Hadoop Rollout

For the Hadoop rollout 10 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).

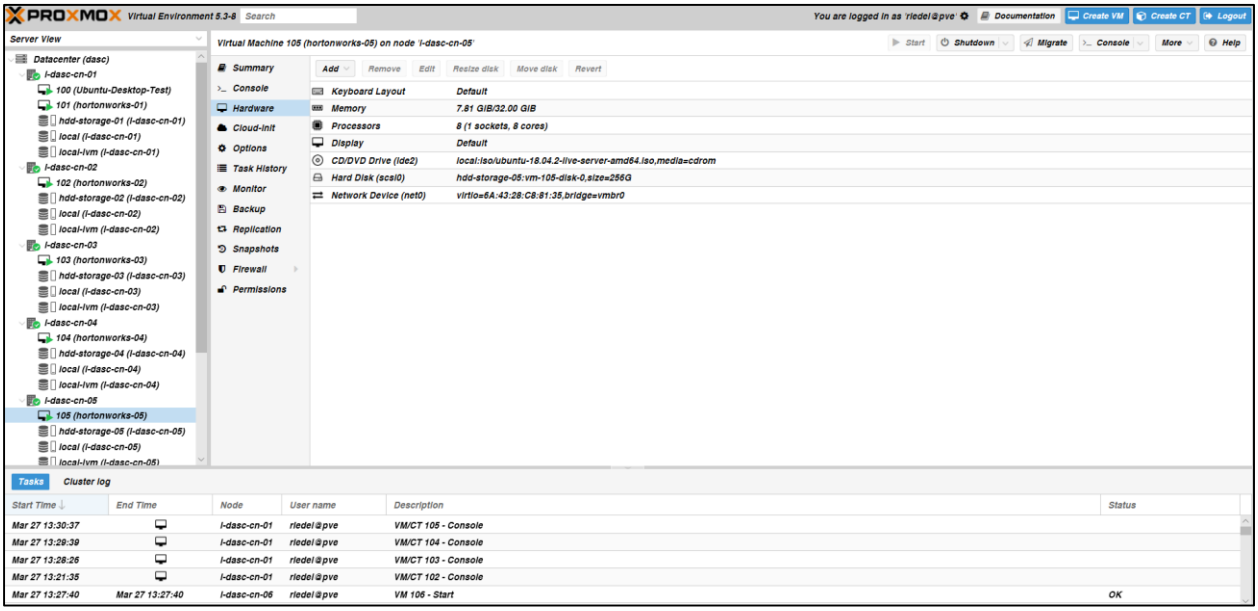


Figure 1: Proxmox surface

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 ([1]).

For the first rollout attempt, 6 cluster nodes were configured for Hadoop. The remaining 4 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 2 (p. 2) 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 ([2]).

Figure 2: 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-Addresse	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 1: 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 1) 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 Chapter 4, 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 [3]. 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 [4]. 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 2 shows.

Figure 3: 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 6 configured Hadoop nodes could be carried out successfully. These tests were based on the Zeppelin notebooks from the previous data profiling chapter, 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 (6! instead of 4) were used.

Thus the cluster (Figure 2) 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.

Data Profiling Part II

As already indicated in Data Profiling Part 1, 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 [5], the plotting of the routes on an hourly level is not performant, because too many polylines 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

```

Listing 1: Route usage over time plot

As the code from Listing 1 shows, the polylines on the map have been added iteratively. The weight parameter can be used to determine the thickness of the polyline. 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 polyline 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:

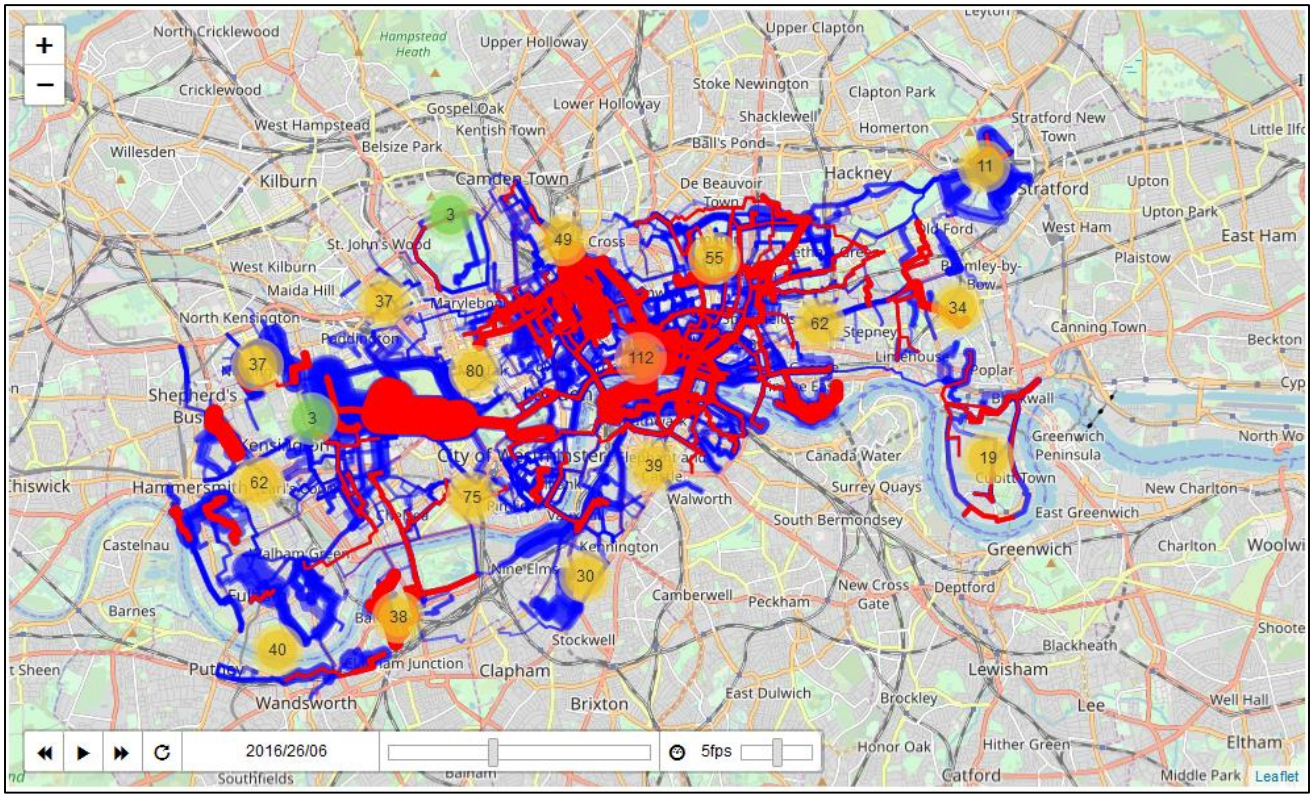


Figure 4: Used bicycle routes over time plot.

From Figure 3 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 polylines shows how often this route was used in relation to the total use of all routes. The red colored routes from Figure 3 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), 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), the inner districts connected by the Waterloo Bridge and the London Blackfriars Bridge form the centre 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, but not for today or in 2 years. A useful feature that comes along with the folium plugin is the automatic data display sequence [5]. When someone clicks on the "Play" button from Figure 3, 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:

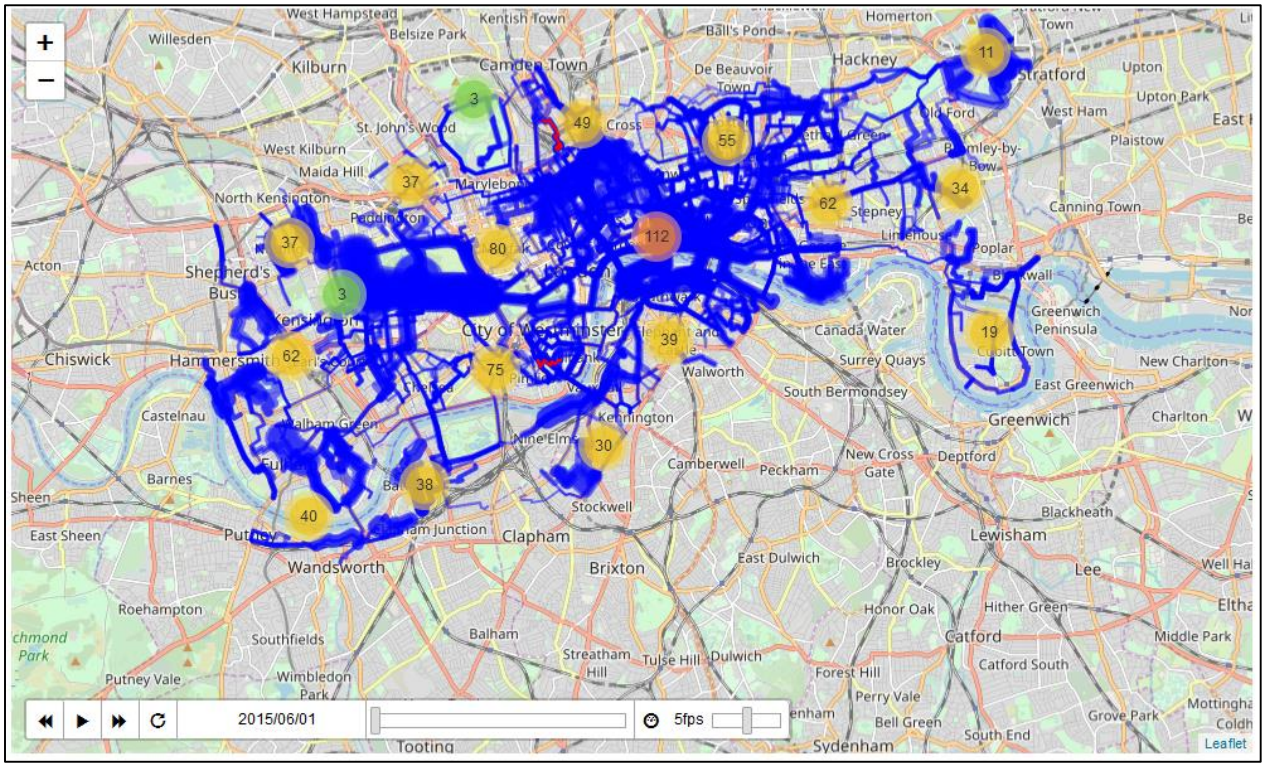


Figure 5: Used bicycle routes over time plot.

On 06.01.2015 obviously less bicycles were rented than in Figure 3. 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 [6]. 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 (http://i-hadoop-01.informatik.hs-ulm.de/routes_frequency.html).

Overall, it can be observed that the usage of the rental stations has risen sharply on average over the last 4 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.

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 [7]. 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 2 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)


```

Listing 2: Fetching the weather API with Python

As the code from Listing 2 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 [7]. 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 6: Prepared data frame of Kings Cross (daily)

As one can see from Figure 5, the past Data were 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 modelling section.

With the processed bicycle data from Figure 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.

Data Preparation – Hourly Base

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“ to manage "big data" transformations. Unfortunately the behaviour 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 columnwise operations. Moreover, Pandas operations 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"))
…
```

Listing 3: Transformation of humidity columns (excerpt)

The excerpt from Listing 3 shows that "structTypes" can be used to search the individual sublists 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 2 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 7: Weather dataframe after transformation for 2 hours

This (Figure 6), 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 6 can be used as ground for applying sliding window method.

However, before the sliding window procedure could be implemented, the "Start Date" from Figure 6 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 easily. The missing hours can be determined by a user defined function. Since every day has 24 hours, this can be calculated manually as following (Listing 4):

```
%spark2.pyspark
#fill empty or missing hours with 0, supplement missing hours (e.g. 1,
8 -> supplement 2,3,4...)
from pyspark.sql.functions import col, min as min_, max as max_

step = 60 * 60

minp, maxp = new_df.select(
    min_("New Date").cast("long"), max_("New Date").cast("long")
).first()

reference = spark.range(
    (minp / step) * step, ((maxp / step) + 1) * step, step
).select(col("id").cast("timestamp").alias("New Date"))

```

Listing 4: Adding missing hours on usage data

A prerequisite of this method from Listing 4 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 [8]. The corresponding code excerpt looks as following:

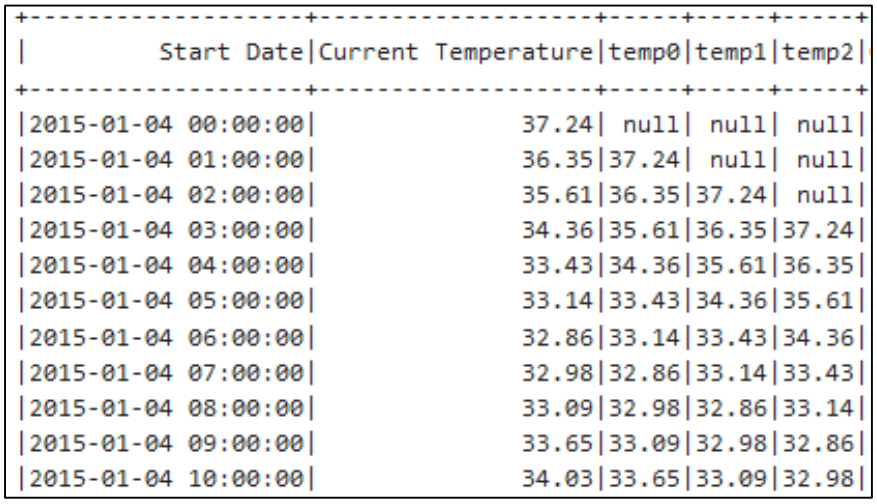
```
1   %spark2.pyspark
2   import pyspark.sql.functions as F
3   from pyspark.sql.functions import lag, col, lead, first
4   from pyspark.sql.window import Window
5   from collections import Counter
6
7   def _combine(x,y):
8       """creates hourly interval for weather df"""
9       print(y)
10      d = str(x) + ' [5]:00:00'.format(y)
11      return d
12
13  def transformWeatherDF(df, past):
14      """add hours on key column and transform columoriented hours to
15  roworiented hours"""
16
17      #Remove daily columns
18      df = df.drop(*["Apparent Temperature (Avg)", "Daily Weather",
19  "Hourly Weather", "Humidity", "Windspeed"])
20
21      #use udf to generate hours on the dates
22      combine = F.udf(lambda x,y: _combine(x,y))
23
24      #fetch relevant columns and keep column datatype
25      cols_temp, dtypes = zip(*((c, t) for (c, t) in df.dtypes if c not
26  in ['Start Date'] and c.startswith("temp")))
27
28      ##temperature
29      #explode function for changing structure of dataframe (temperature)
30      kvs = F.explode(F.array([
31            F.struct(F.lit(c).alias("key"), F.col(c).alias("val")) for c
32  in cols_temp],)).alias("kvs")
33
34  ...
35      # go x hours back (temperature)
36      for i in range(past):
37          w = Window().partitionBy().orderBy(col("Start Date"))
38          df_temp = df_temp.select("*", lag("Current Temperature",
39  i+1).over(w).alias("temp"+`i`))
40
48  ...
```

Listing 5: Further transformation of weather data (excerpt)

The code example from Listing 5 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 [8]. 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 [8]. 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 8: Temperature dataframe after transformation for 3 hours (past)

In this example from Figure 7 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.

Modelling (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 [9]. In fact, it is just a modified version of a linear regression. This means the independent variable x and the dependent variable y is modelled as an nth degress (so-called polynomial) in x [9].

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 scitkit-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 RSME 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 8 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 8.

Figure 9 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.

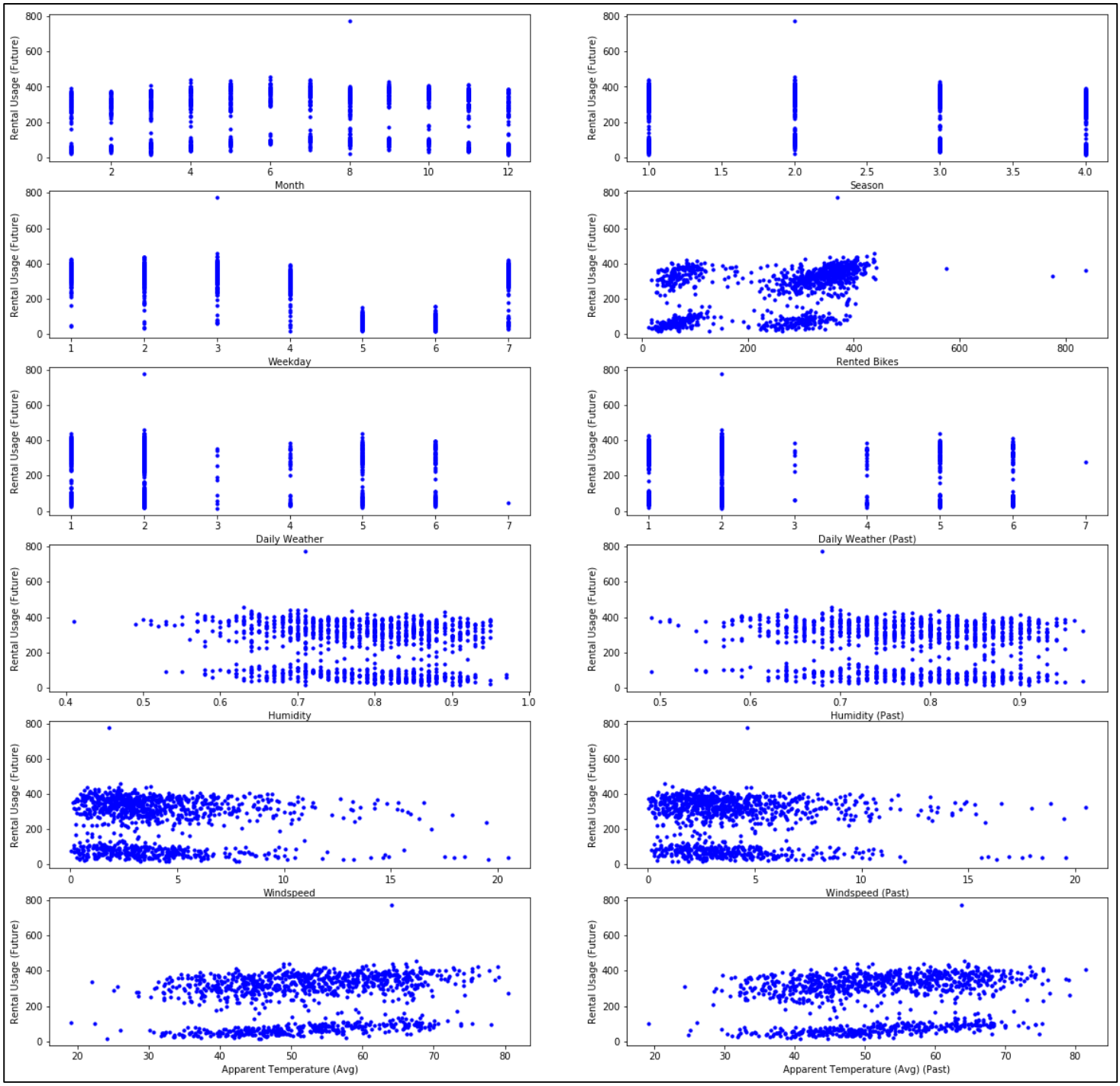


Figure 9: Plot of polynomial regression (features)

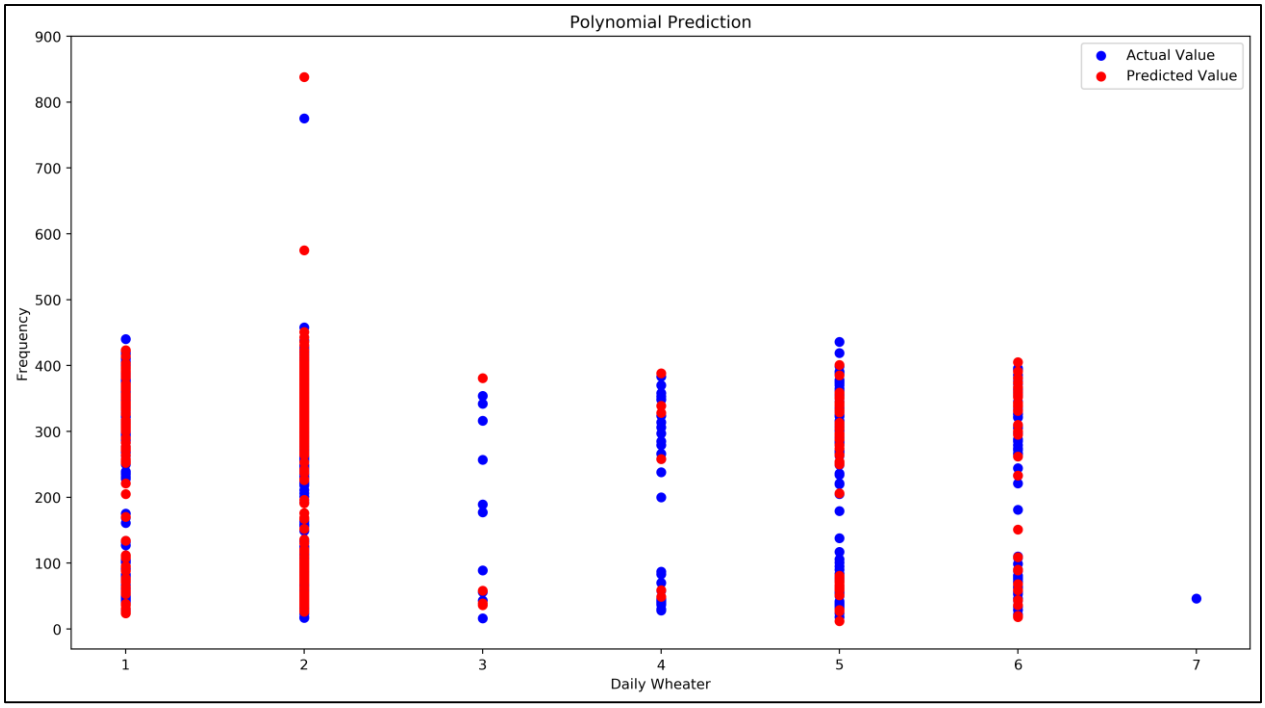


Figure 10: Polynomial Prediction of rental usage on daily weather

Sources

[1] oVirt, "Virtio-SCSI. Available on: https://ovirt.org/develop/release-management/features/storage/virtio-scsi.html," ed, 2019.

[2] 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.

[3] J. Sposetti, "Ambari Blueprints. Available on: https://cwiki.apache.org/confluence/display/AMBARI/Blueprints#Blueprints-Step1:CreateBlueprint," 2017.

[4] 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.

[5] n.d., "Folium package. Available on: https://python-visualization.github.io/folium/," 2019.

[6] R. 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/article-23839406-boris-bike-hire-scheme-gets-a-pound-25m-bonus-from-barclays.do," 2010.

[7] n.d., "Dark Sky Weather API. Available on: https://darksky.net/dev/docs/faq," 2019.

[8] n.d., "Apache PySpark Documentation. Available on: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=explode," 2019.

[9] Y. Wang, L. Li, and C. Dang, "Calibrating Classification Probabilities with Shape-restricted Polynomial Regression," *IEEE transactions on pattern analysis and machine intelligence,* 2019.