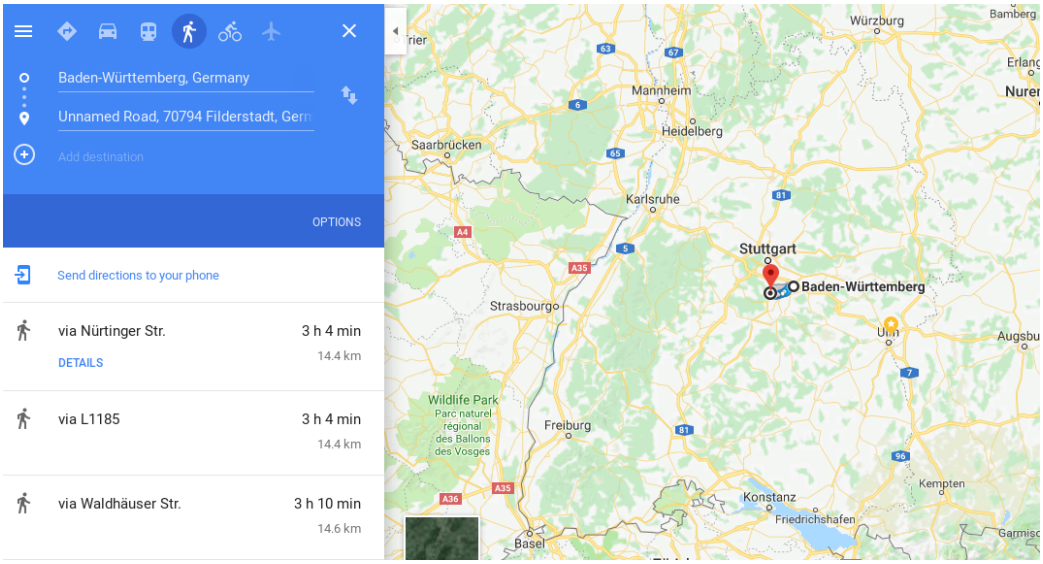

Figure 1.3: Centroid of Baden-Wuerttemberg shown in Google Maps

We can see that there is only 14.4km between the two points. If we consider Baden-Wuerttemberg is roughly 200km wide, we have a difference of 7.2%. The values used here are obviously not very accurate but we can still see that the given centroid is located roughly in the center of the requested region. The difference with Google Maps can maybe be explained by a difference of precision in the borders.

Distances as the crow flies

To compute the distance between two locations as the crow flies we can use the **distance()** method which is provided by geopy. Therefore we have to transfer coordinates into points:

```
def locatePoint (postcode):
    """Gets the coordinates to a given address

        Args:
            postalcode(str): String of postalcode

        Returns:
            Associated coordinates as a Point
```

```
    """
    location = geolocator.geocode(postcode)
    lat = location.latitude
    lon = location.longitude
    p = location.point
    return p
locatePoint("80336")
```

Subsequently we can compute the associated routes:

```
def getDistance(start, end):
    """Get distance of two postalcodes

        Args:
            start(str): String of start postalcodes
            end (str): String of end postalcode

        Returns:
            Distance in km
    """
    a = locatePoint(start)
    b = locatePoint(end)
    d =distance.distance(a,b).km
    print"The distance between Ulm and Munich is: ",d
        ," km"
    return d

getDistance("89075","80336")
```

The result was compared to Google Maps and can be seen in figure 1.4.

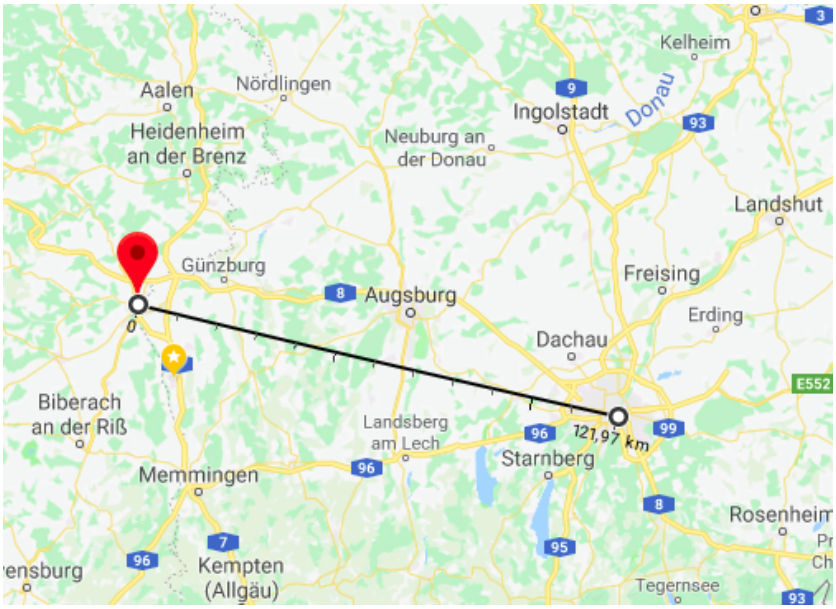

Figure 1.4: Compute Distances as the Crow Flies shown in Google Maps

Distances by Road

There is not much library providing a routing algorithm for python and working with OpenStreeMap. But we found **pyroutlib2** [30]. It does not seem to be supported therefore we had to make a few modifications to the original code to make it work.
The modified version can be found in the NominatimLibrary GitHub repository.
The dependency osmapi is required.
First, we get the coordinates of the starting and finishing points:

```
location = locator.geocode("Prittwitzstrasse, Ulm")
a = (location.latitude, location.longitude)
location = locator.geocode("Albert-Einstein-Allee, Ulm
    ")
b = (location.latitude, location.longitude)
from pyroutelib2.loadOsm import LoadOsm
from pyroutelib2.route import Router
# By default, it uses the open API, this can be
    changed directly in the file
```

```
# Here, we use bicycle to calculate routes.
data = LoadOsm("cycle")
router = Router(data)
# This gets the node ids of the two points
node_a = data.findNode(a[0], a[1])
node_b = data.findNode(b[0], b[1])
# `doRoute` calculates the route and returns a list of
    coordinates tuples
result, route = router.doRoute(node_a, node_b)
if result == 'success':
# Do something...
pass
else:
print("Error calculating the route.")
```

Finally, to get the distance, we compute the distance between each couple of points:

```
from geopy import distance
lats = []
lons = []
if result == 'success':
for i in route:
node = data.rnodes[i]
lats.append(node[0])
lons.append(node[1])
else:
print("Error calculating the route.")
distance_route = 0
for i in range(len(lons)-1):
p1=(lats[i],lons[i])
p2=(lats[i+1],lons[i+1])
distance_route += distance.geodesic(p1,p2, ellipsoid='
   GRS-80').km
print("Total distance on the route : %.2fkm" %
   distance_route)
```

As a result we get from Prittwitzstrasse to Albert-Einstein-Allee, a distance by route of 5.87 km. Google Maps gives a similar result of 5.2 km as shown in figure 1.5.

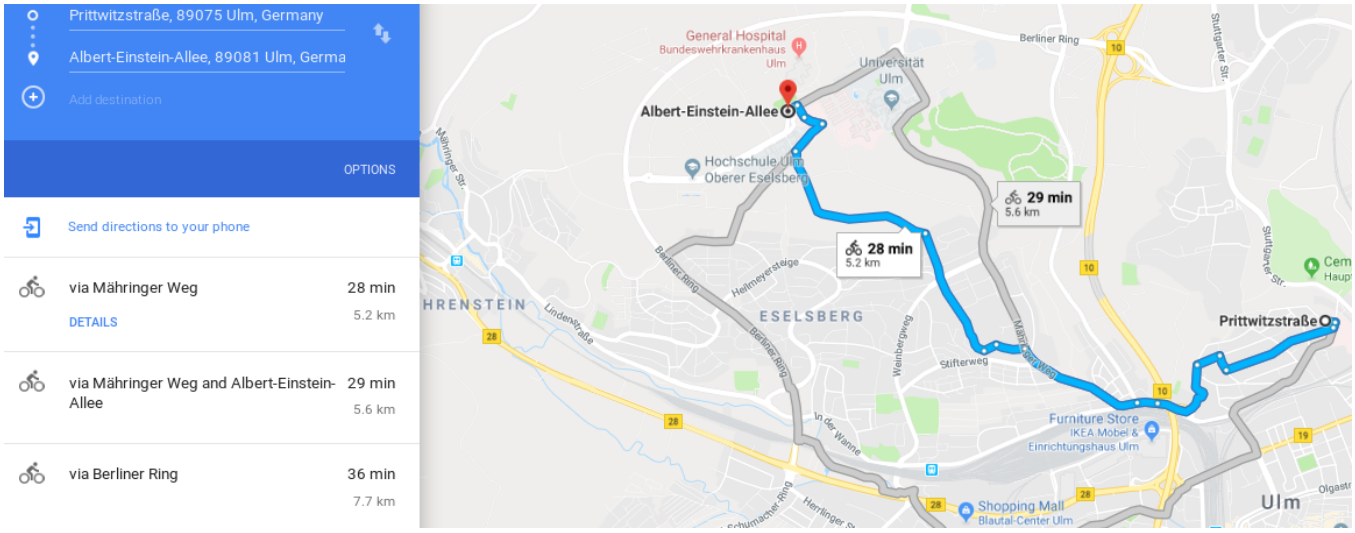

Figure 1.5: Distance shown in Google Maps

For some in-city routes, the library is pretty fast, but for longer routes (Stuttgart → Muenchen), **pyroutelib2** took nearly an hour. Consequently, this is not a possible solution to calculate all the distances between postcodes. We will then have to turn to another way (external to python) of calculating the routes for the project.

2 Graphhopper

Graphhopper is an open-source routing library and server written in Java which provides a web interface called Graphhopper Maps as well as a routing API over HTTP.

2.1 Deployment

For simplicity you could just start jetty from maven and schedule it as background job:

```
graphhopper.sh -a web -i europe_germany_berlin.pbf -d
    --port 11111
```

Then the service will be accessible on port 11111. For production usage you have a web service included. Use **-c config.yml** in the script to point to it.

Increase the **-Xmx/-Xms** values of your server server e.g. for world wide coverage with a hierarchical graph do the following before calling **graphhopper.sh**:

```
export JAVA_OPTS="-server -Xconcurrentio -Xmx17000m -Xms17000m"
```

Graphhopper is able to handle coverage for the whole OpenStreetMap road network. It needs approximately 22GB RAM for the import (CAR only) and 1 hour (plus 5h for contraction). If you can accept slower import times this can be reduced to 14GB RAM - you'll need to set **datareader.dataaccess=MMAP**. Then 'only' 15GB are necessary. Without contraction hierarchy this would be about 9GB. With CH the service is able to handle about 180 queries per second (from localhost to localhost this was 300qps). Measured for CAR routing, real world requests, at least 100km long, on alinux machine with 8 cores and 32GB, java 1.7.0_25, jetty 8.1.10 via the QueryTorture class (10 worker threads).
Especially for large heaps you should use **-XX:+UseG1GC**. Optionally add **-XX:MetaspaceSize=100M**. Avoid swapping e.g. on linux via **vm.swappiness=0** in **/etc/sysctl.conf**. If you want to use elevation data you need to increase the allowed number of open files. Under linux this works as follows:

- sudo vi /etc/security/limits.conf
- add: * - nofile 100000 which means set hard and soft limit of "number of open files" for all users to 100K
- sudo vi /etc/sysctl.conf
- add: fs.file-max = 90000
- reboot now (or sudo sysctl -p; and re-login)
- afterwards ulimit -Hn and ulimit -Sn should give you 100000

2.2 Usage

Our endpoint is **https://graphhopper.com/api/[version]/vrp**. Graphopper can provide up to 6 APIs:

- Routing API

- Route Optimization API
- Isochrone API
- Map Matching API
- Matrix API
- ZGeocoding API

Routing API

Get distances between two points:

```
curl "https://graphhopper.com/api/1/route?
point=51.131,12.414&point=48.224,3.867&vehicle=car&
    locale=de&key=[YOUR_KEY]"
```

Example in our case:

```
https://localhost:11111/1/route
?point=51.131,12.414&point=48.224,3.867&vehicle=car&
    locale=de&key=3160e710-58ed-45da-
bb39-09d383b1c5b2
```

Pricing

The Graphhopper routing engine is open source under the permissive Apache License and is therefore free to use for anything. You could even integrate it in your products, modify Graphhopper and sell this, without noticing or contributing back. Although it is encouraged to contribute back so that your feature gets maintained for free by graphhopper service. Also you can host Graphhopper on your own servers for 'free' and do whatever you want with it. The Graphhopper Directions API that we host falls under usage terms and always requires an API key. It was decided to make it free for development purposes and open source projects, both with a limit of currently 500 queries per day. So, the free usage of the API in a company internally would not be allowed. But there are custom packages possible.

One Routing API request costs one credit. Every 10 via-points cost one more credit. E.g. 11 via- points cost two credits, 21 via-points costs three

credits and so on. And if you specify **optimize=true** the credits will be multiplied by 10 i.e. one requests costs 10 credits for 1 to 10 locations, 20 credits for 11 to 20 locations and so on. Changing the parameter algorithm costs additionally two credits, i.e. calculating the alternative route between two points costs 1+2=3 credits. For more info about the other APIs: Graphhopper API Docs

3 Database

Since we want to compute distances between the centroids of two postalcode areas we need a database where those information can be stored.

3.1 SQL vs NOSql

First, it was necessary to find a suitable database. For this purpose we compared the Not only Sql (NoSQL) and the Structured Query Language (SQL) approaches with each other.
SQL happens to be the more structured, right way of storing data, like a phone book. For a relational database to be effective, you will have to store your data in a very organized fashion. SQL databases remain popular because they fit naturally into many vulnerable software stacks, including LAMP and Ruby-based stacks. These databases are widely supported and well understood, which could be a major plus point if you run into problems.
The main problem with SQL is scaling it as your database grows. You see, even though scalability is usually tested in production environments, it's often lower than NoSQL databases.

For dealing with massive amounts of unstructured data and data requirements which aren't clear at the outset, you probably don't have the luxury of developing a relational database with a clearly defined schema. You get much more flexibility than its traditional counterparts, with non- relational databases. Picture non-relational databases as file folders, assembling related information of all types.

Since we already know the scope of our database and therefore don‘t want to scale it up and due to the lack of time and the good knowledge of team members in SQL, we decided to make use of the relational database **MySQL**.

3.2 Installation

Since MySQL is the most popular open-source relational database management system, we decided to install this system on our server. First we installed the MySQL package with the following command:

```
sudo apt install mysql-server
```

Once the installation is completed, the MySQL service will start automatically. To check whether the MySQL server is running or not, we type:

```
sudo systemctl status mysql
```

After we have made sure that the database is running, we create a user account:

```
CREATE USER IF NOT EXISTS 'database_user'@'localhost'
    IDENTIFIED BY 'user_password';
```

Copy

Afterwards a new database could be created and initialized.

3.3 Datamodel

To implement an appropriate database which offers fast responses to queries it was indispensable to design a datamodel which fulfills these requirements. The figure 3.1 shows the output of the datamodelling process.

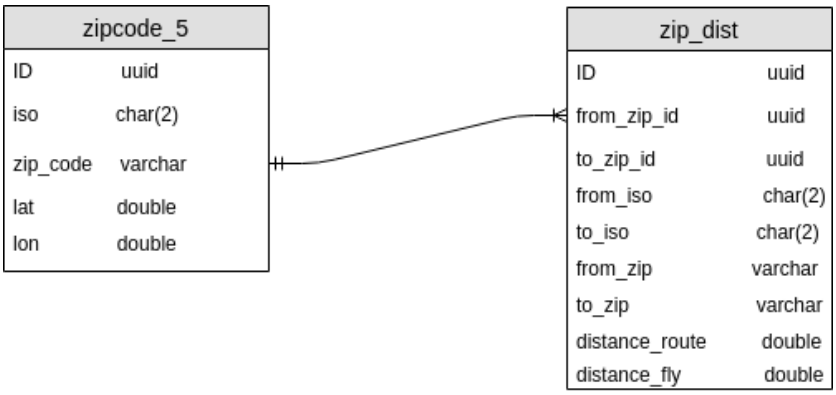

Figure 3.1: ER model for the MySQL Database

The model consists of two entities which are connected over a one-to-many relationship. The datamodels for the postcodes and the distances were directly written in the Django source code. That way, Django directly provides a python object to work with the tables. It is even possible to write methods to manipulate this data directly from it. The implementation is done in the file: model.py. For further information on working with the database in Django, see our corresponding GitHub repository.

3.4 Initialize Database

Geonames

To be able to fill the postalcode table, we need a list of all postcodes per country. The Nominatim API doesn't directly provide a function to do that, however the *GeoNames*, which is a geographical database, provides a list of all postalcodes which are free to download [8]. We implemented a first version of a script using this list and started filling the database. Geonames was incomplete, some countries (like Greece) were missing and the script had problems finding some cites. Therefore we had to find another, more reliable way to get the postcodes.

Nominatim Database

We decided to directly look into the Nominatim PostgreSQL database. There is in fact a table named **location_area** containing the column postcodes and the related **country_code** . Some rows in this table have empty **country_code** or **postcode**, so we need to filter them out. The resulting SQL query looks as followed:

```
1  SELECT DISTINCT country_code,postcode \
2    FROM location_area \
3    WHERE country_code IS NOT NULL \
4      AND postcode IS NOT NULL \
5    ORDER BY country_code, postcode;
```

Then, for every postcode, we need to query the Nominatim API to get the centroid coordinates, but querying only with the country code and the postcode is too ambiguous and can lead to wrong centroids. To fix that,

we used the **country_name** table that lists countries with names and ISO codes. We joined this table to the previous one:

```
SELECT DISTINCT location_area.country_code,postcode,country_name.name \
  FROM (location_area LEFT JOIN country_name \
    ON location_area.country_code = country_name.country_code) \
  WHERE location_area.country_code IS NOT NULL \
    AND postcode IS NOT NULL \
  ORDER BY country_code, postcode;
```

Final Script

To ensure that we don't left out any postal codes, we use both sources to fill the database. Using the resulting data, the script to fill the zipcode table was re-written, the database cleaned and filled with the new version of the script. The script can be found at the GitHub repository WebApp. To run the script, run the accompanying shell script **fill_db.py**, it will install the dependencies before running the python code.

4 Distance Database

4.1 Algorithm

If we computed the distance between all cities in Europe, it would take a very long time and file the hard drive consequently. To reduce the effort was decided to group each postcode by its region (most of the time represented by the two first digits). Then we compute the distance "as the crow flies" and by road between each of these regions and store them into the database.

When requesting the distance between two postcode, we process as such:

1. transform both postcode into coordinates
2. calculate the distance as the crow flies (named $d_{direct,crow}$)
3. determine the two 2-digit regions associated
4. get the distance "as the crow flies" ($d_{region,crow}$) and by road ($d_{region,road}$) between the two regions from the database
5. calculate the ratio $ratio = d_{region,road}/d_{region,crow}$

6. the final distance estimate is $d_{direct,crow} * ratio$

The calculation is summarized by figure 4.1.

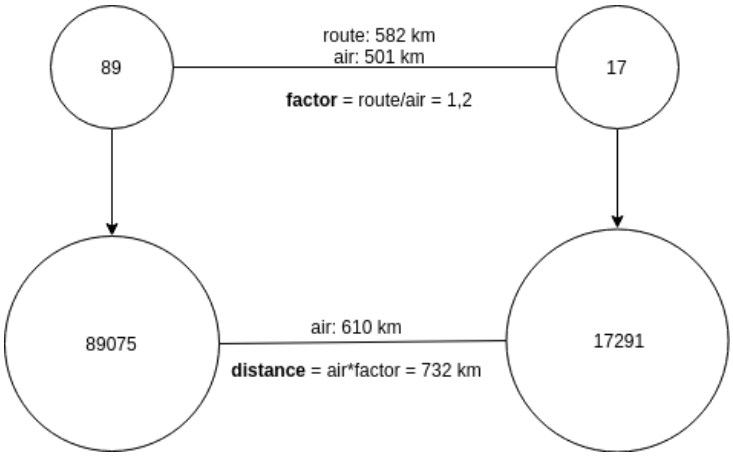

Figure 4.1: Theoretical example of distance calculation

4.2 2-digits Database

List of 2-digits Regions

The countries' regions being often indicated as the two first digits of the postcodes, we can simply create our groups based on that. However, we must take into account that there are some exceptions; for instance, Luxembourg prefixes its postcodes with "LE-". Some other countries also introduce non-alphanumerical characters, such as: spaces, hyphens, dots, …

We also restrict this operation to European countries only, we can do that by filtering postcodes by their coordinates: Europe being found between -11° West and 41° East, and 35° to 71° North.

Calculation of a Region's Centroid

To calculate the centroid for each region, we can compute the average coordinates of all postcodes of that region. However, we will have to use Graphhopper with these coordinates and Graphhopper can only work when coordinates point to a road/street/… To test if a coordinate is working for Graphhopper, we simply query it from our coordinate to an already checked place.

First, we test Graphhopper for the average centroid. If it can't be used, we go through the closest postalcodes to find one accepted by Graphhopper. Finally, if we still don't have a good point, we spiral around the average centroid until we find a good point or we reach 3 degrees radius (3 degrees in longitude is around 245km in Germany). We discard every other regions that fail these three tests.

Filling the Database

For each couple of region centroids calculated above, we calculate the distance "as the crow flies" thanks to the GeoPy function and the distance by road thanks to Graphhopper API and store those two results in the MySQL database.

The script can be found at the GitHub repository WebApp.

5 Django Web Application

The aim of this task was to develop a web application to provide distance calculation based on our Nominatim as well as our Graphhopper server. The calculation should be done by postalcodes as well as iso codes. As a result it should return the air distance as well as the distance by route of the related postalcodes.

5.1 Mockup

After the requirements were clear, we designed a mockup (see figure 5.1) in order to define the layout of the web application.

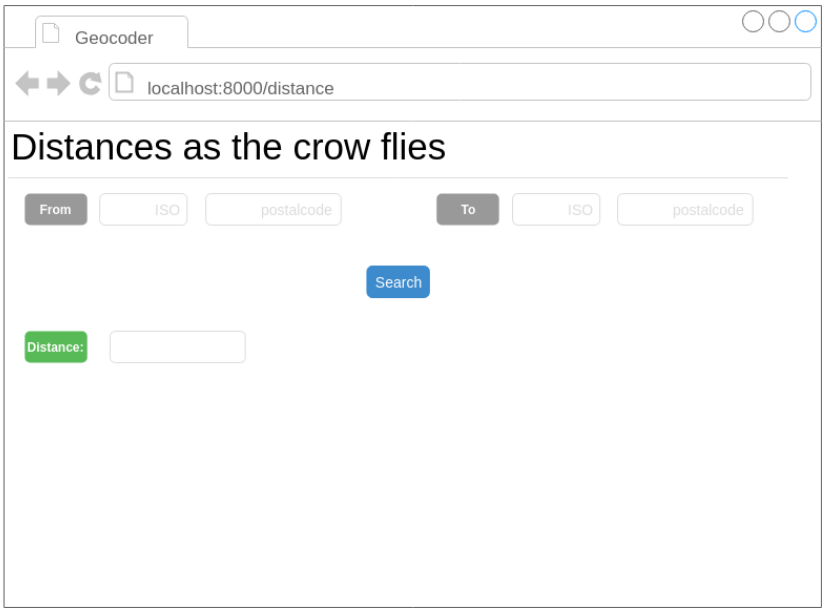

Figure 5.1: Mockup for Django web app

The mockups shows four input fields where the user can enter the postal-code and the corresponding iso code of his start and his starting point as well as his destination. The „Search“-button triggers the calculation and represents the result in the input field with the label „Distance“below. In this mockup only the distance calculation as the crow flies was taken into consideration. However in the final web application we included the distance calculation by route as well.

5.2 Implementation

For our frontend web application, we will use **Django** as it works in python and is the most used.

View

This mockup was then implemented as an HTML file. In order to get a quick result with a sophisticated and responsive design we decided to include the Bootstrap toolkit. This open source framework provides themes, templates and code snippets for a uniform layout and rapid processing. Therefore we

installed bootstrap on our Nominatim server. The view consist mainly of four parts:

1. index.html
2. style.css
3. urls.py
4. views.py
5. forms.py

The *index.html* file is responsible for the general structure of the web application. The *style.css* file is mainly responsible for the layout of the footer which is customized and not just adapted by bootstrap.
In the *urls.py* we defined which HTML file should be displayed for which url. Since the web application consist of one page only we set the index.html as the entry point by calling the web application.
Django provides a very secure form handling which is normally a very complex business including editing the form with a convenient interface, send it to the server, validate and clean the input and then save or pass it to further processing. With Django's form functionality all this work is automated and also do it more securely then we would probably be able to do. Therefore we created the file *forms.py* where we directly specified the input fields with their corresponding length as well as if they are required or not.

```
class CrowForm(forms.Form):
    from_zip = forms.CharField(max_length=5,required=
        True)
    from_iso = forms.CharField(max_length=2,required=
        True)
    to_zip = forms.CharField(max_length=5,required=
        True)
    to_iso = forms.CharField(max_length=2,required=
        True)
```

The input of this form is then processed within the *views.py* file, since here the post and get requests are handled. The *views.py* file is responsible to handle post ad get requests. This file calls also the responsible methods of the datamodel in order to calculate the distances. Moreover it returns

the results to the input fields „Direct“and „By route“which can be seen in figure 5.2.

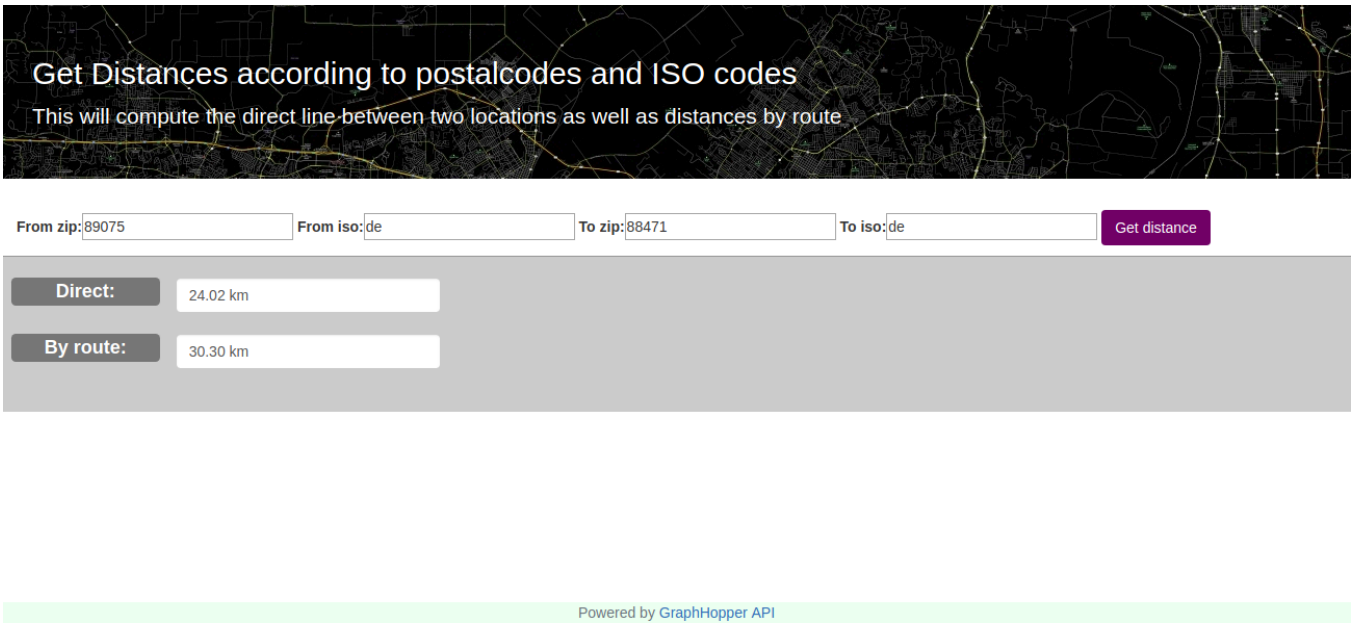

Figure 5.2: Final layout of the Django web application

Calculation

As described in Section 4.1, we first need to determine the corresponding region for the two given postal codes and country ISO codes and extract from the distance database the two distances to calculate the air/road ratio. We use Nominatim to convert the postcodes to coordinates to calculate their “as the crow flies” distance and we finally apply the regional ratio to it.

If the two postcodes are in the same region, we could apply an average ratio to correct a little the difference with the by road distance. We could also directly use Graphhopper again as these are two close points and shouldn’t require much resources to compute.

Chapter 3

Bike Rental in London

1 Evaluation of Hadoop Distributions

This chapter shows a general conducted comparison of Hadoop distributions which are dominating the big data market. Since it is necessary for our data science project to compute in a parallel and efficient fashion, such a comparison helps us to identify the right distribution. Firstly, an overview of the identified distributions with its features and pricing tiers will be showed. Afterwards we will declare some important criteria for our project based on an extensive research of internet blogs and manufacturer websites. Then a weighting of these criteria will be done. These criteria will be used to evaluate the different Hadoop distributions. The resulting evaluation matrix shows that Hortonworks might be the best distribution for our data science problem. However, a detailed look at the performance criterion shows us that MapR is the most most powerful in five5 out of six cases. For this performance comparison, evaluation results of various MapReduce jobs found on the Internet are used. Hence, in a pure performance comparison MapR would win over Hortonworks, but in an overall comparison Hortonworks convinces more. To round off the evaluation, a test implementation of Hortonworks was carried out on the four virtual machines procured for this purpose where some simple MapReduce jobs are carried out.

1.1 Hadoop Distributions Overview

For the first part of the evaluation, a search for widespread Hadoop distributions was conducted [23, 37, 17, 4, 19]. As the table 3.1 below shows, seven

noteworthy distributions were found.

Hadoop Distribution	**Features**	**Pricing Gear**	**Hadoop Ecosystem**
Apache Hadoop 3.1.1 hadoop.apache.org	• No extra tools • Blank installation of Apache Hadoop (HDFS, YARN and Hadoop Web UI)	Open Source 100%	Has to be installed manually [2]
Cloudera CDH (v5.15.0) www.cloudera.com	• Oldest distribution • Very polished • Comes with good (and proprietary) tools to install and manage a Hadoop cluster: – Cloudera Manager for managing and monitoring clusters [3]. – Cloudera Search (Free-Text) on Hadoop [5]. – Apache Crunch framework for MapReduce Pipelines [5].	Freemium (Cloudera Manager require license). Also source code is not fully available and enterprise edition has 60 trial-day [7]. Final costs may depend on cluster size [4].	Accumulo,Flume, HBase, HCatalog, Hive, HttpFS, HUE, Impala, Kafka, KMS, Mahout, Oozie, Pig, Sentry, Snappy, Spark, Sqoop, Parquet, Whirr, ZooKeeper [3, 5]

Hadoop Distribution	**Features**	**Pricing Gear**	**Hadoop Ecosystem**
Hortonworks Data Platform (HDP3.0.1) www.hortonworks.com	• Newer distributions • Tracks Apache Hadoop closely • Comes with standard and open source tools for managing clusters: – Improved HDFS in HDP 2.0: automated fail over with a hot standby and full stack resiliency [10]. – Ambari for managing and monitoring clusters. – Includes almost all Hadoop extensions from the Apache foundation [10].	Open Source, optional enterprise paid support [7]	Atlas, HBase, HDFS, Hive Metastore, HiveServer2, Hue, Spark, Kafka, Knox, Oozie, Ranger, Storm, WebHCat, YARN, SmartSense, ZooKeeper [11]
MapR 6.1 www.mapr.com	• Uses own HDFS (MapR-FS) • Integrates own database systems (MapR-DB seven times faster than HBase) • Mostly used for big big data projects • Free from Single Point of Failures • Offers Mirroring and Snapshotting • Might be the fastest Hadoop distribution • MapR supports backward compatibility across multiple version of projects • Supports a global event replication for streaming at IoT scale (MapR-ES)	Freemium [21]:Community Edition contains no high availability, no disaster recovery and no global replication for streaming data.	AsyncHBase, Cascading, Drill, Flume, HBase Client and MapR Database, Binary Tables, Spark, HCatalog, Hive, HttpFS, Hue, Impala MapR Event Store For Apache Kafka Clients and Tools, Myriad, OpenStack, Manila, Oozie, Pig, Sentry, Spark, Sqoop, MapR Object Store with S3- Compatible API[20]

Hadoop Distribution	**Features**	**Pricing Gear**	**Hadoop Ecosystem**
Intel www.hadoop.intel.com	• Partnered with Cloudera [24] • Encryption support • Hardware acceleration added to layers of stack to boost performance • Admin tools to deploy and manage Hadoop	Premium (90 days trial)	Offers same services as Cloudera.
Pivotal HD www.gopivotal.com	• Partnered with Hortonworks [24] • Fast SQL on Hadoop • Proprietary Software [32]: – Pivotal DataLoader – USS (external file system) – Spring Data – Pivotal ADS-HAWQ (parallel SQL query engine)	Premium	Offers same services as Hortonworks.
IBM Open Platform www.ibm.com	• Partnered with Hortonworks [24] • Proprietary tools: – IBM Big SQL allows concurrent process of Hive, HBase and Spark and other sources using a single database connection [24]. – IBM BigInsights v4.2 provides Text-Analytics module [15]. • Highly compatible to other IB products.	Free for non-commercial purposes, optional enterprise paid support [16]	Offers same services as Hortonworks.

Table 3.1: Overview of Hadoop distributions

It is worth to mention that each Hadoop distribution in table 3.1 comes up with a minimum of services from the Apache foundation: Yet Another Resource Negotiator (YARN), Hadoop Distributed File System (HDFS), Hive, Pig, HBase, Kafka, Storm, Mahout, HCatalog... But they may use a proprietary implementation of them e.g. MapR uses own HDFS instead of Apache HDFS implementation. It is also worth to note that a few cloud providers are offering Hadoop cluster services over their platforms. For instance, Microsoft Azure provides with HDInsight a full manageable Hadoop respectively Spark Cluster [25]. The user could profit from fast provisioning and also from less costs since no on-prem hardware cluster infrastructure is required. Therefore, Hadoop (or better Spark) by Cloud Computing is also a noteworthy option for our data science project in the future, although it may be over sized at the moment. It is worth to mention that each Hadoop distribution in table 3.1 comes up with a minimum of services from the Apache foundation: YARN, HDFS, Hive, Pig, HBase, Kafka, Storm, Mahout, HCatalog etc. But they may use a proprietary implementation of them e.g. MapR uses own HDFS instead of Apache HDFS implementation. It is also worth to note that a few cloud providers are offering Hadoop cluster services over their platforms. For instance, Microsoft Azure provides with HDInsight a full manageable Hadoop respectively Spark Cluster [25]. The user could profit from fast provisioning and also from less costs since no on-prem hardware cluster infrastructure is required. Therefore, Hadoop (or better Spark) by Cloud Computing is also a noteworthy option for our data science project in the future, although it may be over sized at the moment.

Nevertheless, Intel, ERM/Pivotal and IBM partnered with Hortonworks or Cloudera 3.1. These vendors therefore have the same extensions as their partners offer, but also can provide exclusive tools for users (e.g. IBM Big SQL [24]). If we take this into account, we find that there are mainly 3 different Hadoop distributions: Cloudera with CDH, Hortonworks with its Hortonworks Data Platform (HDP) and MapR with its own HDFS. In addition to the distributions shown in 3.1, there are many other Hadoop distributions such as Altiscale from SAP or Elastic MapReduce from Amazon. Both run only in a cloud environment. The Forrester Wave shows also the three global players Cloudera, Hortonworks and MapR (see 1.1). Since cloud solutions do not play a role for this use case, Microsoft, Google or Amazon will not be considered.

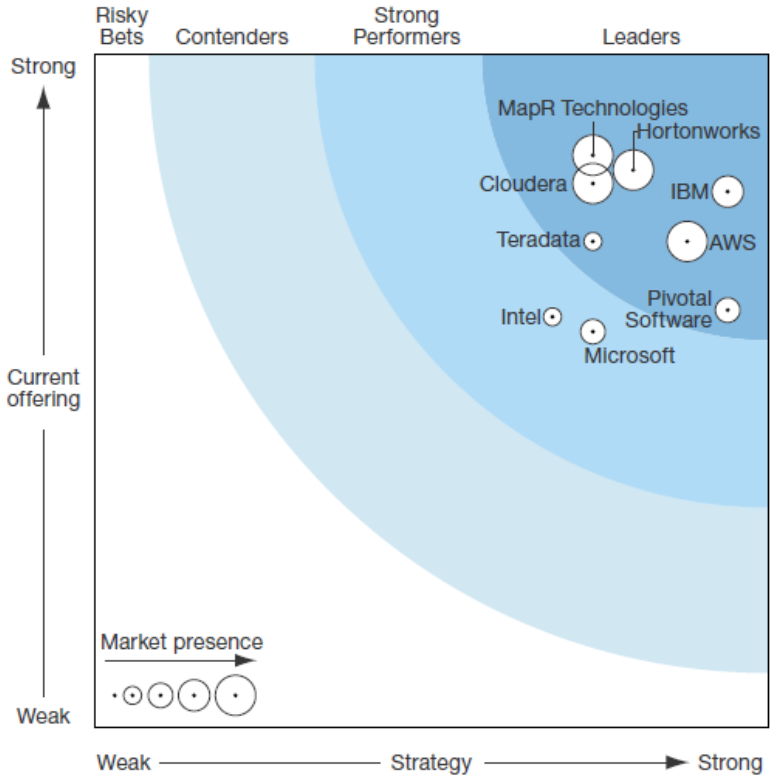

Figure 1.1: Forrester Wave of Hadoop Distributions [37]

As we can see from 1.1 the three mentioned Hadoop distributions are building a cluster and Cloudera even overlaps with MapR slightly. That means, they are providing similar products and enjoy a similar market position. From the Forrester Wave (1.1) we can derive that MapR seems to have the best offering. However, since we want to use one of them for our analytics project we have to dig a step deeper and make a decision based on well-defined criteria.

1.2 Definition of criteria

Hadoop distributions are rated using selected criteria, with scores ranging from 1 (very poor) to 5 (very good). Based on the research carried out, the following 20 clear criteria can be determined:

Criterion	Description
Batch Processing	Batch process jobs can run without any end-user interaction or can be scheduled to start up on their own as resources permit
Cloud Support	Cloud compatibility of Hadoop distributions
Cluster Scalability	Creating new clusters ad-hoc and set up cluster size (e.g. 4,8 or 12 nodes)
Data querying possibilities	Ways to query Hadoop with SQL (e.g. Hive, Stinger, Spark SQL,...)
Database Support	Additional databases in Hadoop (e.g. PostgreSQL, MongoDB, HBase, Impala etc.)
Disaster-Discovery	There is a fallback option available in case of unexpected Hadoop errors
Ease of Use	The complexity of the usage of the Hadoop distribution
High-Availability	Self-healing across multiple services or single failure recovery (i.e. fail-over clusters)
Install complexity	Is there a simple installation routine or does it require lot of manual configuration (e.g. Multi node Cluster...)?
Licensing and Pricing	Cost model of Hadoop distributions
ML support	Providing interface for enlarged machine learning tasks (e.g. Spark MLlib...)
Operating System Support	Guaranteed and certified OS compatibility
Performance	Cluster performance in case of parallelized MapReduce Jobs
Programming Language Support	Support of data science programming languages (Python and R)
Streaming Analytics	Vendor offering real-time analytics capabilities (e.g. IoT platform hub..)
Support of secondary services	Services on top of Hadoop (e.g. Zookeeper, Spark...)
Third party module integration	Usage of additional components from other vendors
User Support/Community	How fast is the response in the community?
Expertise	Experience with Hadoop (life time of company, ...)

Table 3.2: Definition of criteria

The weight scale ranges from 1 (trivial) to 4 (very important), with the focus on our Santander Bicycle project. This means that cloud support,

for example, is a crucial criterion in many use cases, but has only little importance for our project which leads to a trivial weight. A percentage weighting is not applied because 20 criteria would result in a fine-granular distribution (which is not good to read at all). Therefore, absolute values are used for the comparison 1.2 in the next section.

1.3 Weighted evaluation matrix

As already mentioned in Section 1.1, the three major Hadoop distributions are: Cloudera CDH 5.15.0, Hortonworks HDP 3.0.1 and MapR 6.1.0. With regard to the defined comparison criteria, a weighted decision matrix can be mapped. At the same time, this matrix represents the starting point for the decision of a Hadoop distribution. It cannot be denied that a certain degree of subjectivity is included in the evaluation (table 3.2). In addition, only free editions are compared. For example, if a feature only exists in the premium version, this feature will be rated one (worst) since we don't want to spend money.

Criteria	Weight	KO	MapR	Hortonworks	Cloudera CDH
Batch Processing	3		5	5	5
Cloud support	1		3	4	5
Cluster Management Complexity	4		3	3	4
Cluster Scalability	2		4	4	3
Data querying possibilities	3		4	4	5
Database Support	4		5	3	5
Disaster-Discovery	2		1	4	1
Ease of Use	2		3	3	4
High-Availbility	2		1	4	5
Install complexity	2		3	2	3
Licensing and Pricing	2		1	5	2
ML support	4	x	5	5	5
Opearting System Support	3	x	2	3	2
Performance	4	x	5	4	4
Programming language support	3	x	5	5	5
Streaming Analytics	1		5	5	5
Support of secondary services	3		3	5	4
Third party module integration	2		2	4	1
User Support & Community	2		2	4	3
Expertise	2		4	5	3
Total Score			179	**209**	199

Legend:
(bad) 1
(is okay) 2
(mediocre) 3
(good) 4
(ideal) 5

Figure 1.2: Weighted Comparison Table of Hadoop Distributions

The comparison in figure 1.2 clearly shows that there are almost no significant differences between the selected distributions. Although there are a few deviations such as disaster-discovery or pricing model, the overall distribution of points is relatively the same. From the comparison matrix it can be deduced that Hortonworks HDP achieves the highest score and is probably the best option for our project at the moment. It is possible that after the first test phase (see chapter 1.5), it turns out that the distribution is not convenient after all which may require to update the evaluation matrix. Therefore figure 1.2 can be considered as a continuous iterative updateable weighted comparison matrix that will likely be updated on further sprints. Another representation of the evaluated Hadoop distributions, a spider chart may be appropriate as it makes it easier to read and to identify outliers or

similarities even quicker as with a raw table.

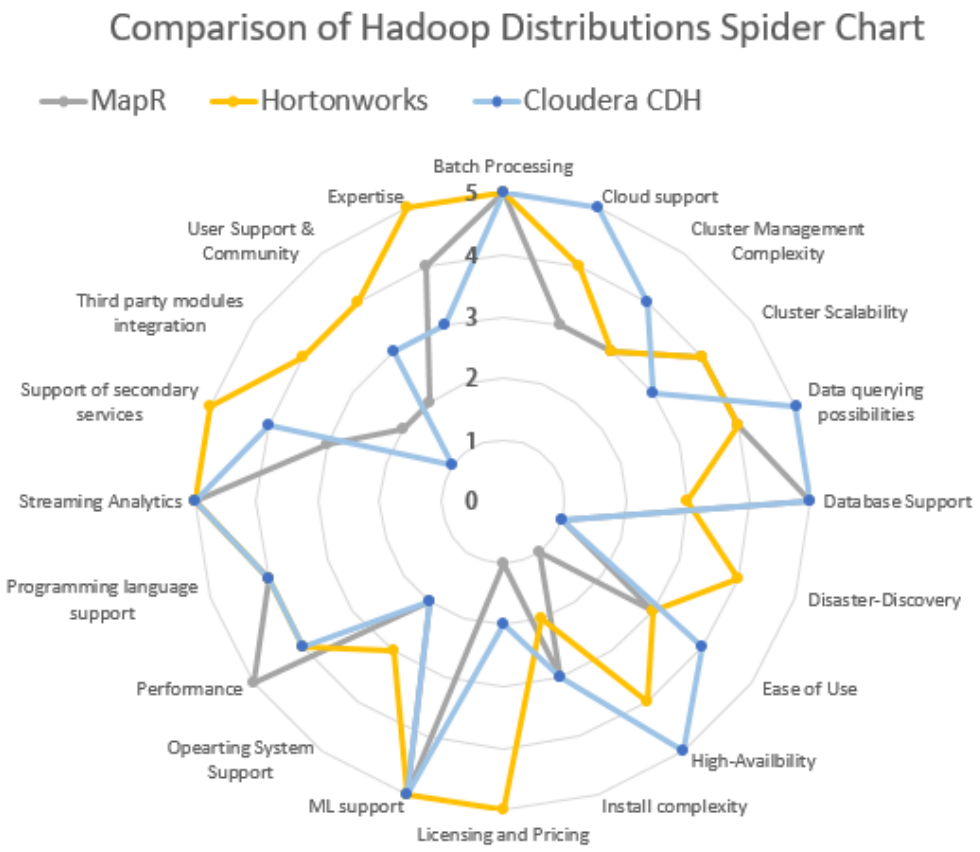

Figure 1.3: Spider Chart of compared Hadoop Distributions

The KO-criteria from figure 1.2 are all satisfied by the selected Hadoop vendors. One criterion that stands out is the license model, which was rated with 1 point for MapR and 2 points for Cloudera. This is due to the fact that the price models of both vendors are not transparent. So Cloudera offers a free community edition but to use the Cloudera Manager (the real strength of Cloudera) you have to pay again. Only Hortonworks offers a complete Open Source package with except the Hortonworks business support is fee required. However, Hortonworks has a very active community as well. MapR comes off worst with the criterion “Support of secondary services”, because there is no Spark and HBase support in the ”Converged Community Edition” [21], but both are important Hadoop components for our project. So if we wanted

to use MapR, we would have to use the paid version. There are also some other differences in figure 1.3 such as Disaster-Recovery which is only on Hortonworks's HDP completely free. An interesting aspect of the comparison is the performance criterion where MapR has the most points. Since this criterion is a KO-criterion that means it is a very important component for our project, it makes sense to examine the performance comparison between the distributions in a more detailed fashion.

1.4 Performance Comparison with micro benchmarks

For a performance comparison it is good to know how fast they compute when running in a concurrent mode. For this task MapReduce Jobs like WordCount or Data File System I/O (DFSIO) Read/Write might be helpful. Following figures are extracted from a sophisticated evaluation work by Altoros [1].

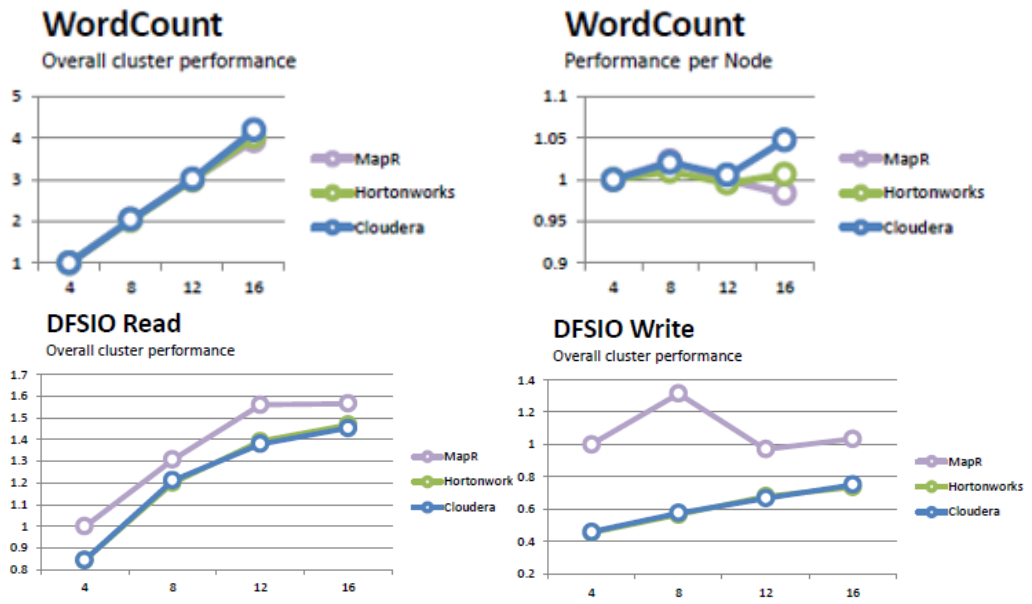

Figure 1.4: Micro benchmarks on DFSIO / WordCount MapReduce Jobs

For the performance comparison of figure 1.4 Hadoop cluster each one with 16 nodes have been established in order to measure computation time. An interesting point is that all three Hadoop distributions almost achieve the same overall speed on the famous WordCount MapReduce job. But if we

look at the performance per node then there is a small difference between them. This could be due to a certain error rate in the execution of the job. If the same test were repeated, the results would be negligibly different. The DFSIORead/Write (figure 1.4) job shows MapR is definitely faster than Hortonworks and Cloudera. The reason for this might be the fact that MapR uses its own HDFS which is according to the vendor seven times faster than the original one from Apache. This is the reason why it gets five points in the evaluation matrix (see table 3.2) whereas other ones only reach four points. Hortonworks and Cloudera seems to have the same performance on the DFSIO job because both using Apache HDFS.
In overall, Hortonworks HDP wins the competition for the moment, because they have the longest experience and are completely compatible with secondary services. Based on the evaluation matrix from chapter 1.3 and the performance measurements, it can be deduced that MapR is the most powerful Hadoop distribution on the market today, but when considering the other criteria, Hortonwork's Hadoop is simply more convincing. Especially the complete Open Source guarantee at Hortonworks is a decisive criterion for the choice of this distribution.
Beside from the comparison, we would generally recommend to use rather Spark than Hadoop since Spark is around 100 times faster due to in-memory processing. Also Spark provides the most important libraries (ML, Streaming, etc) for data science and works well on top of Hadoop (thanks YARN). It can even be installed in a standalone manner, even though it doesn't benefit from distributive multi mode computing.

1.5 Installation of sample Hadoop Distribution

Hortonworks offers a configurator on their website that can be used to check products, operating systems, databases, browsers, JDKs and supported processors. It is noticeable that Ubuntu is only supported up to 16.04 LTS. However, our VMs are already v.18.4 LTS. This means Hortonworks does not official support our installed OS [12]. Also Cloudera CDH 5.15 [6] and the newest MapR 6.1 distribution [22] only support Ubuntu 16.04 LTS (Xenial). Anyway, it is still a worth a try to set up HDP Cluster on our virtual machines since the vendors are not explicitly warning Ubuntu 18.04 is not supported. So it may work. For the installation of HDP 3.0.1 the Ambari Wizard [9] will be used.
First of all, we have to configure the **etc/hosts** since every node in the clus-

ter must be able to communicate with each other. The hosts file should look like this:

```
1  127.0.0.1       localhost
2
3
4  141.59.29.111 i-hadoop-01.informatik.hs-ulm.de i-hadoop-01
5  141.59.29.112 i-hadoop-02.informatik.hs-ulm.de i-hadoop-02
6  141.59.29.113 i-hadoop-03.informatik.hs-ulm.de i-hadoop-03
7  141.59.29.114 i-hadoop-04.informatik.hs-ulm.de i-hadoop-04
```

Figure 1.5: etc/hosts configuration file

After that step, it is necessary to add public key authentication with SSH. The master node (**i-hadoop-01**) should login to its worker nodes without using a password. We can achieve this goal by generating a private / public key pair and distribute the public key to the 3 worker nodes by using **ssh-copy-id** command. Also for using Ambari the Hadoop user need sudo execution without password. Adding an additional line to **visudo** configuration file should solve that issue. At next, all nodes need to have Java installed. Since Ubuntu 18.04 doesn't come up with a default Java JDK we install Oracle JDK 1.8 manually on each node. After that it is time to download the Ambari repository file to a directory on our Hadoop master host. The Ambari v. 2.7.1.0 is used for installation. With the command **apt-get install ambari server** the server will be installed on i-hadoop-01. Afterwards, we can start the server. Now we can go to the Ambari surface via following link: http://i-hadoop-01.informatik.hs-ulm.de:8080. The default login credentials are **admin admin**. After successful login, the real installation process begins.

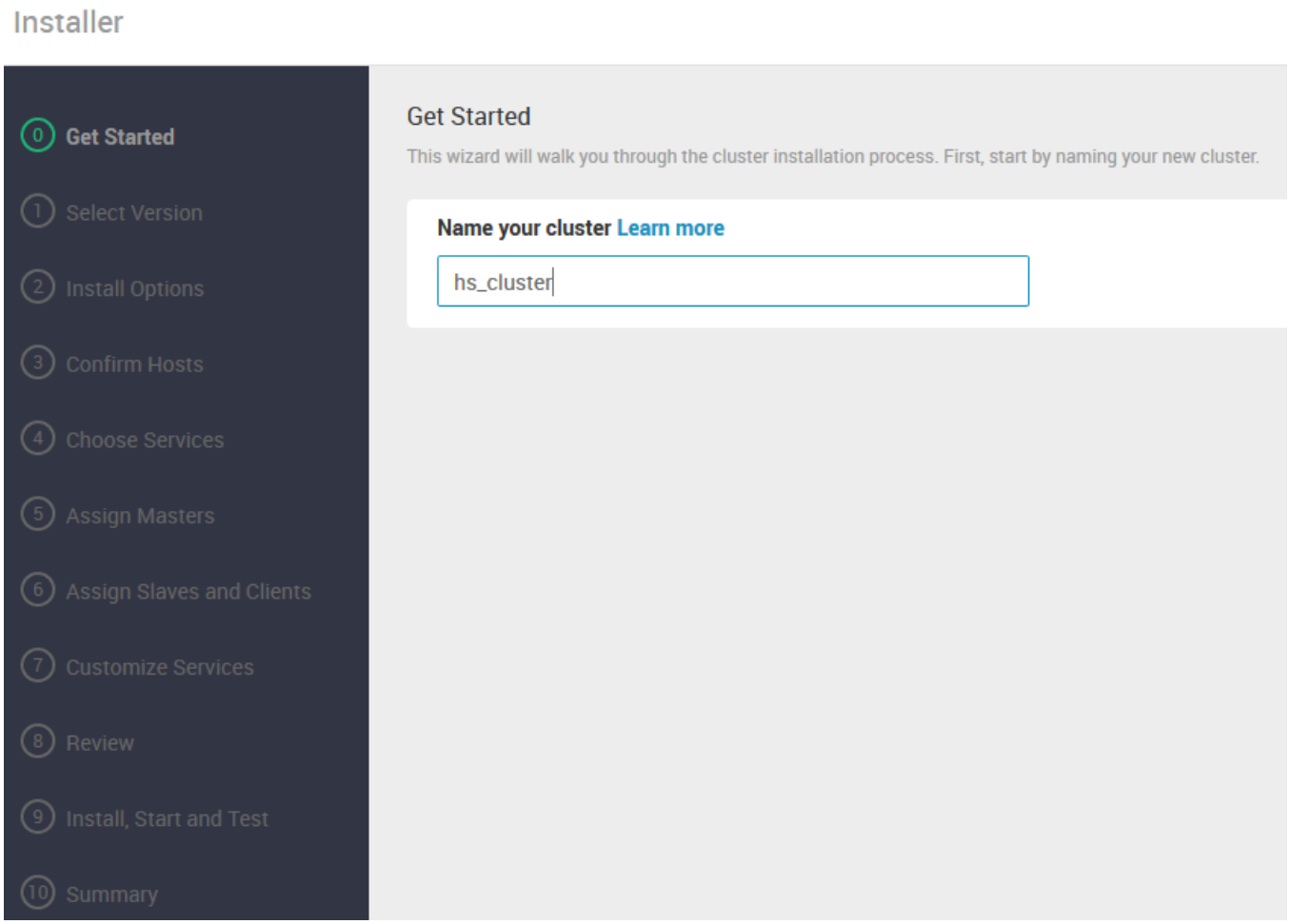

Figure 1.6: Ambari Wizard Installer

Next, we click through the installation routine until the point appears where the cluster nodes are defined. At this point the fact that we are using Ubuntu 18.04 will cause some troubles. If we adding the hosts Ambari is trying to check some dependencies and is running also some check routines. It will fail to add the worker nodes to the cluster since OS is not supported (termination condition). The solution to this problem would be a downgrade of the OS but we don't have the privileges to do this. So we manually changed the **/etc/issue**, **etc/lsb-release** and **etc/os-release** file to Ubuntu 16.04 version. Of course, a backup of the original ones has been created as well. After this workaround, the check condition will not fail because Ambari

consider our OS as Ubuntu 16.04. Obviously, that's a dangerous operation, so we change it back to original state after installation has been completed. In the next step we are choosing our Hadoop services that may be relevant for our data science project. For a start we choose following services: HDFS, YARN, MapReduce2, Tez, Hive, ZooKeeper, Ambari Metrics, SmartSense, Spark2 and Zeppelin Notebook. Interesting to see is that the proprietary service SmartSense is the only one that cannot be deselected. We have to install it, even if we don't want to use it. That's certainly not very user-friendly. However, it is simple to add further Apache services on running Ambari but more difficult to uninstall them so we am do not enable all possible Hadoop services from the beginning. In addition, some services are not for interest at the moment, hence they would only consume storage without having a practical effect.
After some additional configuration steps we can choose which services should run on master or worker nodes. Also we have to decide which node should be data node and node manager. In this cluster all four VMs are data nodes as well as node managers. A database connection is also mandatory at this step. Depending on selected services there are several database connections. For instance, the service Hive needs a working SQL database connection. We can choose between MySQL (MariaDB), PostgreSQL and Oracle. For a first test, MySQL is used as Hive database. We can change the database connection setting on Ambari at any time. If the configuration has been finished, a summary with all decisions taken is shown. That's the last possibility to change a fundamental Hadoop setting. It is worth to check all configurations carefully. Afterwards there is no go back option and the uninstall routine of Ambari is complicated and might lead to tremendous work of removing operations. That is also negative criteria of HDP and Ambari. As soon as the installation completes we can switch to the Ambari monitor. Unfortunately, almost all selected services were offline or could not start properly. A deeper investigation showed that there were some permission problems with HDFS and the Hadoop user. The HDFS folder was only for root user but not for user „Hadoop". With the command **chown -R hdfs:hadoop /hadoop/hdfs** the correct permissions could be granted.
Another problem was that YARN could not start the name nodes due to connection refused error messages. As **netstat -tupln | grep 50070** shown the service was only running for localhost address. After changing the binding address to **0.0.0.0** on HDFS, the communication between the cluster nodes was working and the name nodes could properly start. However, after fixing

this issue, the most services were running. The community of Hortonworks was quite helpful in this case. We got an answer within four hours whereas for getting an answer in Cloudera's community took three times longer.
The data node of the master node was not running correctly. It could be started from Ambari and after few seconds the data node went offline. A look at the error log of this node showed that the cluster id was already used by another node. Removing the HDFS data folder of the data node solved this issue because afterwards it generates a new cache file with a new cluster id.
The following screenshot shows a (successfully) completed Ambari and HDP installation on our VMs.

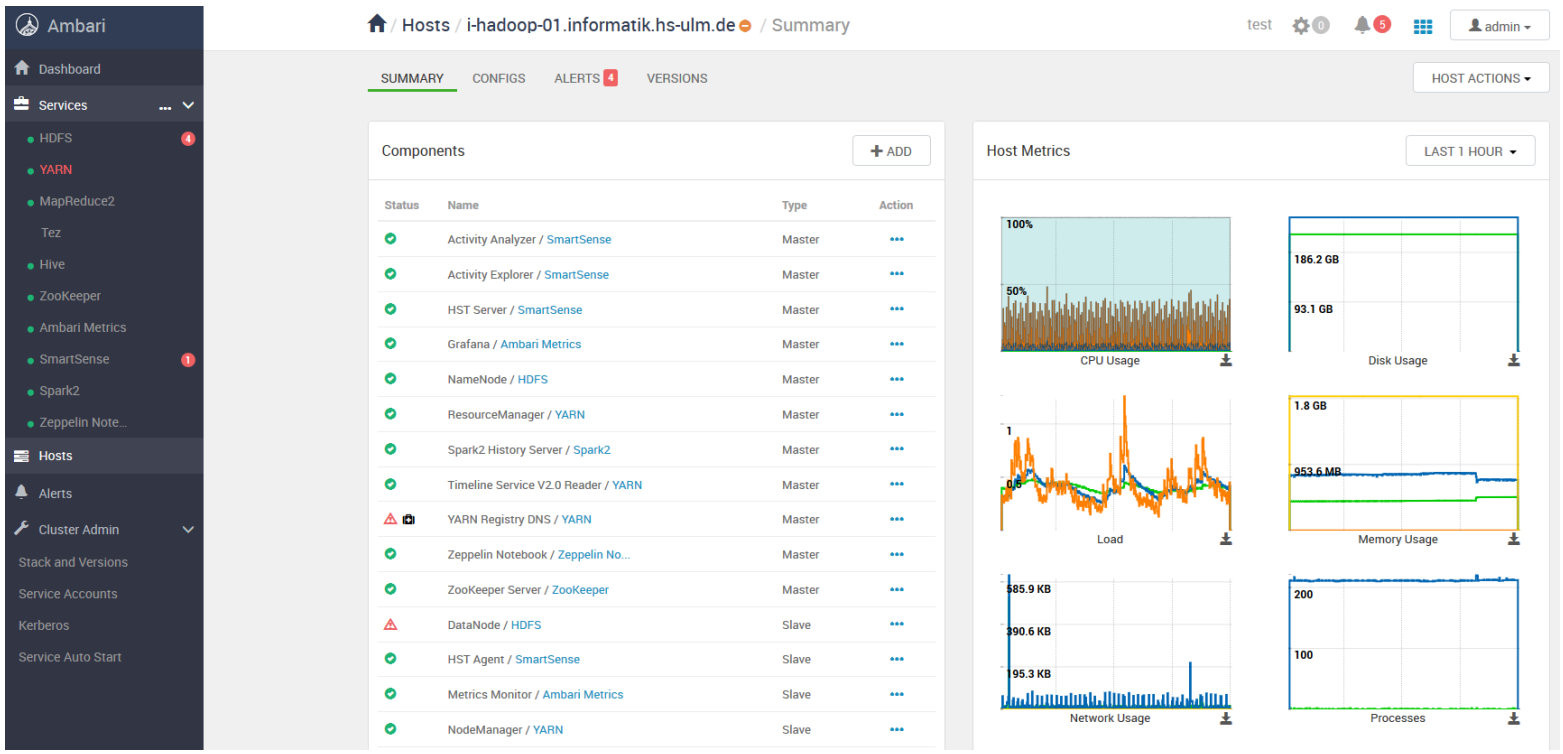

Figure 1.7: Ambari Surface, Host metrics on master node

Ambari has been set up and most Hadoop services are running (see Figure 1.7). Ambari shows lot of statistics that might be helpful for cluster management. We can control each service and either turn it off or set it into a maintenance mode. The ladder one encapsulates the service without influenc-

ing other services (high availability). Overall, Ambari is a powerful Hadoop cluster management tool that has lot of control options and supervising tools but also lacks in removing services and usability (e.g. no Ubuntu 18.04 support, only Python 2.7.x support, etc). The fact, that there is no official uninstaller from Hortonworks makes HDP with Ambari a risky installation. On the other hand, it is completely open source and has a strong community. Furthermore, HDP 3.0.1 comes up with a bunch of useful Hadoop services and well documentation so that we will likely stay on HDP.
Now it is time to run some MapReduce jobs on our newly created Hadoop cluster.

1.6 Run MapReduce Jobs

With four workers (master is also a worker node) a significant speed advantage over single computing should be achieved. First, with a MapReduce job, the accuracy of the decimal places of PI is determined using the quasi Monte Carlo method. This MapReduce job is also offered by the Apache foundation (Source). In the console of the master node the MapReduce job can be started with YARN as follows:

```
yarn jar /usr/local/hadoop/share/hadoop/mapreduce/
    hadoop-mapreduce-examples-
3.1.1.jar pi 16 1000
```

The *PI job* calculates 16 maps where each map process contains 1000 samples. After a short time, the job is finished and the following output is achieved:

```
Job Finished in 39.463 seconds
Estimated value of Pi is 3.14250000000000000000
```

Figure 1.8: PI Quasi Monte Carlo method executed as Cluster job

The same test has been performed with two worker nodes (i.e. two of the four workers were temporarily disabled via Ambari). After that the job took much longer (about 56 seconds). This shows that the Hadoop cluster works and already has a performance advantage over single computing. The next test was a simple WordCount job in Java that calculates the frequency of unique words in an input file distributed across the cluster. A MapReduce job consists of at least a mapper method, a reducer method and a driver method which is often the start method. Before executing the new job one

has to put the file of interest on Hadoop's HDFS. This is simply done as shown below:

```
hadoop fs -put big.txt
```

The put command will upload the file into the HDFS (i.e. it is stored fault-tolerant and replicated 3 times). After that, the following command runs a WordCount job on this text file:

```
hadoop jar hadoop-mapreduce-examples-3.1.1.jar
    wordcount wordcountFile big.txt
```

Note that now „Hadoop"is used, which starts also the YARN service. There is no difference between executing this with the Hadoop or YARN command. (Although latter is newer and should be used). The output of the „reduced"file looks like figure 1.9.

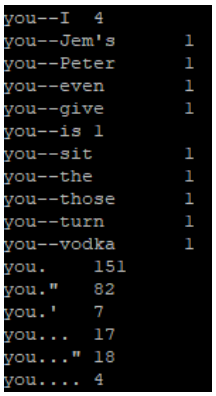

Figure 1.9: Output of WordCount job

Quick to recognize that the words were separated and counted by spaces. In fact, the input text was split afterwards and each word was mapped as "key value pair", where the key in this case is the word. The original text file was 6.4 MB, while the newly created file is only 0.93 MB in size. This shows another potential benefit of the cluster, namely the possible separation of files to minimize the total size of the initial data set. The running time was around 10 seconds. The same job has been executed locally on a commodity Windows client in NetBeans as well and it took around 8 minutes. There is a plenty of further typical MapReduce jobs (e.g. terasort, randomwriter, sudoko, etc), but for an evaluation these two jobs should already show that the cluster is working apparently faster than a single node.

1.7 HDP Issues

Due to the workaround in Chapter 1.5, a few services could not start properly. Also, permissions for some subfolders were set incorrectly, hence permission denied errors occurred when running the services. During the test phase some other problems were discovered, which are described below. Table 3.3 also shows the solution that was found to resolve these issues.

Issue	Solution
No starting NameNodes, Connection refused.	Go to HDFS→ Filter at https 0.0.0.0 and localhost address on master node http 127.0.0.0:50070 → 0.0.0.0:50070 Hive could not start on Node 2 since database was missing (mysql) Permission on HDFS was set to root and not to user hadoop
Wrong Permissions on HDFS	tail hadoop-hdfs-namenode chown -R hdfs:hadoop /hadoop/hdfs
Hive Database not found on i-hadoop-02	• Start mysql • Show databases; • create database hive; • CREATE USER 'hive'@'localhost' IDENTIFIED BY 'hive'; • GRANT ALL PRIVILEGES ON . TO 'hive'@'localhost'; • CREATE USER 'hive'@'%' IDENTIFIED BY 'hive'; • GRANT ALL PRIVILEGES ON . TO 'hive'@'%'; • GRANT ALL PRIVILEGES ON . TO 'hive'@'localhost' WITH GRANT OPTION; • GRANT ALL PRIVILEGES ON . TO 'hive'@'%' WITH GRANT OPTION; • FLUSH PRIVILEGES;
Bind address in mysql-conf (0.0.0.0):	• netstat -tupln | grep 3306 • Nano /etc/mysql/mysql.conf • Change bind-address to 0.0.0.0 • Systemctl mysql restart • Service mysql restart
YARN DNS Server not starting, connection refused	• Change Port 53 to other Port • Set right permissions on YARN folder

Issue	Solution
Master Node worker is not starting (Conflicting cluster id)	• Stop service • sudo wget archive.apache.org/dist/zeppelin/zeppelin-0.8.0/zeppelin-0.8.0-bin-all.tgz • sudo tar -xf zeppelin-0.8.0-bin-all.tgz • Move interpreter to /usr/hdp/current/zeppelin-server/interpreter/ • sudo chown -R zeppelin:zeppelin/usr/hdp/current/zeppelin-server/interpreter/ • Start service
Umask is changing from 0022 to 0002	• sudo gedit /.bashrc • Add line umask 022 • Logout and login
FAILED [2018-12-0813:36:00.609] java.io.IOException: mkdir of /hadoop/-yarn/local/file-cache/60_tmp failed	• YARN permissions • sudo chown yarn yarn • sudo chmod 750 yarn

Table 3.3: Problems and solutions of HDP

The described problems and their solutions of table 3.3 show that many setting options should be checked carefully during the Ambari installation routine (e.g. if a MySQL database exists). Furthermore, it can be assumed that most of these issues can be explained by the „officially“unsupported operating system version. However, after the fix, all Hadoop services in Ambari had a green status point.

1.8 Recommendation

The evaluation showed that Hortonworks with the Hadoop distribution HDP is best suited for our goals. Ambari provides an efficient, comprehensive clus-

ter management tool for monitoring and controlling the Hadoop cluster. In addition, an active community and complete open source freedom can convince more than its competitors. Although all three Hadoop distributions may have their individual advantages and disadvantages (see evaluation matrix in figure 1.2), HortonWorks was able to convince us due to the reasonably good documentation and the easy to follow Ambari installation routine. It should be noted, however, that the Cloudera Manager installation wizard also has good documentation and a comparable installation effort. However, we liked the Ambari interface more than the CDH one. In addition, Cloudera's pricing is not entirely transparent and third-party module integration is rather difficult. For example, since CDH 5.11 there has been a proprietary „data science workbench"service replacing Jupyter and IPython notebooks. HortonWorks, on the other hand, offers Zeppelin notebooks that resemble the famous Jupyter notebooks. Apart from that, Jupyter could also be installed as a third module during the test phase of HDP, even though such installations are naturally risky and are not officially supported by HortonWorks.
It is interesting that MapR, HortonWorks and Cloudera offer Docker images for immediate testing. For example, MapR offers a VMware image free of charge. However, all these packages have the disadvantage that they only have a single node cluster and limited functionality. For example, I could not select any services or create new ones in MapR and the function to add more worker nodes to the existing cluster was also not available. Therefore, such images or sandboxes are ideal for a quick check of the user interface and some standard functions, but rather insufficient for a complete range of functions or a productive system.
In conclusion it can be said that all three Hadoop distributions would meet the requirements of a big data project or our project perfectly, but HortonWorks could simply convince more with the overall score than the more commercially oriented counterparts. Ultimately, the performance also depends on the hardware of the individual cluster nodes. Due to a low physical RAM, MapReduce jobs even with MapR's high performance HDFS would be processed slowly by YARN and would probably lead to *MemoryOutOfExceptions*. Because Hadoop and every single service on top needs resources and these can quickly bring a commodity computer with standard equipment to its knees. In any case, the HDP cluster is ready for big data tasks and will be extremely helpful to us in the further course of the project. With Apache Spark, even better performance values might be achieved due to its

„in-memory“engine. The test cluster with the four VMs also showed which special features and requirements you should pay attention to (e.g. operating system version).

1.9 Hadoop Rollout

For the Hadoop rollout ten fat clients each with 32 GB RAM and 2 TB HDD as well as 128 GB SSD have been used. The nodes were connected with a 48port Gigabit switch and a routable gateway, so that all nodes were in a single local domain. For various reasons (e.g. security) the installation of a hypervisor (Proxmox) has proven to be successful, with which any number of VMs can be generated and configured (figure 1.10).

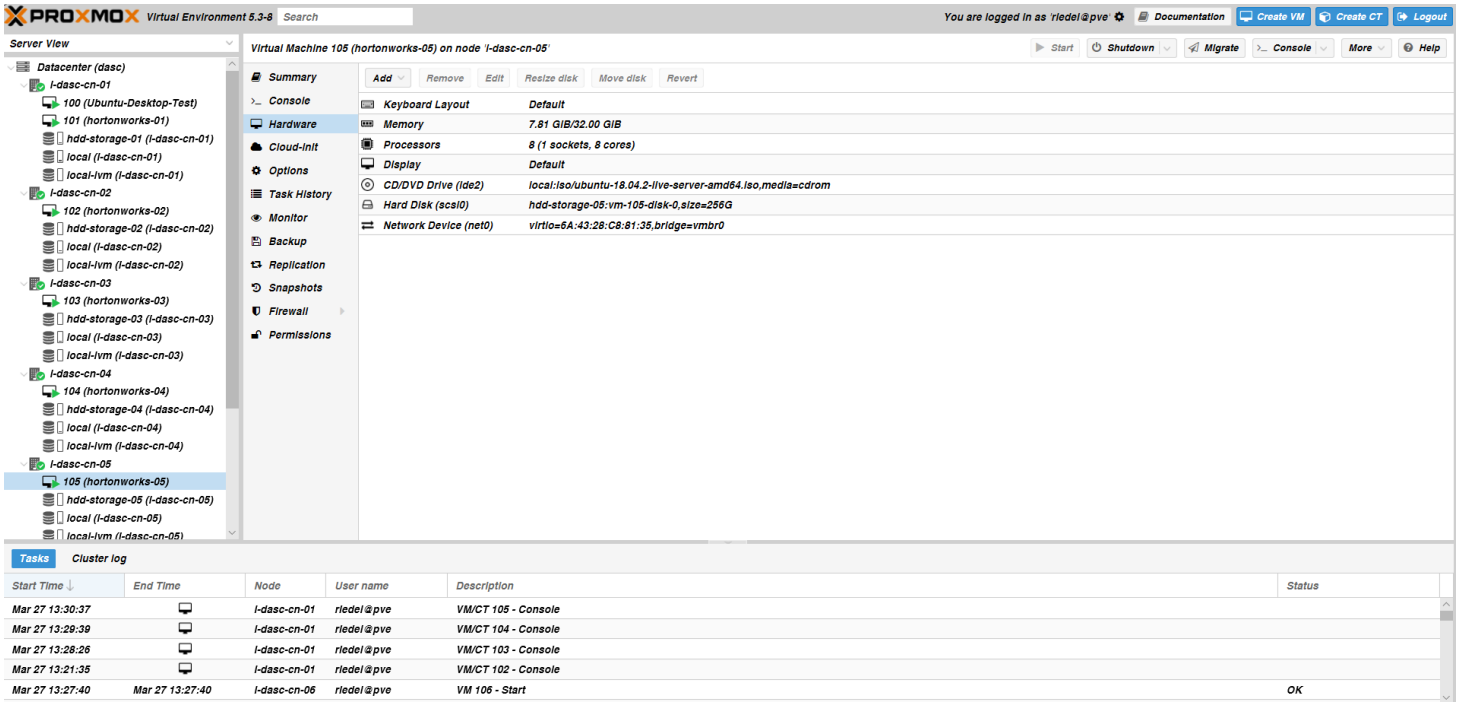

Figure 1.10: Proxmox surface

A VM with Ubuntu Server 18.04.02 LTS was installed for each node. Each VM got an allocated memory of 256 GB and a reserved memory for the Logical Volume Manager (LVM). Furthermore, sparse files were allowed and VirtIO SCSI interfaces were configured for the individual VMs, which

should guarantee maximum VM performance ([31]).

For the first rollout attempt, six cluster nodes were configured for Hadoop. The remaining four nodes were later added to the existing Hadoop cluster via Ambari surface. The chosen Hadoop stack was HDP from Hortonworks, as already explain in the Hadoop evaluation part of this work. In addition, Webmin was installed on the master node, which offers extensive monitoring services on the node. For example, the current CPU load, memory usage, kernel information and much more can be displayed. Figure 1.11 shows the surface of Webmin. Although monitoring services are also offered with Ambari Metrics, they run on top of Hadoop and can only be called when Ambari is running as well ([13]).

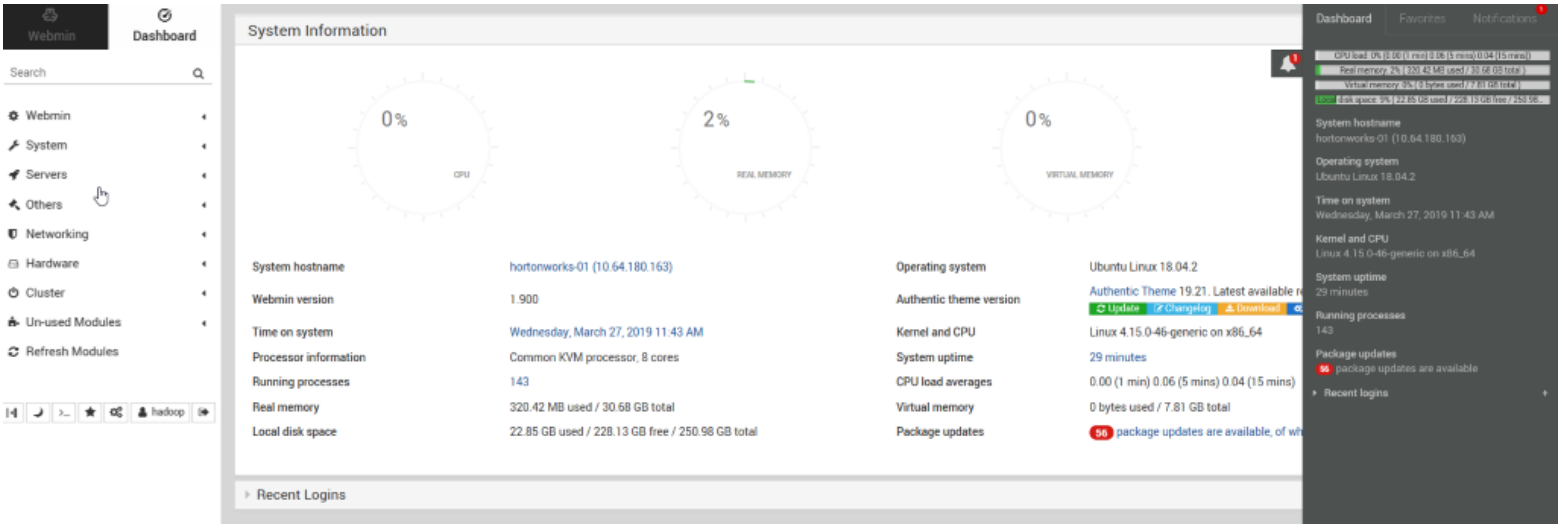

Figure 1.11: Webmin dashboard

Before the Ambari Wizard can install HDP, some pre-configurations have to be done on each VM. On the one hand, „ulimit“must be set to at least 10000, because Ambari installs several thousand dependencies. On the other hand, a password-less ssh authentication is necessary so that the master node can connect to its worker nodes without entering a password. In addition, the master node must be able to execute „sudo“commands without entering a password. This can be done by editing the „visudo“file and adding „username ALL=(ALL) NOPASSWD:ALL“. Another necessity is to add the IP addresses and hostnames of all cluster nodes under „/etc/hosts“. This

must also be done on each node individually. The following table shows the current host configurations of a worker respectively slave node:

IP-Address	**List of hostnames**
10.64.180.163	hortonworks-01.dasc.cs.thu.de hortonworks-01
10.64.83.106	hortonworks-02.dasc.cs.thu.de hortonworks-02
10.64.79.161	hortonworks-03.dasc.cs.thu.de hortonworks-03
10.64.227.154	hortonworks-04.dasc.cs.thu.de hortonworks-04
10.64.159.100	hortonworks-05.dasc.cs.thu.de hortonworks-05
10.64.204.57	hortonworks-06.dasc.cs.thu.de hortonworks-06

Table 3.4: etc/hosts of a worker node

Databases (Hive, Ranger, Druid...) are created by the Ambari Wizard as PostgreSQL databases automatically. Java as well as a corresponding JDBC connector are required for the individual services to execute database statements. The node „hortonworks-01“(see table 3.4) is both a master and a worker node. On the master node the Ambari Wizard installs Ambari server. The Ambari Agents, which are required for communication in the cluster, are installed on each worker node. Afterwards, the Ambari Wizard can be called under the following URL:
hortonworks-01.dasc.cs.thu.de:8080.
The installation routine guides the user through all necessary steps. It is important that the Ambari agents are installed on the individual worker nodes, otherwise the Ambari Wizard cannot add the nodes to the cluster. The most important step is probably the selection of the HDP services. Similar to the evaluation step with the VMs described in section 1.5, the identical service packages were selected, i.e. YARN+MapReduce2, Tez, Hive, HBase, Pig, ZooKeeper, Ambari Metrics, Spark2, Zeppelin and SmartSense. But also some new services were added for the production cluster: Storm, Accumulo, Infra Solr, Atlas and Kafka. Since the cluster is to become a long-lived high performance cluster, it might reasonable to rollout the playground for

distributed streaming platforms like Kafka or Storm which can be used for streaming analytics use-cases.

Selection of services that are running on top of Hadoop is an important part of the Hadoop cluster setup process. Following services have been chosen from the HDP stack:
all 6 virtualized nodes are DataNodes (workers) and each have a YARN NodeManager (which takes care of the resource distribution and monitoring of a node).
Furthermore, all master components run on „hortonworks-01“. However, the cluster is fault-tolerant, so if the master node fails, the worker „hortonworks-02“becomes the active master. This is made possible by the secondary NameNode service that runs on another worker node.

Once the individual accounts have been created for the different services, a final briefing is performed by Ambari before the cluster is started. A useful feature of Ambari is to download the complete configuration as a JSON template [34]. This makes another Hadoop installation much easier because the template can be reused.

This time, the installation process was done without any problems. HortonWorks (HDP 3.1.0) supports recently Ubuntu 18.04.02 LTS which makes tweaking of the operating system superfluous. Initializing and starting Hadoop services may take a few hours, depending on the size of cluster. The productive cluster now runs on Ambari 2.7.3.0 and HDP 3.1.0. By default, Zeppelin comes with a Python 2 kernel. However, it is possible to switch to the Python 3 kernel (IPython with Python 3.x).

The Ambari interface is intuitive to use. A first look at Ambari Metrics showed that the services worked properly and all workers in the cluster were active. Only YARN Registry DNS did not seem to start due to a connection refused error because Hadoop relies heavily on a functioning DNS server [14]. However, changing the YARN RDNS Binding Port from „53“to „5300“solved the problem.
Remark: the same issue happened in the evaluation part where a port conflict prevented a successful start of the Ambari server.

The physical hardware configuration of the cluster consists of 10 fat nodes,

as figure 1.12 shows.

Figure 1.12: Physical Hadoop cluster

These nodes were locally connected with a switch and a gateway so that all nodes could communicate with each other. Unfortunately, access outside the intranet was not possible, as the necessary infrastructure measures on the part of the data center are still pending. In principle, however, it would be possible to access the cluster from outside using VPN and a reliable authentication method.

First tests with the six configured Hadoop nodes could be carried out successfully. These tests were based on the Zeppelin notebooks from the previous data profiling chapter 2, which already worked successfully in the virtual cluster. Compared to the virtual cluster this time the execution was much faster, because more RAM was available and more workers (six! instead of four) were used.

Thus the cluster (figure 1.12) is in an operational and ready configured state. Of course, it is possible that a service may fail on its own or no longer run properly over time. In the evaluation phase, for example, it was shown that the YARN Timeline service fails more frequently. Usually, however, a restart of the corresponding service via the Ambari interface is sufficient.

Most Hadoop services also run autonomously, i.e. a corrupt service cannot block other running services (exception: HDFS). With the new ready to use Hadoop cluster, further data profiling action of the bicycle data can now be performed in the cluster.

2 Data Profiling

Data profiling is the process of reviewing source data, understanding structure, content and interrelationships, and identifying potential for data projects. For the project, the Santander Bicycle data will be profiled more closely [35]. For this I mainly used Zeppelin notebooks and the HDP cluster with four worker nodes initialized in chapter 1.5. Zeppelin works similar to Jupyter notebooks and also supports magic commands. However, Zeppelin stores the notebooks in .json format, while Jupyter notebooks (python) uses **.ipynb**. A corresponding import of Zeppelin notebooks into Jupyter is therefore not possible and should be considered when working with one of them. Furthermore, as already mentioned, HDP only supports python 2.7.x, so all Python code in Zeppelin is python 2.7.x as well. (The Nominatim server and Django already use Python 3.7.x).

2.1 Data Preparation

The Santander bicycle data is available on the public Transport for London TfL website. The structure of these data is as followed:

- CycleCounters: has information about speed, bike details (length, wheels, axles, class...), direction, timestamps and serial number.
  - Blackfriars May and June and July
  - Embankment May and June and July
- CycleParking: contains proprietary map material
- CycleRoutes: contains geographical spatial data (GeoJSON) and .KML files
- Usage-stats: information about rental stations, duration, start / end time

With regard to the folder structure shown above, CycleRoutes and Usage-stats are decisive. In „CycleCounters“, apart from the maximum driven speed (56 mph!) with a borrowed Santander Cycle, no noteworthy information is contained. Therefore, only the CycleRoutes and Usage-stats data are considered in the first step of Data Profiling. The CycleParking folder could contain useful map material, but I only know from Google Earth that can read **.kml** files and that it is neither compatible with Geopandas nor with Nominatim. Most raw data are available either as .csv files or as .xlsx files. One problem that arose when reading the usage-stats was that the folder contained 156 csv sheets and each file was about 30 MB in size. Reading and concatenating as Pandas DataFrame unfortunately completely occupied the maximum free memory of the master node, so the python job could not be finished and was in an endless run mode. To work around the problem, we started a Spark job with the PySpark API on the cluster instead of the Panda DataFrame. The spark job had created a DataFrame from 156 raw files within 12 seconds (i.e. it read 38.122.372 lines of raw text!). What we get here is a so called Resilient Distributed Dataset (RDD) or Spark DataFrame which, in contrast to the Pandas DataFrame, is redundant and distributed over the Cluster DataNodes. With the following PySpark method an RDD can be generated:

```
1  %spark2.pyspark
2
3  rdd_usage = spark.read.csv("/user/hadoop/usage-stats/*.csv", header=True)
```

Of course the files of the folder „usage-stats“need to be stored on HDFS before it can be read in with Spark. There are several ways of doing this, we used just a Zeppelin cell with magic command shell (%sh) to execute shell based commands within the notebook. With that one can use:

```
execute hdfs dfs -put
/home/hadoop/TFL_Cycling_Data_raw/usage-stats
/user/hadoop/usage-stats
```

to push the files on the datanodes of HDP. Remember the default replication size of Hadoop is 3, i.e. the files are replicated on three worker nodes but not on fur, which should still be resilient enough. The next decisive step is to prepare the usage-stats data and clean it up a bit. If we read in the data

as described above, our Spark DataFrame has the following columns:

```
|Rental Id|Duration|Bike Id| End Date|EndStation Id|
   EndStation Name|
Start Date|StartStation Id| StartStation Name|
```

Since we only want to plot the bicycle stations at this time, the Rental ID columns and the date columns have been removed. This has the advantage that the DataFrame is already smaller and we don't carry any unnecessary ballast. Furthermore, NaN values have been replaced by mandatory „0"and duplicates have been removed. The TFL also offers numerous .xml based live feeds, from which we can get the coordinates (longitude and latitude) to the rental stations. We can also use this to fetch the coordinates:

```
%spark2.pyspark
import requests
from xml.etree import ElementTree as ET
import pandas as pd

#Getting the coordinates tuples of the rental stations
site = "https://tfl.gov.uk/tfl/syndication/feeds/cycle-
hire/livecyclehireupdates.xml"
response = requests.get(site)
root = ET.fromstring(response.content)

#fetch relevant information
id_list = [int(root[i][0].text) for i in range(0, len(root))]
name_list = [root[i][1].text for i in range(0, len(root))]
lat_list = [float(root[i][3].text) for i in range(0, len(root))]
lon_list = [float(root[i][4].text) for i in range(0, len(root))]
capacity_list = [int(root[i][12].text) for i in range(0, len(root))]

#zip tuples
all_locs = pd.DataFrame(list(zip(name_list, id_list, lat_list,
                                 lon_list, capacity_list)), columns =
["name","id","lat","lon","capacity"])

#zip
all_locs.to_csv('/home/hadoop/TFL_Cycling_Data_raw/rental_stations_save
d.csv', header=True, index=None)
print(all_locs.shape)

#786 unique rental stations
all_locs.head()
```

This gives us not only the longitude and latitude, but also the maximum bicycle capacity of any station in London. This information could still be

quite useful. As you can see, we have 786 unique bicycle stations in London (which are quite a lot). Remark: Livecyclehireupdates.xml contains also following attributes:

- installed
- locked
- installDate
- nbBikes
- nbEmptyDocks
- nbDocks

Given that there are 786 stations across London (at least at the time of writing), this allows for 786 * 785 = 617.010 possible journey combinations if we ignore those that start and end at the same station. The next step is some data cleansing steps such as sorting, renaming and converting data types. PySpark offers similar operations for the RDDs as for Pandas DataFrames. However, some pandas such as „iloc()“cannot be transferred to Spark RDDs. Therefore Spark offers the option to directly execute SQL queries in DataFrames. Anyway, the fetched data from the live feed must still be merged with the rdd_usage Spark DataFrame. For this purpose we first saved the Pandas DataFrame locally and then uploaded it to the HDFS. This checkpoint is then read in again with PySpark as RDD. Merging works similar to SQL or/and python. To do this, simply use the „join“command to link the two DataFrames:

```
%spark2.pyspark
from pyspark.sql.functions import *

##Merging
#Merge cycle_usage RDD with RDD from livecyclehireupdates for
StartStations
ta = unq_rentals.alias('ta')
tb = rdd_rental_locs.alias('tb')
unq_rentals = ta.join(tb, ta["StartStation Id"] == tb.id)

```

In this case we also used alias (similar to the alias of SQL) in order to shorten the joining function name. The rest of the merging stuff is automatically done

by the Spark API. After all the data munging steps we got following schema of our combined and prepared usage-stats table:

```
StartStation Id|EndStation latitude|StartStation Id|
    Duration|StartStation longitude|StartStation
Address|StartStation capacity| EndStation Address|
    EndStation latitude|EndStation longitude|EndStation
     capacity|
```

One can note, that there are also the columns „StartStation ID Used“and „EndStation ID Used“which stands for frequency of rented bikes on a single station. This gives us an indication of how popular the station is in comparison with other ones. It might be also used as a weighting factor for plotting on maps which will be described in just a moment. With this, we can begin to plot statistics. But at first let compare the size of the new file with the total size of all loaded cycle-usages. The total size was around 5.4 GB (which could not be simply loaded into a Pandas DataFrame) whereas the new DataFrame has only 0.9 GB size. The reduction of the size can be explained by the fact that we removed some unused columns and dropped duplicates. This means also we are now able to use Pandas because to load 1 GB into a local memory should be possible for most common computers. In fact, the Hadoop test environment has shown its functionality and legitimation for the using in big data projects.

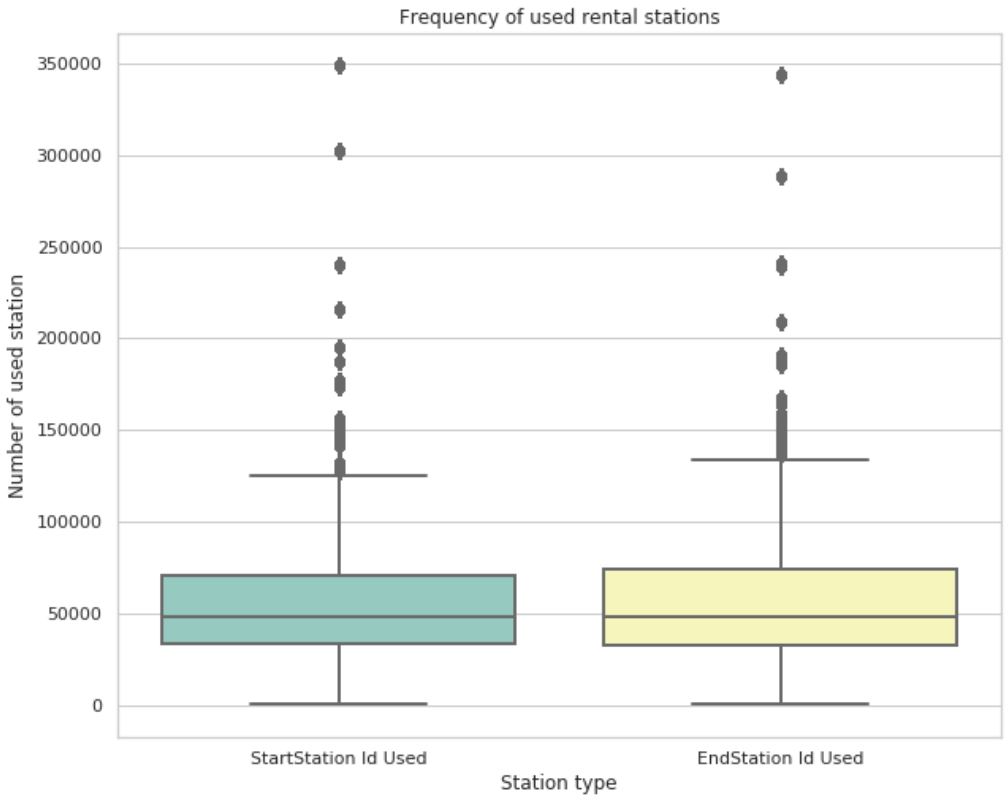

Figure 2.1: Frequency of used StartStations and EndStations

As we can see from Figure 2.1, the use of end stations slightly outweighs the starting stations, which merely means that more different end stations than different start stations were used. Apparently there are some very popular rental stations like the outliers from figure 2.1 show. It is worth noting that the data were all collected from 2016 - 2018. (The year 2014-2016 are not included in this case). The next plot should serve us to show the density center of the rental stations locations. For this a joint kernel density estimation plot could be helpful. We are plotting longitude against latitude of each single StartStation. The illustrations looks then like this:

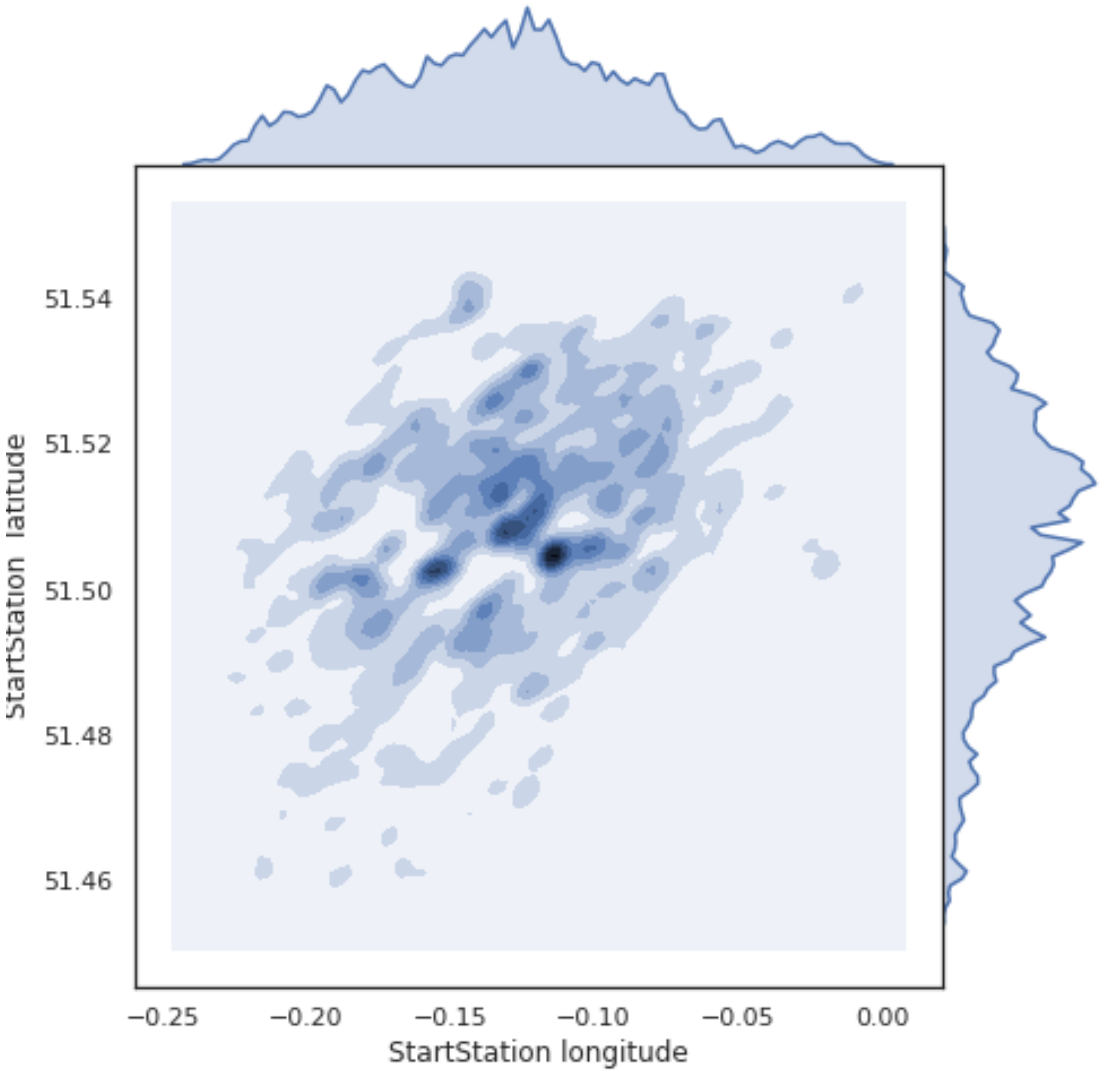

Figure 2.2: Frequency of used StartStations and EndStations

The centroid or center of this plot is indicated by the dark spots. There are the most rental stations as also the frequency distributions (i.e. histograms) on the right border and on the top are clearly showing. Apart from that center of gravity, on the right corner there are also rental stations placed (guessing between longitude: -0,01 and latitude 51,6) which indicates that

these rental stations must be placed in an outer borough of London. At a next, it would be useful to show the rental stations on a map based on their usage. As already mentioned we have 786 unique rental stations so that we can drop the duplicates for this case. In fact, we have saved the number of used stations on a separate column. At this step we prefer to use Jupyter on a local machine since Zeppelin on the cluster is not supporting IPython Display which is required by third-party packages such as folium. So we loaded the prepared cycle-usage data into a local Pandas dataframe. Now we can plot a heatmap for the rental stations:

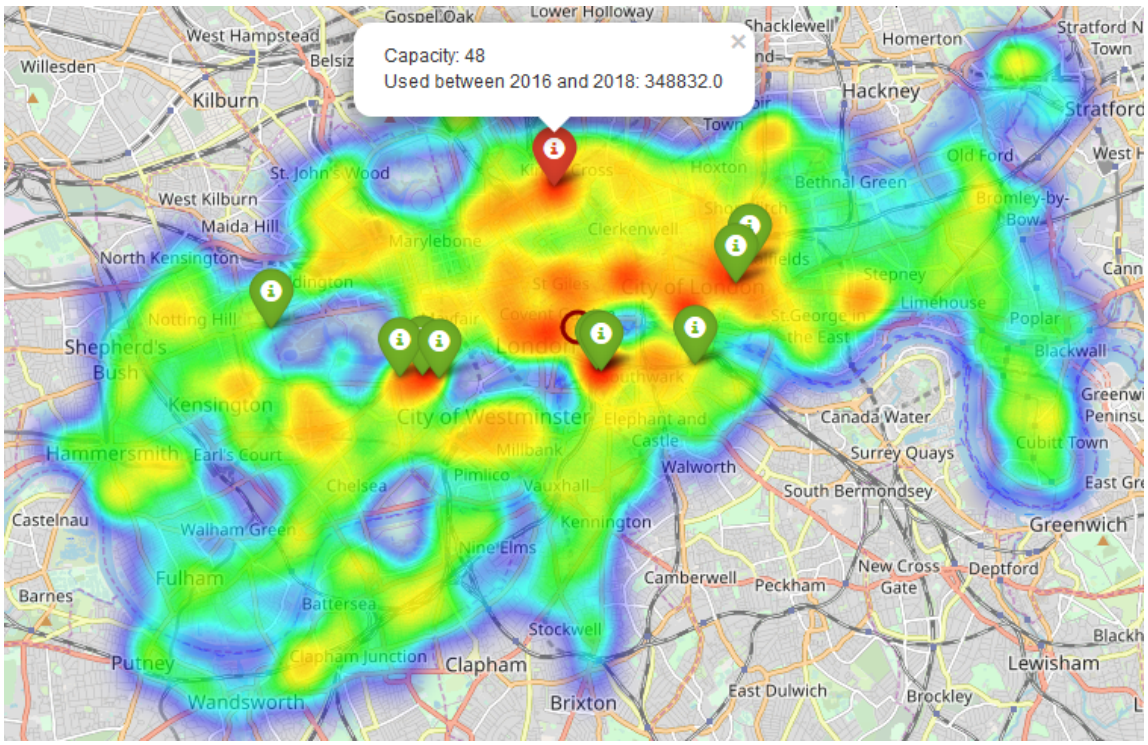

Figure 2.3: Top 10 used Santander rental stations

The red marker in figure 2.3 indicates the most often used bicycle station in London in last two years. This is not surprisingly because close to the biking station there is London train station. Probably many travelers who are arriving by train are using a Santander cycle. However, this interactive map has been saved and uploaded on a webserver on the Hadoop cluster (link) As we can see, there are many rental stations close together. For a

first impression this should be enough as one can already see the capacity of each single Santander rental bike station by clicking on the marker and also the location of each station is visible on the map. As an addition one can see the frequency of each station by looking at the sizes of the heat spots or/and reading marker details.

2.2 Conclusion of the file sizing problem

1. The sizing problem can be well handled by Spark
2. The HDP is working and storing the cycle-usage files on HDFS
3. For plotting tasks packages like folium or seaborn cannot be used with Spark DataFrame
    - Converting back to Pandas DF might exceed local RAM
    - Using Jupyter notebook on local computer for plotting maps (e.g. folium)

2.3 CyclingRoutes

The Cycling routes from the folder of TFL [24] are geojson files and represent real geographical routes within London inner city. There are 5 *geojson* files in this folder available:

- *Central_London_Grid.json*
    - *Central London Grid*: i s a set of safer, connected routes for cyclists across central London
- *Mini_Hollands.json*
    - Mini-Hollands have features that make cycling feel safer and more convenient.
- *Quietways.json*
    - *Quietways* are signposted cycle routes, run on quieter back streets to provide for those cyclists who want to travel at a more relaxed pace.

- *super-highways-quietways.json*
  - *Combination of super highways and quietways*
- *Cycle_Superhighways.json*
  - *Cycle_Superhighways* are on main roads and often segregate cyclists from other traffic.

The routes should be concatenated by a single GeoPandas GeoDataFrame. Doing so requires to read the geojson files with following scripts:

2.4 Plotting Data

Let's see if we can compare the frequency of used rental stations between StartStation and EndStation. We can not use swarm plots or any other kind of plots with fine granular details since „matplotlib"or Python would try to draw over six million dots which will easily exceed the local RAM of the master node. Beside from that, we also would see to many points at the same location since lot of stations are drawn several times. A better plot in my opinion is to use a box plot as following figure shows:

```
%spark2.pyspark
import pandas as pd
import geopandas as gpd

pd.set_option('display.expand_frame_repr', True)
pd.options.display.max_columns = None

path ='/home/hadoop/TFL_Cycling_Data_raw/CycleRoutes/'
grid_data = ["Central_London_Grid.json",
             "Mini_Hollands.json",
             "Quietways.json",
             "super-highways-quietways.json",
             "Cycle_Superhighways.json"]


cycle_routes_df = gpd.GeoDataFrame(columns=["OBJECTID", "Label",
"Route_Name", "Status", "Shape_Length", "Tag", "geometry", "gridType"])

for i in grid_data:
    print "Load " + i
    df = gpd.read_file(path+grid_data[0])
    #Extract and load json as gridType
    df["gridType"] = str(grid_data[0])[:grid_data[0].index(".json")]
    #Concatenate
```

```
25      cycle_routes_df = pd.concat([cycle_routes_df, df])
26
27
28  cycle_routes_df.dropna()
29
30  #Save it back as shape
31  cycle_routes_df.to_file(path+"CycleRoutes_Cleaned")
32
33  cycle_routes_df.tail()
```

So we get in only GeoPandas DataFrame which contains the 5 Cycling routes. In addition, the type of each route was stored in the column „gridType“, so that it can be filtered later for certain routes. For a first plot a Choropleth should show the different routes.

2.5 Conclusion

1. The data lacks in documentation and quality (Data Preperation & Cleansing necessary).
2. Some column names can be misunderstood (e.g. SITE_NUMBER <> SITE_ID).
3. Data like SPEED or SPEED_MHP should be stored in one way.
4. Many columns like the axles are not meaningful and useful for our project. (e.g. Cyclecounters has 107 columns!)
5. Raw data seems not to be complete (e.g. only May, June, July)
6. *usage-stats* is very big. Necessity to use all usage-stats?

In the next sprint we will create similar plots as shown above, but this time with duplicates and times in mind. Because it is quite often the case that between two same bicycle stations several same entries exist at different times with different rental periods. For example, a use case would be to display the day/night change on the map, i.e. do more cyclists ride quietways or super-highways at night between rental stations?

3 Prediction on Daily Basis

3.1 Data Profiling Part II

As already indicated in Data Profiling Part 1 in chapter 2, the next step is to display the use of the routes at different times between the individual bicycle stations on a map. Since the Python package "folium" uses leaflet maps based on Javascript [28], the plotting of the routes on an hourly level is not performant, because too many poly lines have to be drawn and the map can no longer be efficiently displayed in the browser. Therefore only the top 10% most used routes were plotted on the folium map. Another restriction was the aggregated granularity on a daily base. This means that the plotted map always showed the route usage for day x. With the folium plugin „TimestampedGeoJson"time series data can be plotted in JSON format. An excerpt from the script „Cycleusage & Cycleroutes [allStations].ipynb"shows the corresponding function call:

```
import folium as fo
from folium.plugins import HeatMap, MarkerCluster, TimestampedGeoJson

m = fo.Map(
    location=(51.509865, -0.118092),
    zoom_start=12,
    prefer_canvas=True
)

#Init
marker_cluster = MarkerCluster().add_to(m)
liste = []
routes_complete.fillna(0, inplace=True)
freq_mean = routes_complete["frequency"].mean()
freq_std = routes_complete["frequency"].std()

for index, p in routes_complete[:top10].iterrows():
    startX = p["lat"]
    startY = p["lon"]

    if (startX == 0):
            fo.PolyLine(liste, weight=((freq - freq_mean)/freq_std),
opacity=0.5, popup="Used between 2015 and 2018: " + str(int(freq)) +
"x",color="blue").add_to(m)
            liste = []
    else:
            freq = int(p["frequency"])
            liste.append((startX, startY))

#Add markers (unique)
df_startStations[:len(df_startStations)].apply(lambda
row:fo.Marker(location=[row["StartStation latitude"], row["StartStation
longitude"]],popup=row['StartStation Address'] + '</br>' + "Capacity: "

```

```
+ row['StartStation capacity'] + '</br>' + " Used between 2016 and
2018: " + str(int(row['StartStation Id Used']))


#Put dates and coordinates in required format
features = [
    {
        'type': 'Feature',
        'geometry': {
            'type': 'LineString',
            'coordinates': coordinates[i],
        },
        'properties': {
            'times': dates[i],
            'style': {
                'color': 'red',
                'weight': weights[i][0],
                'opacity': 1
            }
        }
    }
    for i in range(len(dates))
]

#Allows polylines over time (i.e. red lines), JSON format required;
TimestampedGeoJson({
    'type': 'FeatureCollection',
    'features': features,
    },
    auto_play=False,
    loop=False,
    loop_button=True,
    date_options='YYYY/DD/MM',
    duration='P2M',
    add_last_point=False).add_to(m)

m.save(outfile= "routes_frequency.html")
m

```

Figure 3.1: Route usage over time plot

As the code from figure 3.1 shows, the poly lines on the map have been added iteratively. The weight parameter can be used to determine the thickness of the poly line. Since these should look as dynamic as possible on the map, the weight has been standardized. The fixed stations were initially

added, but no duplicate stations were plotted. A disadvantage of the plugin is that it needs the data in a JSON format. Therefore the coordinates for the single points of a poly line as well as the time series data had to be converted into a compatible format. As can be seen from the Python code, the coordinates must be given the type „LineString“. A LineString is defined as a sequence of uniquely assignable points. In this case, the longitude and latitude previously requested using the Graphhopper API on our Graphhopper server were sufficient. It is important that the parameter „date_options“in „TimestampedGeoJson“corresponds exactly to the date format as in the nested list dates, otherwise the time slider function on the map will not work properly. Applied to the bicycle data the following picture results for the time 26.06.2016:

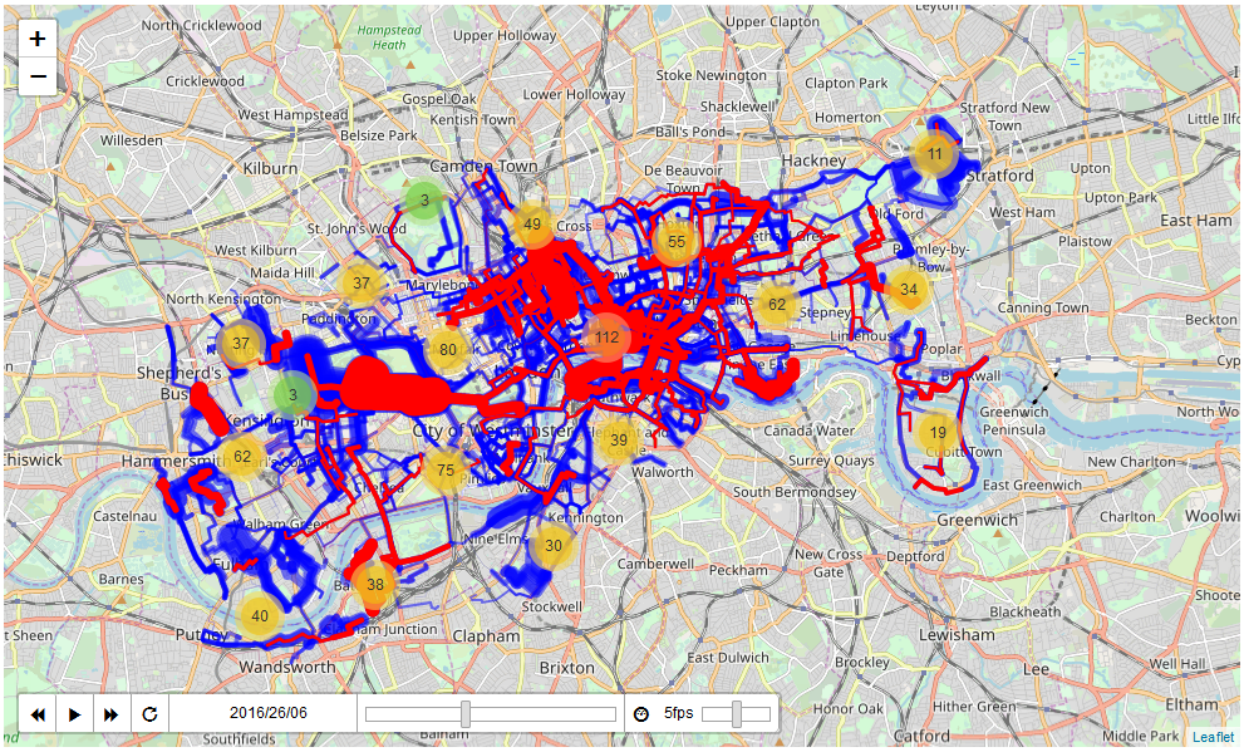

Figure 3.2: Used bicycle routes over time plot

From figure 3.2 it can be seen that on this day there was a moderate use of Santander Bicycles in inner London. Especially the hubs such as Kings Cross or Hyde Park were obviously used very often with Santander Bicycles

on this spring day. Furthermore, the map with the thickness of the poly lines shows how often this route was used in relation to the total use of all routes. The red colored routes from figure 3.2 are the actually used routes on this particular day, while the blue routes are the „inactive“ones. The bubbles with the numbers represent the respective rental stations. These have only been aggregated to provide a clearer representation. If the map is zoomed in (figure 3.2), the granularity is refined and the markers of the individual rental station locations are displayed. The color of these bubbles correlates with the number of aggregated stations in the vicinity. This method also has the advantage that it is easy to see where most rental stations are located. In fact, with 112 stations (see figure 3.2), the inner districts connected by the Waterloo Bridge and the London Blackfriars Bridge form the center of most stations which are close by. It should be noted, however, that new rental stations are constantly being added (even given up again!), and a new data extract could result in a different picture. Therefore this assumption is valid for the selected date from figure 3.2, but not for today or in two years. A useful feature that comes along with the folium plugin is the automatic data display sequence [28]. When someone clicks on the „Play“button from figure 3.2, all time series data is automatically run through. The speed can be adjusted with the „fps“slider. To give a counterexample, the following map shows the use of the routes on a winter's day:

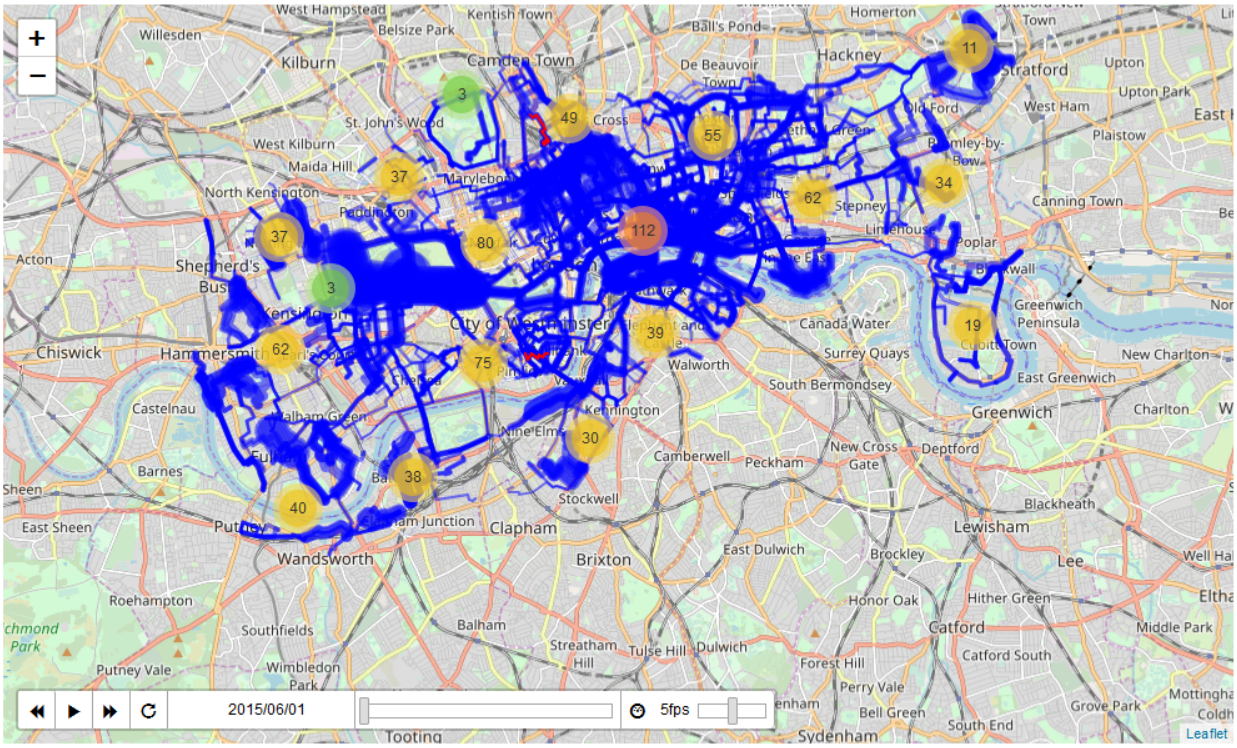

Figure 3.3: Used bicycle routes over time plot

On 06.01.2015 obviously less bicycles were rented than in figure 3.2. This is due to the fact that at this time it was winter and on the other hand there were only 315 Santander bicycle rental stations all over London [18]. In the course of time, new stations were added again and again, resulting in a network which is monitored and managed by the local government and administrated by TfL. The interactive map is also available on the Hadoop Cluster. Overall, it can be observed that the usage of the rental stations has risen sharply on average over the last four years. This can be explained by the expansion of the network but also by the increased environmental awareness of the citizens. It is to be expected that further stations will be connected in the future, so that ideally there will be one rental station at each crossroad in the town.

3.2 Feature Engineering Kings Cross

Next, the features for the most frequently used station (Kings Cross) were prepared and aggregated on a daily basis. In addition, these features have been extended by more like *Holidays, Weekdays, Months, Seasons* and weather data. This enrichment of the features should help to achieve a higher accuracy of the learning model. For the weather data the weather API „Dark Sky“was used. Like every available weather API the free use is limited. In the case of Dark Sky a maximum of 1000 API request calls per day can be executed with one key [27]. However, this limitation can be bypassed if multiple accounts are used. Since each account has a key, the keys can be collected and thus in fact significantly more requests can be executed. However, this cannot be applied to the existing data sets of the bicycle data, since the daily aggregated data of Kings Cross alone has already 376625 rows. Therefore a „dates“file was created, which contains all days from 04.01.2015 - 14.04.2019 and thus covers all possible date values of the rental station data. This has the advantage that now only 1562 API requests are necessary and two keys are sufficient. The disadvantage of this method is that of course no precise weather data can be retrieved at every station. Therefore the coordinates for the weather data of the centroids (51.509865,-0.118092) of London were chosen. Dark Sky API also returns a JSON file as response, which can be searched for the desired information. The following function from the script „Cycleusage & Cycleroutes [allStations].ipynb“shows the corresponding request call:

```
### Read weather data for London with darksky api (1000 requests per
day)
df_dates = pd.read_csv("dates.csv")
df_dates['Start Date'].replace('\.','/',inplace=True, regex=True)
df_dates.head()
#api_key2 = "f79a70fd65182047247a0506f998a96f"
api_key = "fbe9812e95cfc01d44c8d28e28d155e0"

df_dates["Daily Weather"] = ""
df_dates["Humidity"] = ""
df_dates["Windspeed"] = ""
df_dates["Apparent Temperature (Avg)"] = ""
df_dates["Hourly Weather"] = ""

for index, p in df_dates[0:1].iterrows():
    timestamp = str(int(datetime.strptime(df_dates["Start
Date"][index], '%d/%m/%Y'

).replace(tzinfo=timezone.utc).timestamp()))
    weather_resp =
requests.get('https://api.darksky.net/forecast/'+api_key+'/51.509865,-
0.118092,'+timestamp+'?exclude=currently,flags,minutely')
    df_dates["Daily Weather"].iloc[index] =
weather_resp.json()['daily']['data'][0].get('icon', 'No weather data')
    df_dates["Humidity"].iloc[index] =
weather_resp.json()['daily']['data'][0].get('humidity', 0)

    df_dates["Windspeed"].iloc[index] =
weather_resp.json()['daily']['data'][0].get('windSpeed', 0)
    df_dates["Apparent Temperature (Avg)"].iloc[index] =
(float(weather_resp.json()['daily']['data'][0].get('apparentTemperature
Low',
0))+float(weather_resp.json()['daily']['data'][0].get('apparentTemperat
ureHigh', 0)))/2.0
    df_dates["Hourly Weather"].iloc[index] =
weather_resp.json()['hourly']['data']

df_dates.head()
df_dates.to_csv("dates and weather.csv", index=None, header=True)
```

Figure 3.4: Fetching the weather API with Python

As the code from figure 3.4 clearly shows, the following weather data were

considered as relevant: „Daily Weather, Humidity, Windspeed“and „Apparent Temperature (Avg)“. The feature „Daily weather“returns a string value, how general the weather was on that particular day (e.g. sunny, partly-cloudy...). Dark Sky makes this value dependent on the worst condition that occurred on that day [27]. That is, if it snowed for an hour, but the rest of the day was cloudy, „Daily Weather“will still show „snowy“because this condition is weighted higher.

In addition to the weather data mentioned above, the hourly weather data of each day was also queried as JSON lists, as otherwise every hour of each day would have to be queried separately, which would drastically increase the API calls. A drawback of this variant is that the dates and weather dataframe has a mixed structure. While the column „hourly weather“is a JSON format, the other columns have a normal structure. This problem is further addressed in the section „Data Preparation - Hourly Base“.

The processed weather data were next merged with the processed cycle usage dataframe. For the Holidays the Python package „holidays“was used, which returns the corresponding holidays to a selected location. Similarly, the features *Weekdays, Months* and *Seasons* were be generated with defined functions and afterwards merged back with the main dataframe.

Subsequently, the dataframe was supplemented by „past“and„future“data. This should improve the training of regression models, for example. A kind of „sliding window“was implemented for the past data. This means that these columns always contain the value of the previous day. With the future usage data, the daily number of borrowed bikes of the next day was also displayed. The column „Rented Bikes“(i.e. usage) corresponds to the class variable of interest. This has the consequence that the first and last day in the dataframe have missing values, because there is no data for this naturally. The prepared dataframe for Kings Cross on a daily basis looks after the preparation steps like this:

	Month	Season	Weekday	Holiday	Daily Weather	Daily Weather (Past)	Humidity	Humidity (Past)	Windspeed	Windspeed (Past)	Apparent Temperature (Avg)	Apparent Temperature (Avg) (Past)	Rented Bikes	Rented Bikes (Future)
1510	March	Spring	Friday	False	wind	wind	0.80	0.71	19.56	16.31	54.260	50.685	258	28
1511	March	Spring	Saturday	False	wind	wind	0.80	0.80	20.50	19.56	42.860	54.260	28	41
1512	March	Spring	Sunday	True	partly-cloudy-day	wind	0.74	0.80	13.56	20.50	38.345	42.860	41	327
1513	March	Spring	Monday	True	clear-day	partly-cloudy-day	0.74	0.74	7.60	13.56	46.570	38.345	327	323
1514	March	Spring	Tuesday	False	partly-cloudy-day	clear-day	0.83	0.74	6.26	7.60	50.315	46.570	323	33

Figure 3.5: Prepared data frame of Kings Cross (daily)

As one can see from figure 3.5, the past data was only included for the weather data whereas the future data was only added for the usage of the rental station. The string values were later be transformed into numerical values since machine learning methods always require number values instead of string values. Therefore a proper schema was defined, which is described in more detail in the modeling section.

With the processed bicycle data from 3.5, machine learning can already be used to predict, for example, the daily usage of bicycles (i.e. how many bicycles will be borrowed tomorrow starting from today?). The script „Feature Engineering Kings Cross.ipynb“contains the preparation functions described above and can be found on our GitHub repository.

3.3 Data Prediction with MLPRegressor

Since we wanted to implement the prediction algorithm in Python we used the library *Scikit Learn* which provides a class *Multi-layer Perceptron Regressor (MLPRegressor)* with according functions. As we wanted to train a non-linear model we decided to use a Multi-layer Perceptron since it can learn a non-linear function approximator for either classification or regression. However the MLPRegressor uses backpropagation which is the most widely used algorithm for supervised learning with multi-layered feed-forward networks [33], this algorithm was implemented to train a prediction model for the rental bike usage on a daily base.

The outcome of of the Data Profiling Part 2, as described in chapter **??** served as a basis for the prediction. As a first implementation we used the following data as an input for the feature matrix: *Month, Weekday, Day, Season, Daily Weather, Daily Weather Past, Humidity, Humidity Past, Windspeed, Windspeed Past, Apparent Temperature (Avg), Apparent Temperature (Avg) Past, Rented Bikes*. Since we assume that weather conditions play an important role in bike usage we decided to add weather data. Another assumption was that on weekdays the frequency will rise, because a lot of people use rented bikes to travel to work. Furthermore we added the number of rentals of today in order to predict the ones of tomorrow. Therefore *Rented Bikes Future* served as our target variable which is to predicted.
Since the MLPRegressor works with data represented as dense and sparse numpy arrays of floating point values, data had to be encoded accordingly beforehand. Therefore we mapped all of the ordinal scaled data like *Month, Weekday, Season, Daily Weather* to numerical data, by implementing a dictionary which assigns numerical values to ordinal data. E. g. For the weekdays we used the following dictionary:

```
"Weekday": {"Monday": 1, "Tuesday": 2, "Wednesday": 3,
    "Thursday": 4,"Friday": 5, "Saturday": 6, "Sunday"
   :7 }
```

After the encoding we split the data into 80 % training data and 20% test data. Since the MLPRegressor is sensitive to feature scaling we normalized the data accordingly to the activation function. Since we used the *logistic* activation function which expects values between [0,1] we normalized the feature matrix as well as the target variable beforehand. Scikit Learn provides several scaling mechanism to rescale the data. After data was scaled we applied the MLPRegressor with *logistic* as an activation function and ten hidden layers with five neurons. After the prediction we denormalized the data back to its original state. In order to rate the accuracy of the prediction we computed the Root Mean Square Error (RMSE) which measures the difference between actual and predicted values of a model. The less the difference respectively the value, the better is the model. To get a better impression of how the prediction worked, the results were plotted as time series. The time series was plotted over for years, which causes the plot to be very large. In order to make it more user friendly, we implemented an interactive plot which gives the user the opportunity to zoom into specific parts or cut out

some parts to look at them more closely.

Normalization

In the first attempt we used the *MinMaxScaler* which rescales the data such that all vales are in the range of [0,1]. This gave us a RMSE of 122.347. To get a better impression of what that means we plotted this result which can be seen in figure 3.8.

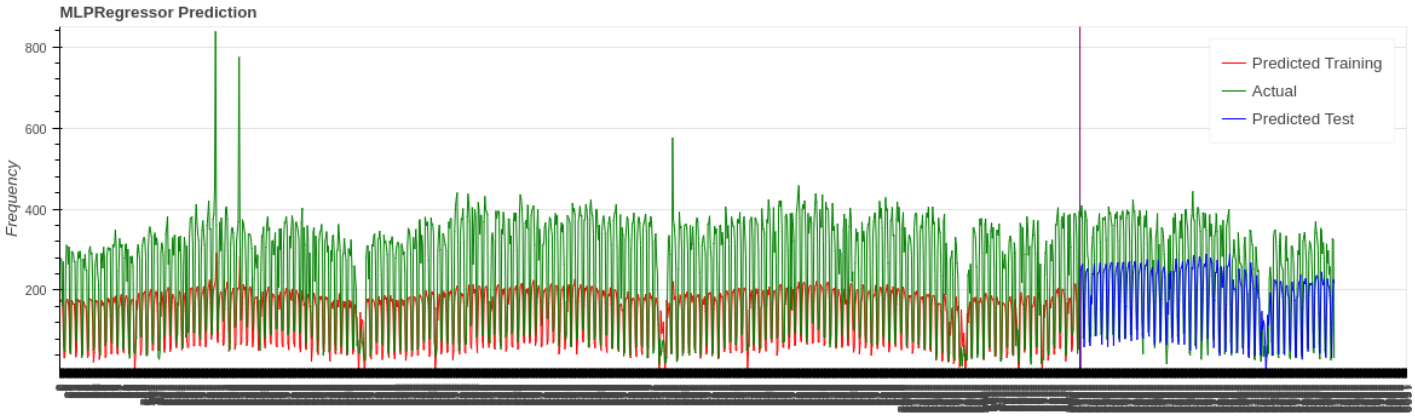

Figure 3.6: Result of MLPRegressor with MinMaxScaler

This result shows that the prediction lack accuracy. As we found out this was caused by the MinMaxScaler, since this scaler is very sensitive to the presence of outliers. Therefore we chose another scaler for normalization which is more prone to outliers. In order to find a more suitable scaler, experiments were made with all available scalers of Scikit Learn which met the requirements for the output of the scaling. The best result was achieved by the *QuantileTransformer* with a RMSE of 57.837, therefore we stuck with this normalization method.

Feature Evaluation

In order to improve accuracy, further experiments were carried out to evaluate the features. As mentioned earlier the previous predictions were made with a feature matrix of thirteen features. To evaluate which features improve the prediction and which are useless 18 test cases were carried out.

Each test case consists of different constellations of features. In the end the different RMSE values were compared, which can be seen in figure 3.7.

	Test	Day	Month	Season	Week-day	Daily Weather	Daily Weather Past	Humi-dity	Humi-dity Past	Wind-speed	Wind-speed Past	A Temp Avg	A Temp Avg Past	Rented Bikes	RMSE
MLP Regressor	1	1	1	1	1	1	1	1	1	1	1	1	1	1	57.837
	2	1	1	1	1	1	0	1	0	1	0	1	0	1	53.178
	3	0	1	1	1	1	1	1	1	1	1	1	1	1	56.406
	4	1	0	1	1	1	1	1	1	1	1	1	1	1	62.382
	5	1	1	0	1	1	1	1	1	1	1	1	1	1	56.827
	6	1	1	1	0	1	1	1	1	1	1	1	1	1	130.375
	7	1	1	1	1	0	1	1	1	1	1	1	1	1	57.200
	8	1	1	1	1	1	1	0	1	1	1	1	1	1	57.547
	9	1	1	1	1	1	1	1	1	0	1	1	1	1	58.855
	10	1	1	1	1	1	1	1	1	1	1	0	1	1	56.681
	11	1	1	1	1	1	1	1	1	1	1	1	1	0	226.706
	13	0	0	0	1	1	1	1	1	1	1	1	1	1	54.890
	14	1	1	1	1	0	1	1	1	0	1	0	1	1	59.239
	15	1	1	0	1	1	1	0	1	0	1	1	1	1	63.752
	16	0	1	0	1	1	1	0	1	0	1	1	1	1	51.414
	17	0	0	0	1	0	1	0	1	0	1	1	1	1	52.561
	18	0	0	0	1	0	1	0	1	0	1	0	1	1	54.866

Figure 3.7: Feature evaluation test cases

Figure 3.7 shows beneath the different test cases, included features, those are assigned to „1“and excluded features which are assigned to „0“. The first RMSE value, colored in blue depicts the prediction accuracy with all 13 features. The red colored values show the constellations which are significantly worse than the original one and in turn the green values represent the values with an increased accuracy than the original one.

Result

This evaluation showed that the features *Weekday* and *Rented Bikes* are the most important ones and in turn the features *Day* and *Season* are less important. Based on our testing results we recommend to use the following nine features: *Weekday, Month, Past Data, Apparent Temperature (Avg), Daily Weather* and *Rented Bikes*. Moreover experiments with scalers showed that the *QuantileTransformer* is more prone to outliers than others and is therefore the scaler of choice.
First plots were made with the library *Matplotlib* which turns out to be very

restricted and complicated to handle in order to create interactive plots. Therefore we recommend to use *Bokeh* which provides easy to implement interactive plots especially for large data sets.
Applying the recommendations regarding scaling and feature evaluation we received a RMSE value of 51.414 which is the best value achieved during the project phase. The overall result visualized with *Bokeh* can be seen in figure 3.8.

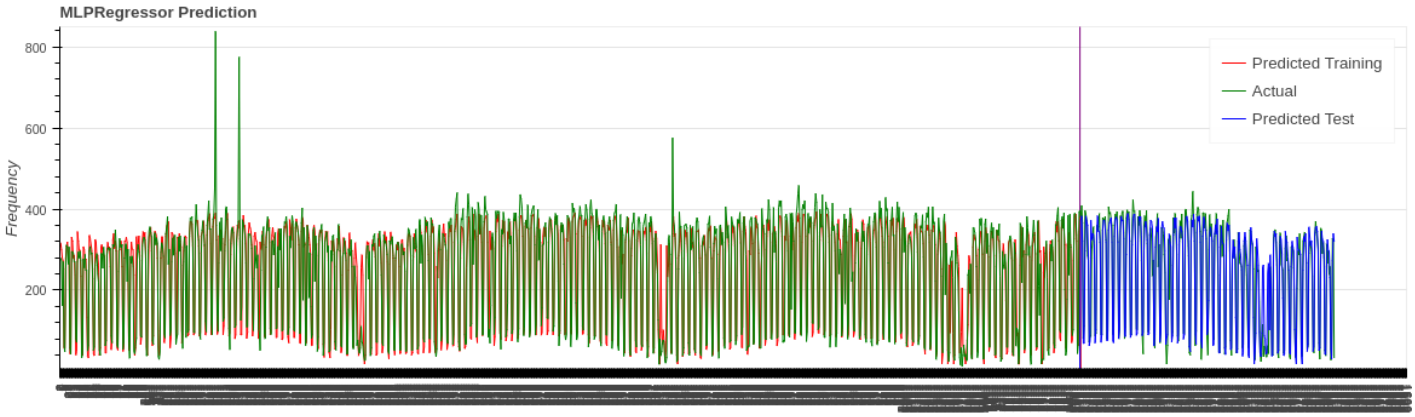

Figure 3.8: Accuracy with recommended features and scaling by the QuantileTransformer

Compared to figure 3.6 one can see significant improvements in accuracy. This model is based on the most used station, which is King´s Cross with a dataset of 1515 records. In order to confirm this model we applied it to the least used station in Farringdon Street, Holborn as well. The result can be seen in figure 3.9.

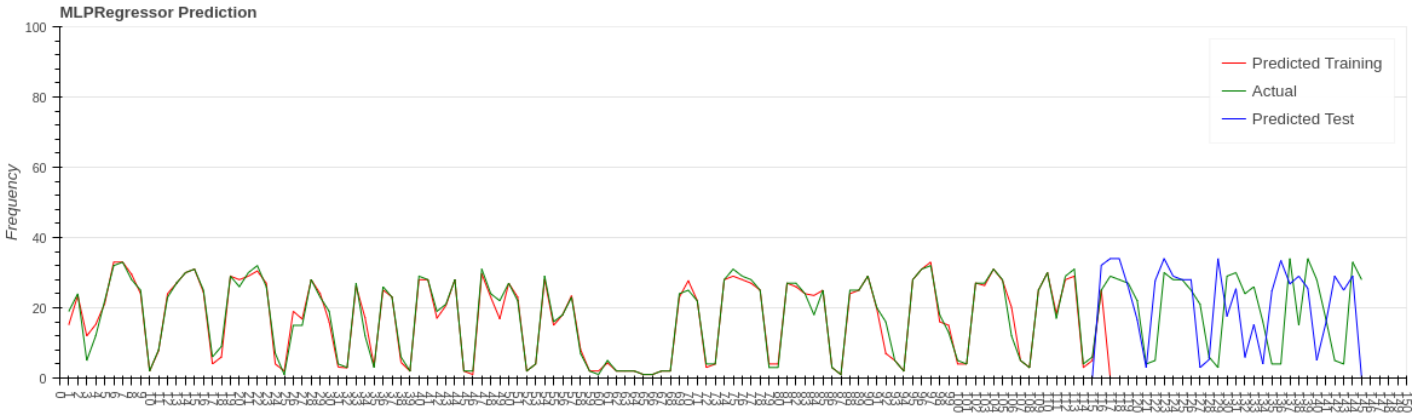

Figure 3.9: Accuracy with recommended features and scaling by the QuantileTransformer

The accuracy of the prediction of the least used station measures a RMSE of 9.311. This shows that the model works not only for highly frequented stations but also for stations like Farringdon Street with only 145 records.

3.4 Modeling (Polynomial Regression)

Since we don't have a linear relationship between the data, a linear regression will not be helpful. For example, the regress „Rented Bikes"is not a linear correlation of temperature as even on rainy days there is slight chance that more people rent a bike than on a sunny day due to a special holiday. Therefore polynomial regression was a selected machine learning method for the prediction of rented bikes on a station.

Polynomial regression belongs to the regression forms [36]. In fact, it is just a modified version of a linear regression. This means the independent variable x and the dependent variable y is modeled as an nth digress (so-called polynomial) in x [36].

In a more formal way, the polynomial regression can be expressed as following:

$$Y = \beta_0 + \beta_1 * x + \beta_2 * x^2 + \beta_3 * x^3 + ... + \beta_n * x^n$$

Where n is the degree of the regression.

With the Scikit-library in python a data scientist can import the function „PolynomialFeatures“from „sklearn.preprocessing“which transforms linear data into higher dimensional data. For example one could apply „poly = PolynomialFeatures(degree = 3)“to get a polynomial regression in the third dimension. This should improve the accuracy as our underlying data has no linear relationships but maybe higher dimensional ones. Furthermore, the higher the degree the better the accuracy should be. Unfortunately the computation time is exponential. A degree of 4 already took several hours to perform and was only slightly better than a regression in the third dimension.

The RMSE error on a degree of 4 was around 48,4, which is in comparison to the other tested ones not really bad but maybe also not best one.

Figure 3.10 shows the different plots of each feature and the prediction (rented usage). It turns out that the feature Season, for example, has no German influence on the use of bicycles, whereby there was apparently an outlier in spring with 800 rented bicycles. It is also becoming apparent that bicycles are rented more frequently at low wind speeds than at high wind speeds. The average temperatures between 30 and 70 Fahrenheit are particularly high. Unfortunately, the addition of the past data has not caused much change in accuracy, as shown in figure 3.10.

Figure 3.11 shows a plot of the tested data (prediction) with the feature „Daily Weather“. The plot looks relatively good, except for a few single outliers at 2 (partly-cloudy-day), the prediction of the tested data matches the training data.

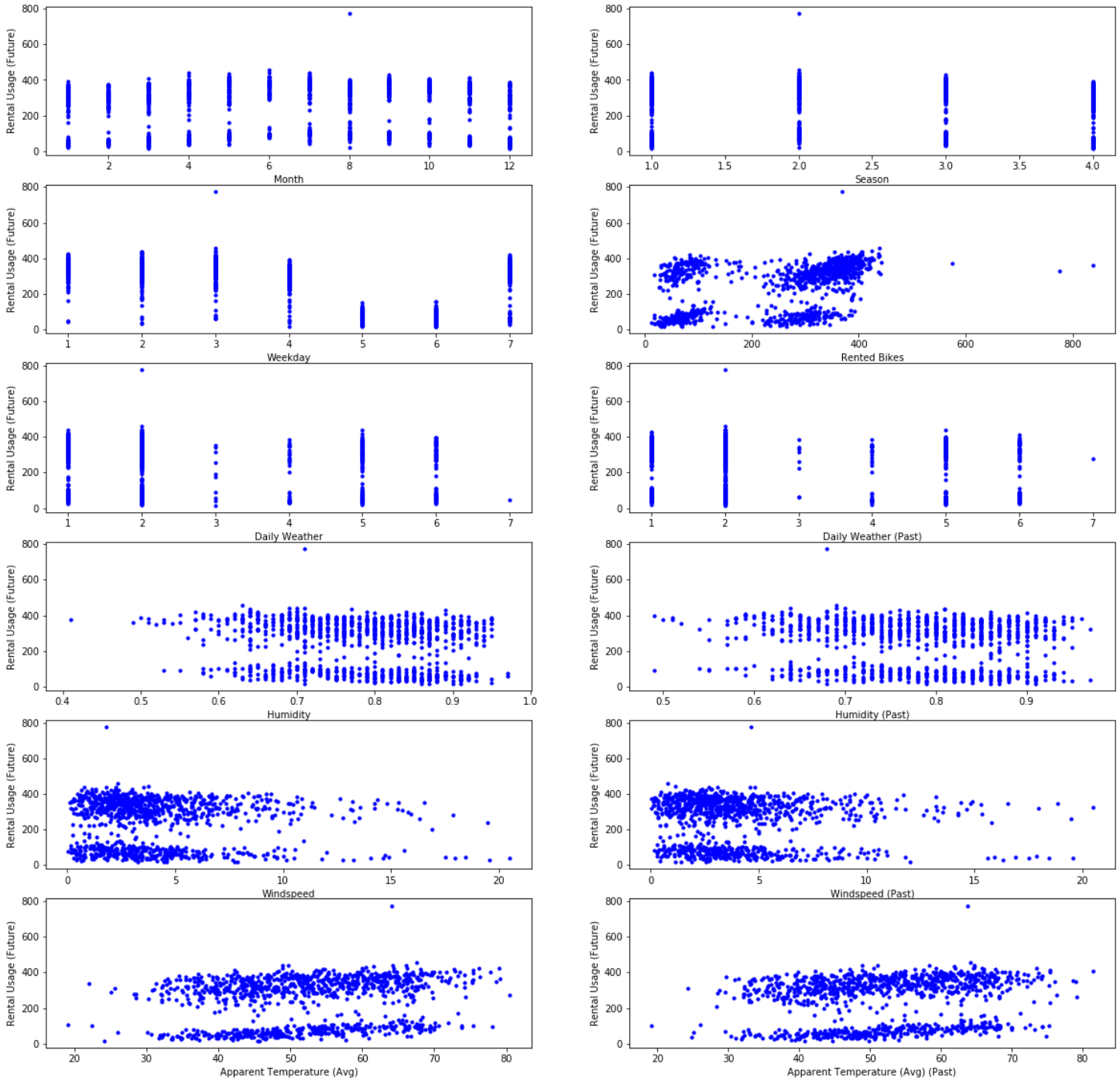

Figure 3.10: Plot of polynomial regression (features)

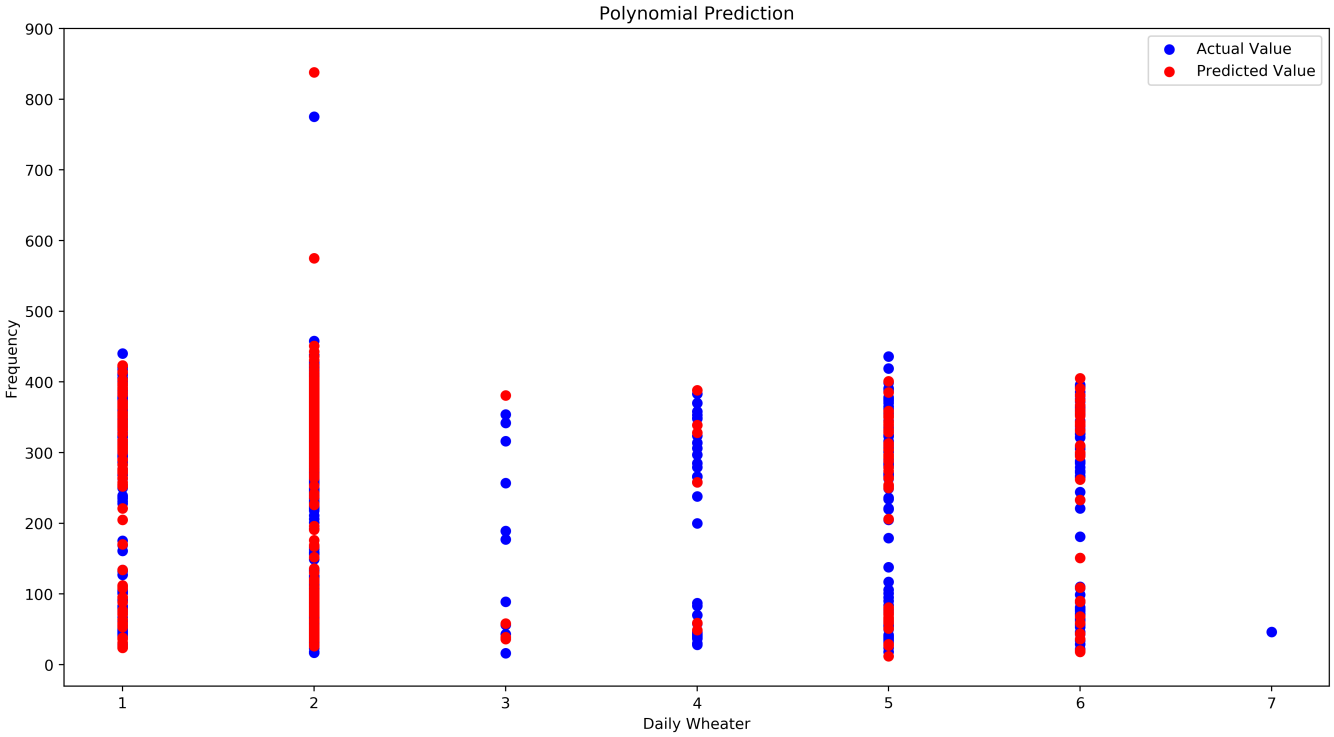

Figure 3.11: Polynomial Prediction of rental usage on daily weather

4 Prediction on Hourly Basis

4.1 Data Preparation

Since the long term goal was to predict the usage on a hourly base, some further data transformation steps are necessary to achieve this goal. Furthermore the data records will increase significantly on an hourly granularity. Therefore the normal use of Anaconda and Jupyter on a local computer may be not sufficient due to low physical memory. An ideal use case for our newly installed Hadoop cluster! There PySpark can be used as already used under Data Profiling Part 1 in chapter 2 to manage „big data“transformations. Unfortunately the behavior and syntax of PySpark is sometimes a little more complicated than Pandas. For example, in PySpark it is not easily possible to iterate over rows since the data frame is distributed over the worker nodes and thus it only allows column wise operations. Moreover, Pandas opera-

tions such as „iloc“ are not available in PySpark. But the API comes also with some advantages, e.g. it is quite performant on big data scale (i.e. it can easily perform several million of records) and it has an SQL approach. Functions like „select“„where“and „filter“are syntactical close by to SQL as we know from MySQL and other database management systems.

The structure of the weather data is inconsistent due to „hourly weather“While the other columns only contain simple values, the column „hourly weather“contains nested JSON lists. This means that these nested lists must somehow become normal columns. This was a little more complicated than expected, but not impossible. Fortunately, PySpark allows one to define „schemas“that are used as a kind of blueprint by Spark to read the Spark data frame. With the following code one can already create normal columns from the JSON lists:

```
%spark2.pyspark
from pyspark.sql.functions import from_json
…
#Defined schema
schema_hum = ArrayType(
    StructType([StructField("time", StringType(), True),
            StructField("humidity", FloatType(), True)]))

rdd_weather = spark.read.csv("/user/hadoop/weather_new.csv",
header=True, sep=",")

#Add new column
rdd_weather = rdd_weather.withColumn(colChecker(i, "hum")+`i`,
from_json(rdd_weather["Hourly Weather"],
schema_hum)[k].getItem("humidity"))
…
```

Figure 4.1: Transformation of humidity columns (excerpt)

he excerpt from 4.1 shows that „structTypes“can be used to search the individual sub lists of „hourly weather“. The complete script is contained in the Zeppelin notebook „DFGeneration.json“and can also be found on GitHub.

The described transformation creates a new column with the corresponding value for each hour. The script works dynamically. For example, the user can look at the data with two hours from today, which then looks like this:

Start Date	hum00	hum01	sum00	sum01	temp00	temp01	wind00	wind01
2015-01-04	0.93	0.94	partly-cloudy-night	partly-cloudy-night	37.24	36.35	0.94	0.56
2015-01-05	0.94	0.95	partly-cloudy-night	cloudy	37.62	37.97	0.37	0.18
2015-01-06	0.82	0.83	cloudy	cloudy	45.99	45.87	1.44	1.58
2015-01-07	0.9	0.91	clear-night	clear-night	35.64	35.42	0.64	0.64
2015-01-08	0.9	0.9	partly-cloudy-night	partly-cloudy-night	47.21	50.5	6.27	5.51
2015-01-09	0.81	0.79	partly-cloudy-night	partly-cloudy-night	44.75	46.14	6.13	5.73
2015-01-10	0.82	0.85	partly-cloudy-night	cloudy	56.73	55.95	7.88	8.55
2015-01-11	0.76	0.79	clear-night	clear-night	35.06	35.14	4.56	4.27

Figure 4.2: Weather dataframe after transformation for two hours

This (figure 4.2), however, is more like a jumping window technique as it uses days as start date and not the the single hours on each day. Nevertheless the structure of the data frame from figure 4.2 can be used as ground for applying sliding window method.

However, before the sliding window procedure could be implemented, the „Start Date“from figure 4.2 had to be converted to an hourly format, whereby the individual hour columns had to be combined so that only one „current column“existed. For example, „hum00“and „hum01“should become something like „Current Humidity“which shows the humidity value of each hour. This means that further data preparation steps are necessary.

First, the exact timestamps of the bicycle data were read in from the provided TfL website and rounded off to an hourly level. For example, „23:38“became „23:00“Minutes and seconds were ignored. An arithmetic lap (half-round mode) did not make sense, because then there would be overlaps with the successor day when 23 o'clock is rounded up. However, this also means that a slight distortion must be assumed. For example, many bicycles could be rented at a bicycle station at 17:52 and significantly less after 18 o'clock. Then one would notice a peak at 17 o'clock and a low usage at 18 o'clock, although in fact more bicycles were rented around 18 o'clock.

Another problem was that not for every hour there was data about the usage of the bicycle stations. However, this data quality problem could be solved relatively easy. The missing hours can be determined by a user defined function. Since every day has 24 hours, this can be calculated manually as

following (figure 4.3):

```
+ row['StartStation capacity'] + '</br>' + " Used between 2016 and
2018: " + str(int(row['StartStation Id Used']))


#Put dates and coordinates in required format
features = [
    {
        'type': 'Feature',
        'geometry': {
            'type': 'LineString',
            'coordinates': coordinates[i],
        },
        'properties': {
            'times': dates[i],
            'style': {
                'color': 'red',
                'weight': weights[i][0],
                'opacity': 1
            }
        }
    }
    for i in range(len(dates))
]

#Allows polylines over time (i.e. red lines), JSON format required;
TimestampedGeoJson({
    'type': 'FeatureCollection',
    'features': features,
    },
    auto_play=False,
    loop=False,
    loop_button=True,
    date_options='YYYY/DD/MM',
    duration='P2M',
    add_last_point=False).add_to(m)

m.save(outfile= "routes_frequency.html")
m

```

Figure 4.3: Adding missing hours on usage data

A prerequisite of this method from figure 4.3 is that at least one maximum value and one minimum value exist that serve as boundaries. If no

data was collected on a day at a station, null values would appear. This does not happen, however, and even if it did, it would be removed at a later stage as such data does not provide any advantage for training a machine learning model.

Another problem was how to transform the individual hour columns of the weather data so that only one column exists instead of e.g. 24 columns, whereby this new column should contain all values of the 24 columns arranged correctly. Fortunately the function „explode“exists in PySpark and is provided via the class „pyspark.sql.functions“With „explode“an array can be passed, which then returns a new line for each element of the array [26]. The corresponding code excerpt looks as following:

```
%spark2.pyspark
import pyspark.sql.functions as F
from pyspark.sql.functions import lag, col, lead, first
from pyspark.sql.window import Window
from collections import Counter

def _combine(x,y):
    """creates hourly interval for weather df"""
    print(y)
    d = str(x) + ' [5]:00:00'.format(y)
    return d

def transformWeatherDF(df, past):
    """add hours on key column and transform columoriented hours to
roworiented hours"""

    #Remove daily columns
    df = df.drop(*["Apparent Temperature (Avg)", "Daily Weather",
"Hourly Weather", "Humidity", "Windspeed"])

    #use udf to generate hours on the dates
    combine = F.udf(lambda x,y: _combine(x,y))

    #fetch relevant columns and keep column datatype
    cols_temp, dtypes = zip(*((c, t) for (c, t) in df.dtypes if c not
in ['Start Date'] and c.startswith("temp")))

    ##temperature
    #explode function for changing structure of dataframe (temperature)
    kvs = F.explode(F.array([
          F.struct(F.lit(c).alias("key"), F.col(c).alias("val")) for c
in cols_temp],)).alias("kvs")

...
    # go x hours back (temperature)
    for i in range(past):
        w = Window().partitionBy().orderBy(col("Start Date"))
        df_temp = df_temp.select("*", lag("Current Temperature",
i+1).over(w).alias("temp"+`i`))

...
```

Figure 4.4: Further transformation of weather data (excerpt)

The code example from figure 4.4 is used to transform the temperature columns, which are combined into a single column as described above. Thus the weather data frame is prepared to the extent that every hour of a day

belongs to a specific hourly weather value. The lines 35 - 39 show the use of the PySpark function „lag“, that over a selected „window“takes the starting boundary [26]. If one increments the index at this point, he will get the previous value from the window. Depending on how far one wants to go into the past, the window is moved over the data set and thus a sliding window is achieved. This can also be adapted for the future. With the function „lead“the ending boundary of a „window“is retrieved [26]. Again, the future sliding window is only applied for the usage (target variable) and not for the weather data or other columns.

The prepared hourly temperature data with sliding window looks finally as following:

Start Date	Current Temperature	temp0	temp1	temp2
2015-01-04 00:00:00	37.24	null	null	null
2015-01-04 01:00:00	36.35	37.24	null	null
2015-01-04 02:00:00	35.61	36.35	37.24	null
2015-01-04 03:00:00	34.36	35.61	36.35	37.24
2015-01-04 04:00:00	33.43	34.36	35.61	36.35
2015-01-04 05:00:00	33.14	33.43	34.36	35.61
2015-01-04 06:00:00	32.86	33.14	33.43	34.36
2015-01-04 07:00:00	32.98	32.86	33.14	33.43
2015-01-04 08:00:00	33.09	32.98	32.86	33.14
2015-01-04 09:00:00	33.65	33.09	32.98	32.86
2015-01-04 10:00:00	34.03	33.65	33.09	32.98

Figure 4.5: Temperature dataframe after transformation for 3 hours (past)

In this example from figure 4.5 it is clearly recognizable that the initial values contain „null“values, which is due to the fact that no weather data was available before 04.01.2015 (or at least they were not fetched from Dark Sky). The number of zero values increases with the number of selected hours into the past. The same applies to the last lines of the Spark data frame, where null values may be appear for the „future usage“columns. Therefore the first 20 and the last 20 lines of the data frame are removed at the end.

With the described PySpark transformations it was possible to create dynamic hourly data frames, which can then be used in the next step of the

modeling phase. A corresponding test file can be found on GitHub in the folder data preparation.

Holidays

To add the bank holidays are day-based, so they should be set on a 24 hour windows. However, fixing the bank holidays from midnight to midnight the next day might not be the best solution as people might consider the evening of a bank holiday the same way as they do for normal week day, while the evening of the day before will be more interesting. Indeed, when people want to go out in the evening and go to bed late, they will prefer doing that when they don't have to wake up early in the morning, thus they may be more likely to use a bike the end of the day before the bank holiday than of the day itself.

This is why it might be a good idea to test the accuracy of the prediction placing bank holidays from 6pm the day before to 6pm the day itself.

4.2 Learning and Prediction

For the learning part, we used the same algorithm as described in **??**. We trained twenty-four models, one per future hour to predict. For each model, we only use the past and current data which we match to the future expected number of rented bike for the given following hour, no other future data is used.

We used the sliding-windows hourly based data over 48 hours (aggregating per slice of 6 hours after the first 24 hours). We have an accuracy of 0.800638 on the test dataset. On the graph below, you can see several randomly selected days in the testing set starting at midnight and predicting for the whole day. The title above each graph indicates the date displayed, the actual expected value is in red and the predicted one is in blue.

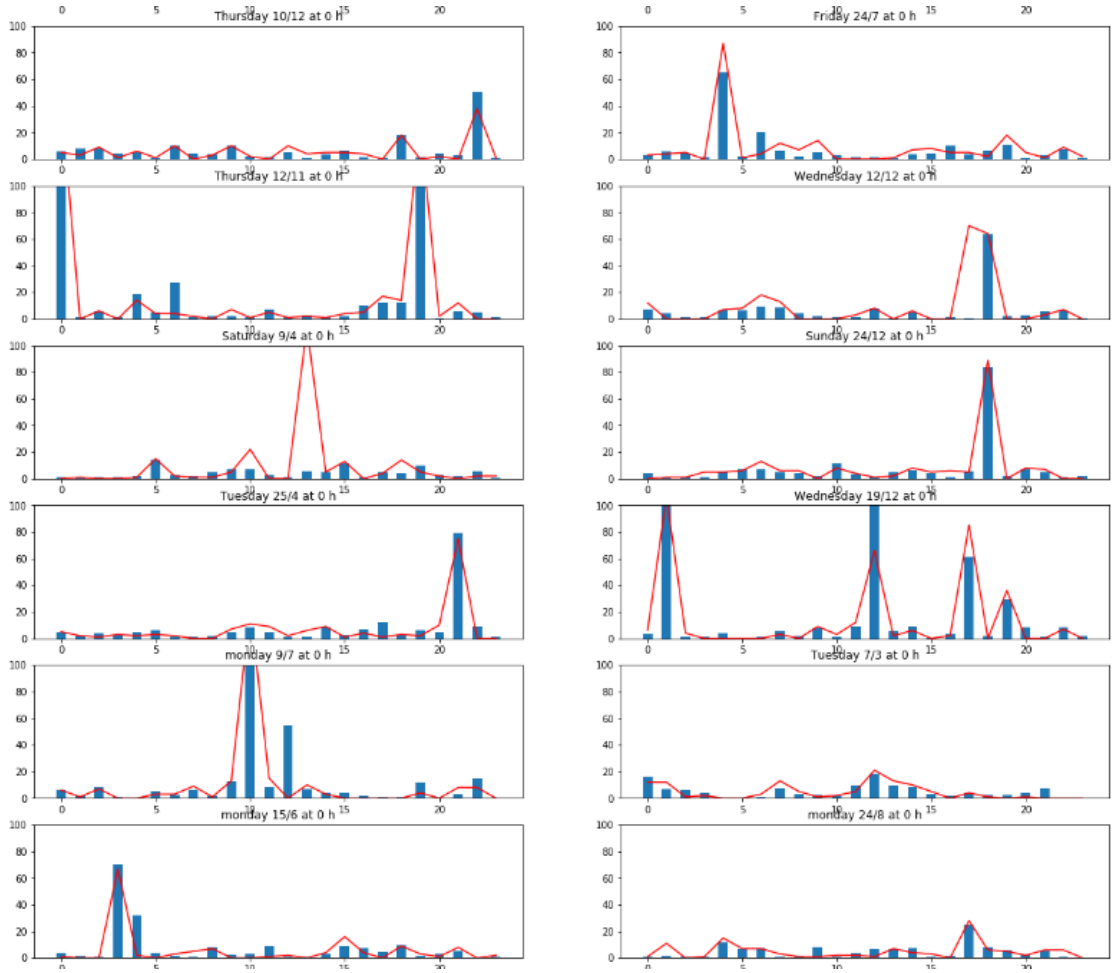

Figure 4.6: Comparison of expected (red) and predicted (blue) hourly rented bikes on test dataset

As we can see on the Figure 4.6 and considering the measured accuracy, we can see that we can predict quite well the amount of rented bikes for the following 24 hours based on the data of the previous 48 hours.

Chapter 4

Postal Code Database (Junior)

Bibliography

[1] Altoros. Hadoop Distributions: Evaluating Cloudera, Hortonworks, and MapR in Micro- benchmarks and Real-world Applications. pages pp. 1 – 65, 2013.

[2] Apache Foundation. Apache Hadoop, 2018.

[3] Cloudera. CDH Components, 2018.

[4] Cloudera. Cloudera Pricing, 2018.

[5] Cloudera. Installing CDH 5 Components, 2018.

[6] Cloudera. Operating System Requirements, 2018.

[7] D. Kumar. Since Cloudera and Hortonworks are 100% open source, can I use them freely as I would a Linux distrubution?, 2016.

[8] GeoNames. Export Zip Codes.

[9] Hortonworks. Apache Ambari Installation, 2018.

[10] Hortonworks. Grundlagen der HDP-Sandbox, 2018.

[11] Hortonworks. Starting HDP Services, 2018.

[12] Hortonworks. Support Matrix, 2018.

[13] Hortonworks. Apache ambari operations. available on: https://docs.hortonworks.com/hdpdocuments/ambari-2.6.2.2/bk_ambari-operations/content/ch_using_ambari_metrics.html. 2019.

[14] Hortonworks. Check dns and nscd. available on: https://docs.hortonworks.com/hdpdocuments/ambari-2.7.3.0/bk_ambari-installation/content/check_dns.html. 2019.

[15] I. K. Center. IBM Knowledge Center - BigInsights features and architecture, 2018.

[16] IBM Analytics. Hadoop-Testversionen, 2018.

[17] G. Kirill. Comparing the top Hadoop distributions, 2014.

[18] Ross Lydall. Boris johnson's bike hire scheme gets a £25m bonus from barclays. available on: https://web.archive.org/web/20100913111233/http://www.thisislondon.co.uk/standard/artic 23839406-boris-bike-hire-scheme-gets-a-pound-25m-bonus-from-barclays.do. 2010.

[19] M. Heo. Hortonworks vs. Cloudera vs. MapR, 2015.

[20] MapR. MapR 6.1 Documentation. 2018.

[21] MapR. MapR Editions, 2018.

[22] MapR. Operating System Support Matrix (MapR 6.x). 2018.

[23] B. Marr. Big Data: Who Are The Best Hadoop Vendors In 2017?

[24] Merv Adrian. IBM Ends Hadoop Distribution, Hortonworks Expands Hybrid Open Source, 2017.

[25] Microsoft. Manage Apache Hadoop clusters in HDInsight using Azure portal, 2018.

[26] n.d. Apache pyspark documentation. available on: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=explode. 2019.

[27] n.d. Dark sky weather api. available on: https://darksky.net/dev/docs/faq. 2019.

[28] n.d. Folium package. available on: https://python-visualization.github.io/folium/. 2019.

[29] Nominatim. OpenStreetMap Nominatim: Search, 2019.

[30] OpenStreetMap. PyrouteLib, 2017.

[31] OVirt. Virtio-SCSI, 2019.

[32] Pivotal. Overview of Apache Stack and Pivotal Components, 2018.

[33] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of the IEEE international conference on neural networks*, volume 1993, pages 586–591. San Francisco, 1993.

[34] Jeff Sposetti. Ambari blueprints. available on: https://cwiki.apache.org/confluence/display/ambari/blueprints#blueprints-step1:createblueprint. 2017.

[35] TFL. cycling.data.tfl.gov.uk, 2019.

[36] Yongqiao Wang, Lishuai Li, and Chuangyin Dang. Calibrating classification probabilities with shape-restricted polynomial regression. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[37] H. Kisker Yuhanna and D. Murphy M. Gualtieri N. Big Data Hadoop Solutions, Q1 2014. *The Forrester Wave*, 2014.

B Illustration Directory

C List of Abbreviations

YARN Yet Another Resource Negotiator 38

SQL Structured Query Language 25

NoSQL Not only Sql 25

PL/pgSQL Procedural Language/PostgreSQL 7

OSM OpenStreetMap 7

HDP Hortonworks Data Platform 38

D Attachments

Attachment 1: Responsibilities

	Sprint 1	Sprint 2	Sprint 3
Anass	hadoop docu creation of users in hadoop install nominatim/ components import planet database compute postcodes reverse geocoding in bulk	design big picture research maps for WebApp research Django	Fixing dead hadoop node installGraphhopper docu Graphhopper import OSM data to Graphhopper write report part of edited tasks
Guillaume	create user for nominatim uploading SSH keys querying centroids distances by route distances in bulk	create GitHub repos unioned all notebooks connect geopy with nominatim reserach Django Django tutorial script for initializing database initialize db with postcodes	research postcodes in PostgresDB script for filling zipcodes in db write report part of edited tasks
Kathi	creating team Box creating templates documentation OSM API geocoding/reverse Geocoding distances as the crow flies design ER model create presentation for sprint review Scrum Master tasks	design & finalize Big Picture Graphhopper docu & example install x2go server docu access vms via x2go client define mockup structure create mockup install mysql database create presentation for sprint review Scrum Master tasks	latex template for project report write report part of edited tasks union texts of team members to report reorganize GitHub repositories create presentation for sprint review Scrum Master tasks
Pascal	research hadoop distributions find weightened comparison data hadoop documentation Installation of HDP Implement Map Reduce Job	Solving Hadoop issues collect TFL data & data munging on Hadoop create mockup installed Django server	create spark file sizing notebook CycleRoutes prepared & collected plotting statistics of cycle usage solving hadoop errors plotting routes of cycle data write report part of edited tasks