Malivai Luce
M.Sc. in Transportation Engineering
15-077 Statistical Learning and Data Mining

FINAL PROJECT REPORT

Predicting full and empty stations for the operations of bike-sharing systems

1. Motivations

Bicycle sharing systems represent a new opportunity to improve mobility within cities and were largely adopted by most major cities around the world for the past decade. Today, the largest scheme is located in Wuhan, China (with more than 90,000 bikes) and the densest system is offered by Paris, with 1 bike per 97 inhabitants. Having reached that scale, bicycle sharing schemes are nowadays critical assets and a non-negligible component of cities' transportation systems, and tens of thousands trips are made each day on public bikes.

To be successful, a public bicycle scheme must be reliable for users: when they plan on going from a point to another in a city, they must be able to find a station with a bike close to their origin and a station with a free docking point close to their destination. The service quality, besides bikes' conditions, is directly linked to the experienced difficulty of finding such stations, and completely full or completely empty stations are the nightmares of both operators and users.

Figure 1: Users waiting for a free spot at a full station (top left) and an empty station (top right) in Paris. A Vélib' truck is moving bikes in Paris (bottom).

As the flux of bikes is polarized during the day, the system does not auto-regulate and operators need to move bikes from stations to stations to ensure availability of both bikes and free docking points. To better plan these operations and to have a better chance to improve the user

experience, knowing the stations likely to become full and the ones likely to become empty is essential to target transfers from the former to the latter.

The objective of this project is therefore to study historical data of Vélib, the Paris bicycle sharing system, in order to be able to predict which stations will be full or empty within the next hour, at any time of the day given the observed crowding at that time. The project is structured in three main parts: the collection and preprocessing of data from Velib', the clustering of stations and the development of prediction models. This report will follow this agenda.

2. Data pre-processing

The choice of Vélib' for this project answers to two imperatives: the availability of data and the large scale of the studied system. JCDecaux, the company that operates the bicycle sharing system in Paris, has implemented an OpenData access that allows any internet users to collect live data on the stations through an API[1]. Targeting a large scale public bicycle scheme allows to collect an important amount of data in a small time and guarantees quality of the data, in the sense that the noise of non-significant trips will be smaller. Paris' system also meets that criteria, with 1,229 stations distributed across the city and an important number of trips each day.

a. Data collection

The data were collected between Monday 25 and Friday 29, April 2016. The days chosen for the study correspond to week days and the data are thus expected to present a certain homogeneity because of the regularity of users' behavior during weekdays. The data were collected from 4am to 4am for each day to capture both early morning and late-night trips. Therefore, four days are included in the data: Monday to Thursday (actually finishing on Friday at 4am). Friday was excluded from the study since the behavior on this day just before the weekend are likely to be different than during the rest of the week, especially during the night from Friday to Saturday.

The data are of two kinds: static data are collected once to gather general information on the station, while dynamic data are updated every minute and provide the current status of the stations. The table in figure 2 describes the information available for each type. A bonus is awarded to users that drop their bikes at station located on hills: in this case, the zero-charge rental period is extended from 30min to 45min.

	Variable	Description
Static	**number**	*Number of the station. Equivalent to an ID number.*
	name	*Name of the staion*
	address	*Postal adress of the station*
	position	*GPS coordinates (WGS84 format)*
	banking	*Indicates whether the station has a payment terminal*
	bonus	*Indicates whether the station awards a bonus to users*
Dynamic	**status**	*Indicates the status of the station (CLOSED or OPEN)*
	bike_stands	*Number of fonctional docking points in the station*
	available_bike_stands	*Number of docking points available for parking*
	available_bikes	*Number of available bikes*
	last_update	*Timestamp in milliseconds since Epoch indicating the last update for the specific station*

Figure 2: Variables available through JCDecaux API and their descriptions.

During the collection of the dynamic data, to reduce the size of the file, every minute only information on stations for which the 'last-update' had change since the previous time were

[1] https://developer.jcdecaux.com/#/opendata/vls?page=getstarted

actually downloaded. The four-day worth of data file is still massive at the end, and it comprises of 1,341,994 lines. To be able to exploit these data, a preprocessing phase is needed.

b. Data pre-processing

The data were then processed to create the time profile of each station. We calculated the occupancy rate (number of available bikes divided by the number of bike stands expressed as a percentage) every five minutes for each station, yielding 1153 points between Monday 4am and Friday 4 am. We then added the banking and bonus static information as well as the number of docking stands to complete the description of the station. This first set of data is called *'profiles'*.

Station number	Bike stands	Banking	Bonus	t	t+5	t+10	etc.									
31705	50	1	1	18.00	18.00	18.00	18.00	20.00	20.00	22.00	22.00	22.00	24.00	24.00	24.00	...
10042	33	1	0	36.36	36.36	36.36	36.36	36.36	36.36	36.36	36.36	36.36	36.36	36.36	39.39	...
8020	44	1	1	13.64	13.64	13.64	11.36	11.36	11.36	11.36	11.36	11.36	11.36	11.36	9.09	...
1022	37	1	0	29.73	29.73	29.73	32.43	32.43	29.73	29.73	29.73	29.73	29.73	29.73	29.73	...
35014	25	1	0	24.00	24.00	24.00	24.00	24.00	24.00	24.00	24.00	24.00	24.00	24.00	24.00	...
20040	26	1	0	3.85	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.85	3.85	0.00	...
28002	60	1	0	35.00	35.00	35.00	35.00	35.00	35.00	35.00	35.00	35.00	35.00	35.00	35.00	...
15111	24	1	0	58.33	58.33	62.50	62.50	62.50	62.50	62.50	62.50	62.50	62.50	62.50	58.33	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 3: Structure of the 'profiles' file.

Another transformation is then operated on the profiles to create the data points that will be used in the prediction models. At each hour, the occupancy rate is noted and the following hour is observed to create the Full/Empty variable. If within the next hour the occupancy rate reaches 100%, then the Full/Empty variable is set to 1. If it falls down to 0%, then the Full/Empty variable is set to -1. If none of those cases occur, the variable is set to 0. Proceeding this way for every station and every hour of the studied period, we obtain 119,213 sample points, that we call *'datapoints'* further on.

Station number	Time of day	Occupancy rate	Full/ Empty
10042	6	36.363636	0
10042	7	36.363636	0
10042	8	54.545455	0
10042	9	63.636364	0
10042	10	90.909091	1
10042	11	90.909091	0
10042	12	90.909091	1
...	...	...	...

Figure 4: Structure of the 'datapoints' file.

These created two sets of data will be the basis of the next phases: clustering with the 'profiles' and predicting with the 'datapoints'.

3. Clustering

In this phase, we attempt to regroup stations that share similar characteristics together, with the idea that in the prediction phase knowing which category a station belongs to will improve the ability of the model to predict whether it will be full or empty or neither one nor the other.

a. Clustering with standardization

We performed a hierarchical clustering of stations with Euclidean distance. Beforehand, we normalized the variables by dividing by their sample variances. In this case, all variables have the same influence on the clustering process, and binary variables such as Banking and Bonus are not penalized compare to the occupancy rates (varying from 0 to 100) or the number of docking points. The dendrogram presented in Figure 5 is obtained.

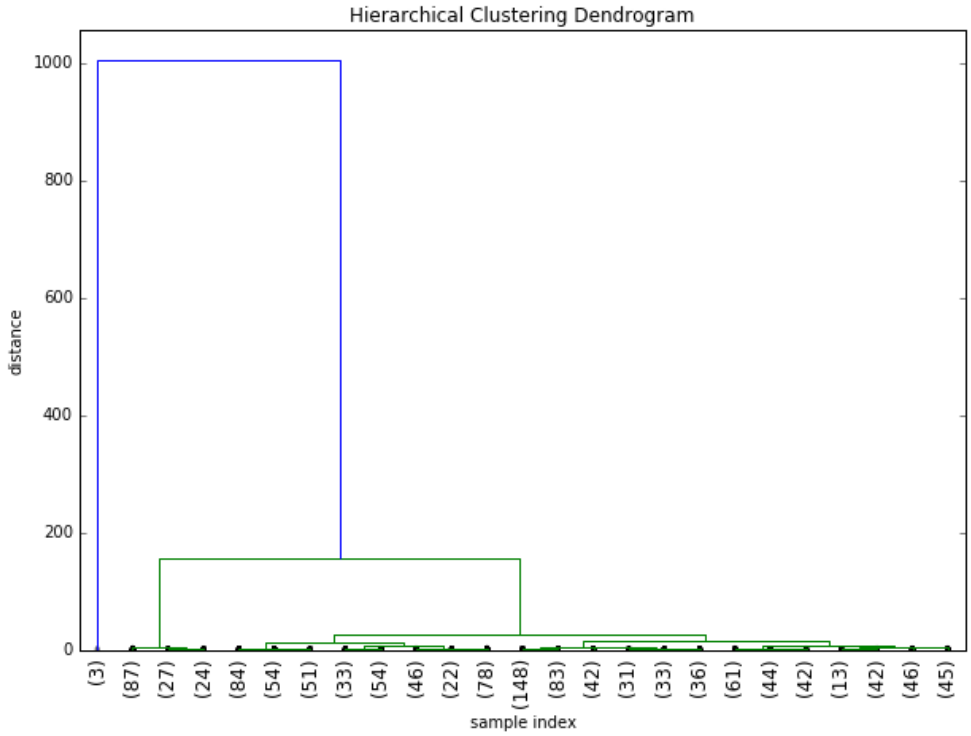

Figure 5: Dendrogram of the hierarchical clustering with standardized data.

It is not easy to differentiate clear clusters, but we can identify three close ones (in green) and a last one formed of only 3 stations, that may be considered as outliers. By analogy with the clustering without standardization, we will define 4 clusters. If we look at the centroids of these clusters (Figure 6), it seems that the stations of Cluster 2 are awarding a bonus, while the stations of Cluster 1 do not offer banking payment terminals. The number of bike stands vary also from clusters to clusters.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Size	3	138	422	666
Stands	21.3	29.3	35	32.1
Banking	0	1	1	1
Bonus	0	1	0	0

Figure 6: Centroids' characteristics and size of each cluster.

When we look at the average time profile within each cluster (Figure 7), clusters 1 and 3 on one hand and clusters 2 and 4 on the other hand have similar profiles: 1 and 3 have high occupancy rates during the day and low during the night, whereas it is the opposite for 2 and 4. Cluster 1 have relatively high occupancy rates throughout the whole period, while clusters 2 and 4 have relatively low occupancy rates.

The clusters described in this part are not really convincing since they do not allow to differentiate clearly the characteristics of each group. It appears that the banking and bonus variables have actually influenced the separation of clusters too much, and to limit their importance, it seems a good idea to look at clustering without standardization. The fact that the occupancy rate expressed as a percentage is already normalizing the differences between the stations (the number of bike stands does not favor big stations over the small ones) is also supporting this idea.

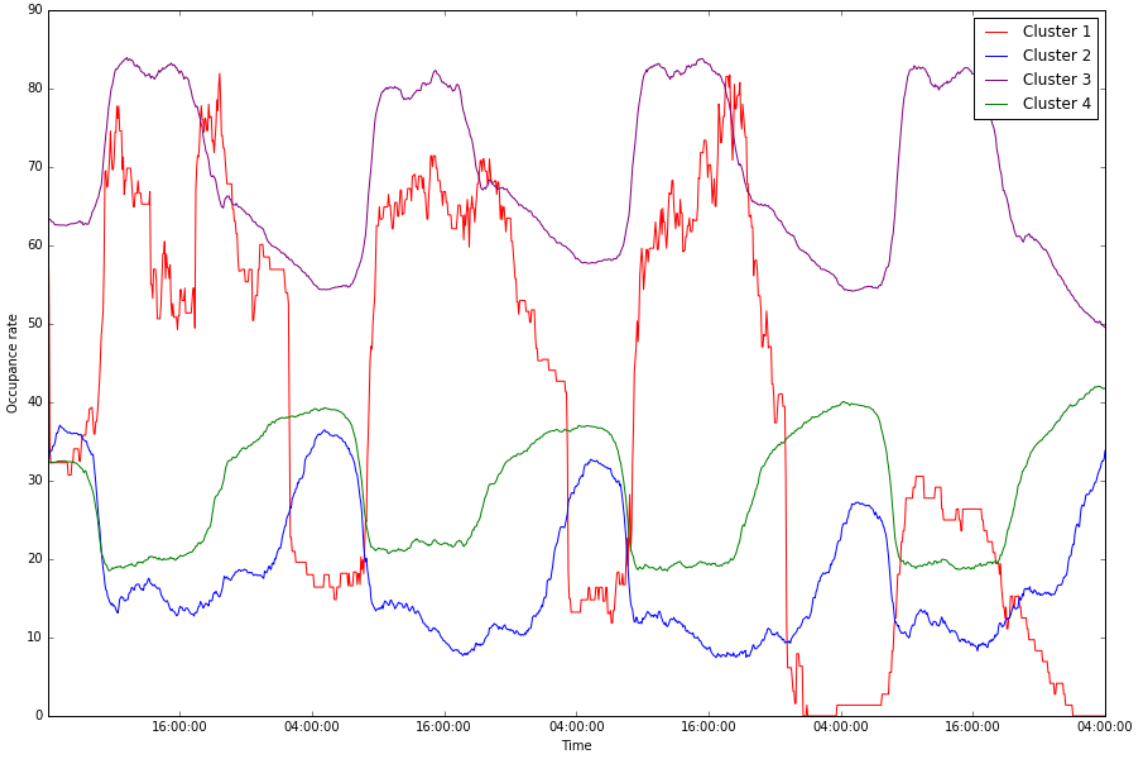

Figure 7: Average time profile of the stations within each cluster

b. Clustering without standardization

The same methods were applied, except that this time data were not standardized beforehand. We obtain the dendrogram shown in figure 8, suggesting the existence of 4 different clusters.

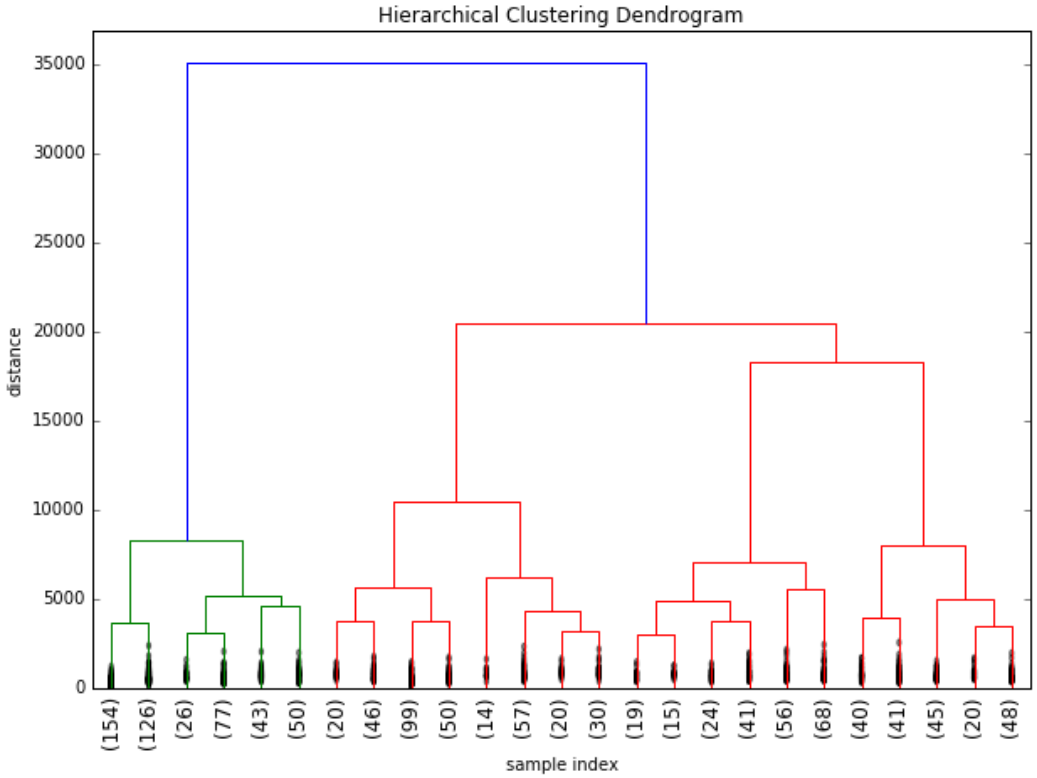

Figure 8: Dendrogram of the hierarchical clustering with non-standardized data.

The centroids of each clustered is given in Figure 9. In this case, no cluster has a significantly small or big size and all of them gather hundreds of stations. The number of docking points doesn't seem to be relevant, and in all clusters, most of the stations offer a banking terminal. It is however to be noted that stations in cluster 1 are more likely to award a bonus.

	Cluster 1	Cluster 4	Cluster 3	Cluster 4
Size	476	336	223	194
Stands	32	34.9	31.4	32.4
Banking	0.998	0.997	1	0.995
Bonus	0.237	0.003	0.045	0.072

Figure 9: Centroids' characteristics and size of each cluster.

The separation of the different groups does not seem to be performed on this variable. If we take a look at the average time profiles within each clusters (Figure 10), it appears to be the main driver in the clustering algorithm. Cluster 1 regroup stations for which the occupancy rate remains low throughout the whole day, with a peak during the night, while cluster 2 regroup stations for which it stays high, with a peak during the day. It is interesting to observe that stations with bonus are more likely to have a low occupancy rate. Clusters 3 and 4 have a stronger amplitude in the variations of their occupancy rate. The stations of cluster 3 are rather full at night and empty during the day, and one can suppose that they are located in neighborhoods dominated by residential housing. On the other hand, the stations of cluster 4 are rather full during the day and empty at night, suggesting that they are situated in business oriented neighborhoods.

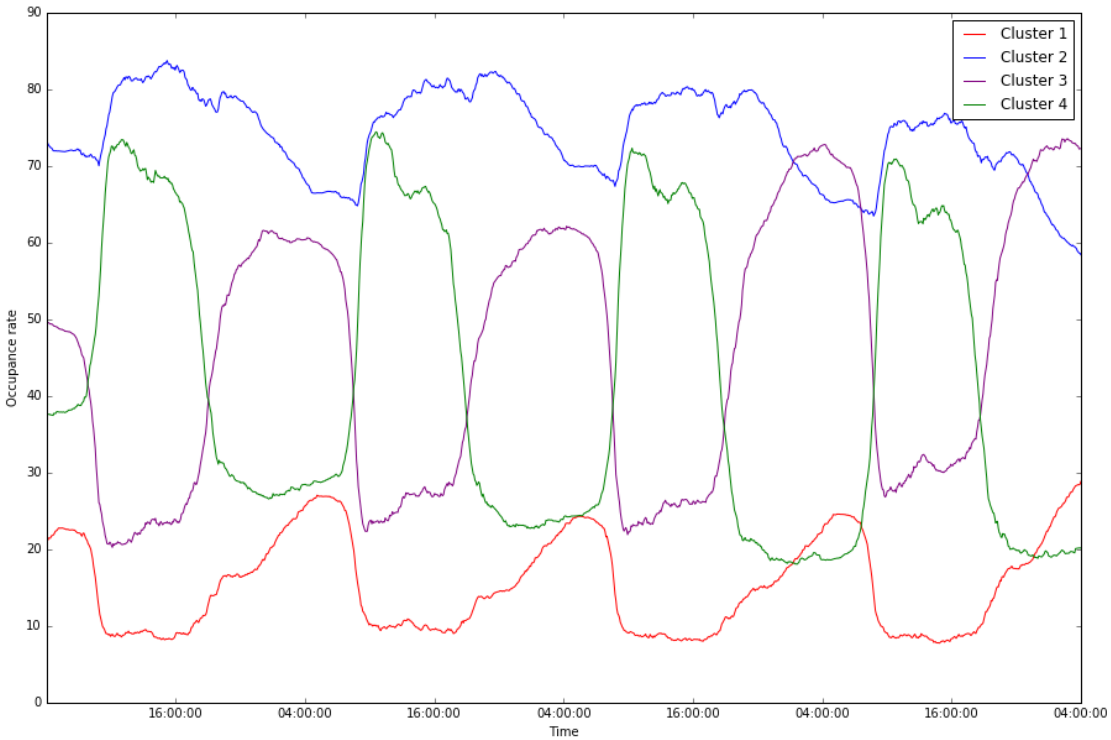

Figure 10: Average time profile of the stations within each cluster

These assumptions can be verified by plotting the distribution of stations of each cluster across the city. We produced a '.kml' file with the GPS coordinates of every stations and attributed to their placemark the color of their cluster: red for cluster 1, blue for cluster 2, green for cluster 4,

but we replaced purple for cluster 3 by yellow so that it will be more distinctive compared to blue. We obtain the map presented in Figure 11, where it can be seen that clusters are spatially separated.

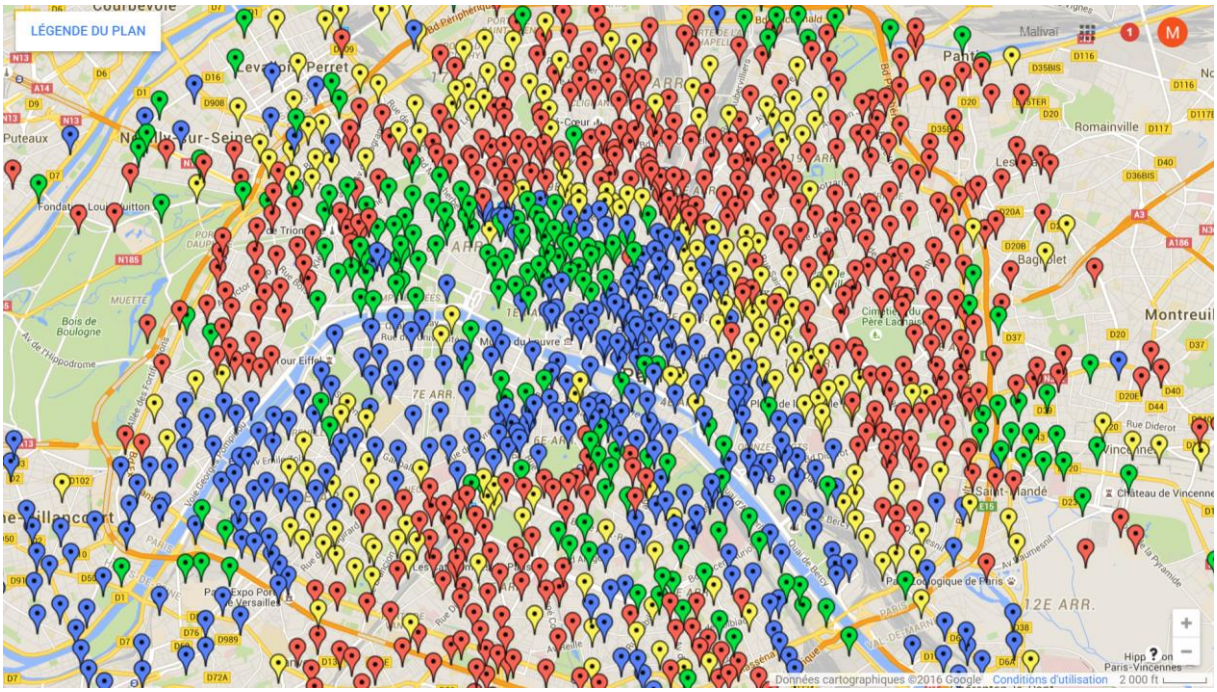

Figure 11: Distribution of stations across the city and by cluster.

Cluster 1 in red corresponds to stations located in the periphery of the city and that serve the hills of the northern and eastern parts of Paris: Montmartre, Belleville, etc. It explains why these stations are usually rather empty. Cluster 2 in blue gathers stations situated in the center of the city and the touristic area along the Seine river, as well as the business suburb cities of Boulogne-Billancourt and Issy-les-Moulineaux in the southwest. They are thus almost always well garnished with a lot of bikes. Cluster 3 in yellow is concentrated in specific area such as the 11th arrondissement in the east of the capital and the 15th arrondissement in the southwest, which are rather residential area and place to go out at night. It is therefore not surprising that the stations of this cluster experience low occupancy rate during the day and are rather full at night. At last, stations from Cluster 4 are mostly located in the 8th, 9th and 2nd arrondissement of Paris on the right bank: there are the neighborhoods with a lot a company headquarters and the biggest department store of the capital along Boulevard Haussmann and the Champs-Elysées. We also observe a fair amount of green stations serving the Latin Quarter and its universities. It is then logical that their occupancy rate increases during the day and drops down at night.

c. Conclusion

The results given by the hierarchical clustering without standardization provide a better picture at the different time profile of occupancy rates that exists across Parisian stations, and these distinctions seem to be of higher interest than those based on banking or bonuses to predict the likelihood that a station will become empty or full at a point in time. Therefore, we associate each station with its cluster as defined by the method without standardization, and add this information into the dataset 'dataset' so that it helps developing the prediction models.

4. Neural Network

To take into account the different parameters of different nature that are the current occupancy rate, the affiliation to a cluster and the time of the day, and combine them to provide a single output – the categorical variable Full/Empty – neural networks appear to be a good option. Before conceiving and training one, the data from 'datapoints' were a slightly modified to enhance the property of neural networks.

The time of the day and the occupancy rates were scaled so that their values range from 0 to 1. Therefore, the variable representing the time of the day is the fraction of the day already gone by, and the variable representing the occupancy rate is the fraction of the docking points that are occupied by bikes. The affiliation to clusters were codified by 4 binary variables X1, X2, X3, X4, such that Xi equals to 1 if the station belongs to cluster i, and 0 otherwise. At last, the Full/Empty output variable was also transformed into three categorical binary variables Y0, Y1, Y-1 indicating that during the next hour the station will become respectively neither full nor empty (Y0 = 1), full (Y1 = 1) and empty (Y-1 = 1).

Two subsets were created: a training set with 23,961 data points and a test set with 11,802 data points. A neural network with a single hidden layer has been trained with these inputs and outputs, but the optimal number of neurons to be included in the intermediate layer and the optimal value of the learning rate remain to be determined.

a. Setting the parameters of the neural network

After several 'blind' tests, the values chosen for the parameters allow the neural networks to provide acceptable results. With 10 neurons in the hidden layer and a learning rate of 0.00002, the network training was working. To improve the results, we fixed one of them and made the other vary around the determined value to observe the effect on the errors and choose the 'optimal' value.

i. Choice of a learning rate

We plotted the error on the training set after 10,000 iterations of the backprogation algorithm as well as the error on the test set made when using the weights and biases obtained from the training for different values of the learning rate, ranging from 0.00001 to 0.5 (Figure 12). The number of neurons in the hidden layer was set to 10 in this part. One point was very high in the training errors, these is due to a computational issue that I was not able to correct – some values taken by the exponential in the sigmoid function were so low that they were stored as NaN (not a number) – and should be discarded.

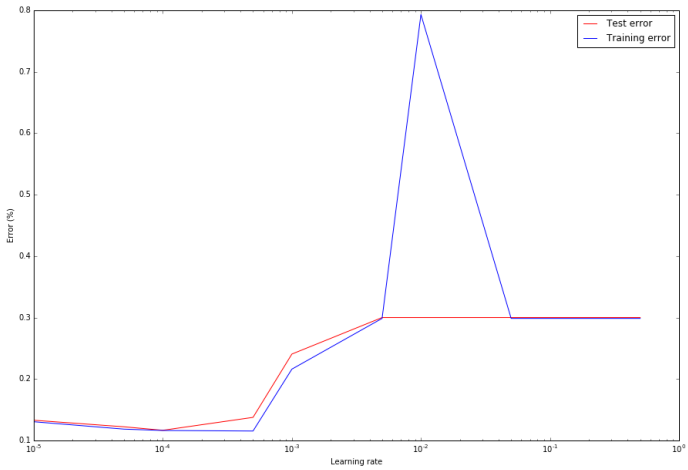

Figure 12: Training (blue) and Test (red) errors against learning rate.

It is worth noticing that the test error is very close to the training error. Overall, the best learning rate seems to be somewhere between 0.0001 and 0.001, and thus it was set to 0.0005 for the rest of the project.

ii. Choice of the number of neurons in the hidden layer

We plotted the error on the training set after 10,000 iterations of the backprogation algorithm as well as the error on the test set made when using the weights and biases obtained from the training for different numbers of neurons in the hidden layer, ranging from 5 to 55 (Figure 13). The learning rate was set to 0.00002 in this part.

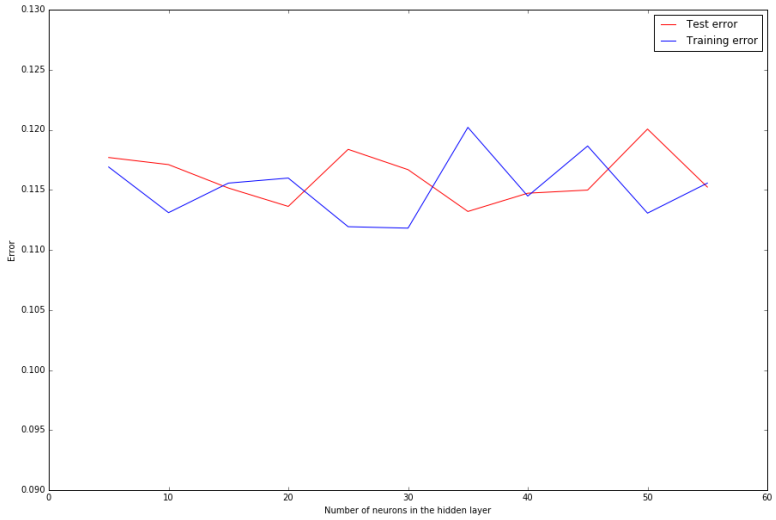

Figure 13: Training (blue) and Test (red) errors against the number of neurons in the hidden layer.

The two errors are still very close, and they do not vary significantly with the number of neurons. Overall, it appears that a small number of neurons is sufficient to fit the data in the training phase, and therefore the 'optimal' value for the number of neurons in the hidden layer was set to 5.

b. Analysis of the results

Now that the parameters are set, we perform 60,000 iterations of the backpropagation algorithm to train the neural network on the training data set. We can then use it to predict the Full/Empty variable on the training dataset and on the test dataset. The results are given in Figure 14. We obtain a score of 88.65% of good predictions on the training dataset, and of 88.19% on the test dataset.

Desired	Computed			
	0	1 (Full)	-1 (Empty)	TOTAL
0	16648	84	73	16805
1 (Full)	838	1361	2	2201
-1 (Empty)	1723	0	3232	4955
TOTAL	19209	1445	3307	23961

Desired	Computed			
	0	1 (Full)	-1 (Empty)	TOTAL
0	7686	149	426	8261
1 (Full)	361	749	3	1113
-1 (Empty)	455	0	1973	2428
TOTAL	8502	898	2402	11802

Figure 14: Confusion matrices for the training dataset (left) and the test dataset (right).

First of all, the network makes a good distinction between stations that are going to be empty and those that are going to be full: only 3 stations of the test dataset that actually were going to be full were classified as becoming empty, and none were misclassified in the reversed situation. The risk of degrading the situation rather than improving it – by bringing bikes to already crowded stations or removing bikes from emptying stations – is therefore quasi null.

To better assess the performance of the neural network, let's take a look at the false positive and false negative cases of the test dataset results. In 83.41% of cases where the prediction was "Full", and in 82.14% of cases where the prediction was "Empty", the network was right. Overall, false "Full" or "Empty" positive occur in 17.51% of cases. Similarly, in 67.30% of cases where the station was to become full, and in 81.26% of cases where the station was to become empty, the network predicted it right. We observe an asymmetry this time between detecting the likelihood of becoming full and of becoming empty: it seems that the network is too optimistic regarding the rate at which stations are filled, and misses a non-negligible fraction of stations that are to be complete. Overall, the false negative is still acceptable at 23.13%.

Overall the neural network provides quite good information on the evolution of the system with a one-hour horizon. In about 20% of the case, it raises concerns on stations that do not necessitate any attention in the end, but including them in the process will not harm the system but it will merely be responsible of additional transfers that will are not useful: it is a source of inefficiency in the operations. However, for about 80% of stations marked as critical by the neural network, the transfers will be correctly and efficiently targeted, which represent already an important step toward an optimal management of the bikes' distribution across the system.

5. Conclusions and further work

The model developed in this project is very simple: it relies on only three observations to make a decision on whether a station will become full, empty or neither one nor the other. Those three observations are the time of the day, the current occupancy rate and a static affiliation to one of the 4 clusters of stations defined for Paris. The neural network after training captures rather well the interaction between the three variables and yields a prediction score of about 88%. As a consequence of the simplicity of the model, the structure of the neural network is kept light: it comprises of only one hidden layer with 5 neurons.

The introduction of additional explanatory variables could help decreasing the error rate. If some geographical considerations are already implicitly embedded in the affiliation to a cluster, one can expect to have dynamic correlations between stations that are close to each other. Adding a mean value of the occupancy rate over the stations located in a perimeter around the studied station would allow the neural network to incorporate those effects in the model. Another idea would be to look for external factors and other data sources, and in particular for bicycle sharing scheme, one might want to include data on the weather as the ridership in the system can vary a lot depending on the meteorological conditions. At last, some changes to the model itself can be made, more specifically taking into account the time series nature of the problem might also help improving the results.

Being able to correctly predict the occupancy rate at a station given a certain horizon is beneficial to both the operator of the system and the users. The former will plan the transfers of bikes from stations to stations more optimally which will result in economies in operating expenses and in a higher service quality for the riders. If the information is also made available to users, they will be able to plan their trips with less uncertainty and choose the station the most likely to offer available bikes or docking points in a perimeter around their origin or destination. Their perception of the system quality will be higher and it might also help reducing the need of transfers as informed users will unconsciously participate in a form of auto-regulation of the system by targeting stations with less risks for them.

```
# -*- coding: utf-8 -*-
"""
ANNEX 1:

Downloading Data from JCDecaux API with an update every minute.
"""

import os
import requests
import datetime
import time

#%% Define working environment

workdir = os.getcwd()
root = os.path.abspath(os.path.join(workdir, os.pardir))
datadir = os.path.join(root,"Data")

class CET(datetime.tzinfo):
    def utcoffset(self, dt):
      return datetime.timedelta(hours=2)

    def dst(self, dt):
        return datetime.timedelta(0)

contracts_query = "https://api.jcdecaux.com/vls/v1/contracts"
contracts_params = {"apiKey": "**********************************"}

stations_query = "https://api.jcdecaux.com/vls/v1/stations"
stations_params = {
    "apiKey": "87a46ce4bef35f8bf470730e15b2ffcfde881849",
    "contract": "Paris",
    }

head = {"Accept": "application/json"}

#%% Define functions

def createLastUpdate():

    s = open("stations.csv", 'r')
    lines = s.readlines()

    lastupdates = {}

    for line in lines:
        entry = line.rstrip('\n').split(',')
        number = int(entry[0])
        lastupdate = entry[5]
        lastupdates[number]= lastupdate

    return lastupdates

def getUpdate(f, lastupdates):

    now = datetime.datetime.now(CET())
```

```
    now = datetime.datetime.strftime(now, "%Y-%m-%d %H:%M:%S")
    r = requests.get(stations_query, params = stations_params, headers=head)
    response = r.json()

    k = 0
    for station in response:
        nb = station["number"]
        update = station["last_update"]
        update = datetime.datetime.fromtimestamp(int(update)/1000,CET())
        update = datetime.datetime.strftime(update, "%Y-%m-%d %H:%M:%S")
        if update != lastupdates[nb]:
            k+=1
            lastupdates[nb] = update
            status = station["status"]
            stands = station["bike_stands"]
            availstands = station["available_bike_stands"]
            availbikes = station["available_bikes"]
            f.write("{},{},{},{},{},{}\n".format(nb,update,status,stands,availbikes,ava

    print("{}: {} updates".format(now, k))

    return lastupdates

#%% Download data

f = open(os.path.join(datadir, "week1.csv"), 'w')

starttime = datetime.datetime.strptime("2016-04-25 04:00:00", "%Y-%m-%d %H:%M:%S")
starttime = starttime.replace(tzinfo=CET())
endtime = datetime.datetime.strptime("2016-04-30 04:00:00", "%Y-%m-%d %H:%M:%S")
endtime = endtime.replace(tzinfo=CET())

now = datetime.datetime.now(CET())
lastupdates = createLastUpdate()

while(now <= starttime):
    print("Waiting {}".format(starttime))
    time.sleep(60)
    now = datetime.datetime.now(CET())

while(now <= endtime):
    lastupdates = getUpdate(f, lastupdates)
    time.sleep(60)
    now = datetime.datetime.now(CET())

f.close()
```

```
# -*- coding: utf-8 -*-
"""
ANNEX 2:

Data preprocessing
"""

import numpy as np
import pandas as pd
import datetime
import matplotlib.pyplot as plt

#%% Fonctions

def datetimify(x):
    return datetime.datetime.strptime(x, "%Y-%m-%d %H:%M:%S")

def stringify(x):
    datetime.datetime.strftime()
    return x.strftime("%Y-%m-%d %H:%M:%S")

def getProfile(Station, X, delta_minutes, output=False):

    nb = Station[1]
    banking = Station[4]
    bonus = Station[5]
    if banking == True:
        banking = 1
    else:
        banking = 0
    if bonus == True:
        bonus = 1
    else:
        bonus = 0
    selected = X['Station'] == nb

    Temp = X[selected]
    start = datetimify("2016-04-25 04:01:00")
    end = datetimify("2016-04-29 04:01:00")
    delta = datetime.timedelta(minutes=delta_minutes)
    K = int((end - start)/delta) + 1
    timestamps = [start + k*delta for k in range(K)]
    profile = []

    N = len(Temp.index)
    index = iter(range(N))
    i = next(index, N)
    stands = Temp.iloc[i,3]
    date = datetimify(Temp.iloc[i,1])
    bikes = Temp.iloc[i,4]

    for ts in timestamps:
        while (i < N and date < ts):
            i = next(index, N)
            if i < N:
                date = datetimify(Temp.iloc[i,1])
        bikes = Temp.iloc[max(0,i-1),4]
        profile.append(str(100.0*bikes/stands))

    if output:
```

```
        f = open('profiles_52.csv','a')
        f.write("{};{};{};{};{}\n".format(nb,stands, banking, bonus,';'.join(profile
        f.close()

    return timestamps, profile

def plotprofile(timestamps, profile, title):

    plt.figure(figsize=(15,10))
    plt.plot(timestamps, profile, color='red')
    plt.ylabel('Occupance rate')
    plt.xlabel('Time')
    plt.savefig("profile_{}.png".format(title))
    plt.show()

def getClosedStations(X, Y):

    closed = []
    for station in Y.itertuples():
        nb = station[1]
        selected = X1['Station'] == nb
        Temp = X1[selected]

        if 'OPEN' not in list(Temp['Status']):
            print(nb)
            closed.append(nb)

    return closed

def createDataPoints(nb, timestamps, profile, delta_minutes):

    step = int(60/delta_minutes)
    K = len(profile)

    f = open('datapoints1.csv', 'a')

    for k in range(0,K,step):
        nexthour = [float(x) for x in profile[k:k+step]]
        time = timestamps[k].hour
        current = profile[k]
        if(100.0 in nexthour):
            test = 1
        elif(0.0 in nexthour):
            test = -1
        else:
            test = 0

        f.write("{};{};{};{}\n".format(nb,time,current,test))

    f.close()

#%% Loading Data
X1 = pd.read_csv('week1.csv', sep=',',header=None)
Y = pd.read_csv('stations.csv', sep=';',header=None, encoding = 'latin1')
X1.columns = ['Station', 'Date', 'Status', 'Stands', 'Bikes', 'Spots']
Y.columns = ['ID','Name','Position','Banking','Bonus','Lastupdate']

#%% Code

closed = getClosedStations(X1, Y)
```

```
print(closed)

f = open('datapoints1.csv','w')
f.close()

for Station in Y.itertuples():
    nb = Station[1]
    print(nb)
    timestamps, profile = getProfile(Station, X1, 5)
    createDataPoints(nb, timestamps, profile,5)
```

```python
# -*- coding: utf-8 -*-
"""
ANNEX 3:

Clustering
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from scipy.cluster.vq import kmeans2

#%% Fonctions

def datetimify(x):
    return datetime.datetime.strptime(x, "%Y-%m-%d %H:%M:%S")

def getTimestamps(delta_minutes):

    start = datetimify("2016-04-25 04:01:00")
    end = datetimify("2016-04-29 04:01:00")
    delta = datetime.timedelta(minutes=delta_minutes)
    K = int((end - start)/delta) + 1
    timestamps = [start + k*delta for k in range(K)]

    return timestamps

def plotDendrogram(Y, title):

    plt.figure(figsize=(10, 7))
    plt.title('Hierarchical Clustering Dendrogram')
    plt.xlabel('sample index')
    plt.ylabel('distance')
    dendrogram(
        Y,
        truncate_mode='lastp',
        p=25,
        leaf_rotation=90.,
        leaf_font_size=12.,
        show_contracted=True,
    )
    plt.savefig("{}.png".format(title))
    plt.show()

def plotprofile(timestamps, profile, title):

    plt.figure(figsize=(15,10))
    plt.plot(timestamps, profile, color='red')
    plt.ylabel('Occupance rate')
    plt.xlabel('Time')
    plt.savefig("profile_{}.png".format(title))
    plt.show()

def plotAll(timestamps, avg_profiles):

    plt.figure(figsize=(15,10))
    plt.plot(timestamps, avg_profiles[0], color='red', label="Cluster 1")
    plt.plot(timestamps, avg_profiles[1], color='blue', label="Cluster 2")
```

```
    plt.plot(timestamps, avg_profiles[2], color='purple', label="Cluster 3")
    plt.plot(timestamps, avg_profiles[3], color='green', label="Cluster 4")
    plt.ylabel('Occupance rate')
    plt.xlabel('Time')
    plt.legend()
    plt.savefig("profile_all_1.png")
    plt.show()

#%% Data

X = pd.read_csv('profiles_52.csv', sep=';', header=None)
Xoriginal = X.copy()
X = X.drop(0,1)

#D = pd.read_csv('datapoints1.csv', sep=';', header=None)
#%% Code

##Standardization
#for c in X.columns:
#    X[c] = X[c]/np.var(X[c])

Y = linkage(X, 'ward')
Z = fcluster(Y, 4, criterion='maxclust')

plotDendrogram(Y, "clusters2")

labels = []
for z in Z:
    if z not in labels:
        labels.append(z)
labels.sort()

timestamps = getTimestamps(5)
avg_values = []
avg_profiles = []
clusters = []
stations = []

for i in range(len(Xoriginal.index)):
    clusters.append(Z[i])
    stations.append(int(Xoriginal.iloc[i,0]))

for z in labels:
    values = np.zeros(3)
    profile = np.zeros(len(X.columns)-3)
    k = 0
    for i in range(len(Xoriginal.index)):
        if(Z[i] == z):
            values += Xoriginal.iloc[i,1:4]
            profile += Xoriginal.iloc[i,4:]
            k += 1
    values = values/k
    profile = profile/k
    avg_values.append(values)
    avg_profiles.append(profile)
    plotprofile(timestamps, profile, "cluster{}_1".format(z))
    print("Cluster {}:".format(z))
    print(k)
    print(values)
```

```
W = pd.DataFrame({'A':stations, 'B':clusters})
W.to_csv('stations_clusters.csv', header=None, index=None, sep=';')

plotAll(timestamps,avg_profiles)
```

```
# -*- coding: utf-8 -*-
"""
ANNEX 4:

Neural Network
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cross_validation import train_test_split


#%% Fonctions

def sigmoid(x,deriv=False):
    if(deriv==True):
        return x*(1-x)
    return 1/(1+np.exp(-x))

def softmax(y,deriv=False):
    if(deriv):
        return softmax(y)
    return np.exp(y)/np.sum(np.exp(y),axis=1,keepdims=True)

def binarize(Y):

    Ytemp = Y.copy()
    Ytemp[Ytemp>0.5] = 1
    Ytemp[Ytemp<0.5] = 0

    return Ytemp

def binarize2(Y):

    Y[Y>0.5] = 1
    Y[Y<-0.5] = -1
    Y[np.all([Y>-0.5,Y<0.5])] = 0

    return Y

def confmatrix2(y, Y):

    Ytemp = binarize2(Y)

    nY = 3
    N = len(y.index)
    M = np.zeros((nY,nY), dtype = np.int)
    cat = [0,1,-1]
    for i in range(N):
        jy = cat.index(y.iloc[i,0])
        jY = cat.index(Ytemp[i])
        M[jy,jY] += 1

    error = 1.0 - np.trace(M)/np.sum(M)

    return M, error

def confmatrix(y, Y):
```

```
    nY = len(y.columns)
    N = len(y.index)
    M = np.zeros((nY,nY), dtype = np.int)

    if nY == 1:
        return confmatrix2(y,Y)

    Ytemp = binarize(Y)

    for i in range(N):
        jy = np.argmax(y.iloc[i,:])-6
        jY = np.argmax(Ytemp[i,:])
        M[jy,jY] += 1

    error = 1.0 - np.trace(M)/np.sum(M)
    conf = np.trace(M[1:,1:])/np.sum(M[1:,:])

    return M, error, conf


#%% Data

dftrain = pd.read_csv('datapoints4train.csv', sep=';', header=None)
dftest = pd.read_csv('datapoints4test.csv', sep=';', header=None)

x = dftrain.iloc[:,0:6]
y = dftrain.iloc[:,6:]

xtest = dftest.iloc[:,0:6]
ytest = dftest.iloc[:,6:]


#%% Code

#Number of hidden neurons
nZ = 5
#Learning rate
ro = 0.0005

N = len(x.index)
nX = len(x.columns)
nY = len(y.columns)

np.random.seed(1)

# randomly initialize our weights with mean 0

a = 2*np.random.random((nX,nZ)) - 1
a0 = np.zeros((1,nZ))
b = 2*np.random.random((nZ,nY)) - 1
b0 = np.zeros((1,nY))

for j in range(60000):

    #Feedforward
    X = x
    V = np.dot(X,a) + a0
    Z = sigmoid(V)
    Y = softmax(np.dot(Z,b) + b0)
```

```
    #Backpropagation
    d2 = Y - y
    d1 = sigmoid(Z, deriv=True)*(np.dot(d2,b.T))

#    ro = ros[int(j/20000)]

    b += -ro*np.dot(Z.T,d2)
    b0 += -ro*np.sum(d2, axis=0)
    a += -ro*np.dot(X.T, d1)
    a0 += -ro*np.sum(d1, axis=0)

    if(j%10000)==9999:
        M, e, c = confmatrix(y,Y)
        print(M)
        print(e,c)
        print("")

X = xtest
V = np.dot(X,a) + a0
Z = sigmoid(V)
Ytest = softmax(np.dot(Z,b) + b0)

M, e, c = confmatrix(ytest,Ytest)
print(M)
print(e, c)
```