Big Data Technologies

Ivan Martini, Beatrice Marsili
(Group 17)

July 2020

Introduction

Design and implement a big data system that explores Bike sharing service in Washington DC (`https://www.capitalbikeshare.com`) to suggest for a one-day tourist with green habits a tour for visiting different Washington DC museums based on bikes' availability. Consider that a bike can be left only at the closest station, define some criteria of closeness, and uniform min time required to visit a single museum.

The development of this project has been totally centered on our academic experience, in order to achieve the best learning result possible. Our goal was to dirty our hands in this experiment providing a functional (and maybe rude) big data pipeline. We also chosen to avoid paying for any of the services required. This report will explain the reasons behind our choices and possibly highlight other ways things could have been done.

In all the steps followed in this project we tried to simulate a real scenario to train our ideas and highlight possible issues; we think that, thanks to this way of proceeding, we managed to have a clear idea of what we were doing and why. One of our biggest concern was an high modularity of the system: for instance we could add suggestion regarding vegetarian restaurants to have a lunch while visiting the museums (to keep on going with the idea of a green day) or totally change the target and produce instead a system that help users going shopping, without changing most part of our system.

The report is organized as follows: In the first part there are the main design strategies an some observation on the data to analyze. Afterwise there is a detailed description of the pipeline stages. Lastly, some conclusion on our work.

1 Pipeline Overview

1.1 Project definition and initial brainstorming

The first point of the development is always the *requirement analysis.* Therefore we brainstormed a series of features and tried to imagine an high level architecture for our system, finding our question marks. Our unknowns were the definition of *closeness* and the *time required* for visiting a museum.

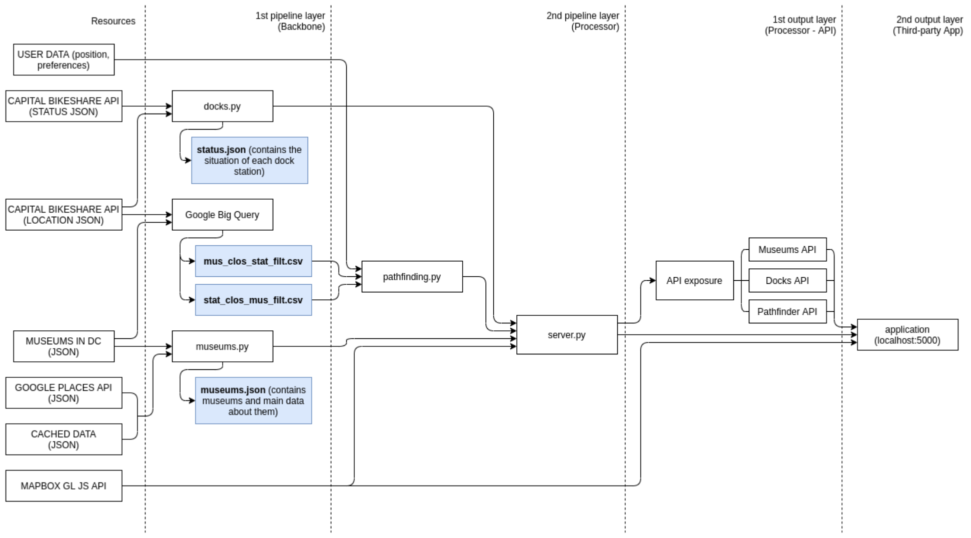
The first issue could have been solved using some "directions api" from some provider online (Google, Mapbox, ..) but finding the closest station for each museums would not have been possible with the free request allowed on any of them. The solution we adopted was therefore the euclidean distance, which might seem trivial but works fine (we performed some rude manual testing testing, and the results were correct in most of the cases, with some imperfections e.g. leading to the second closest on the map) and we decided to adopt this solution.

The second unknown was a little trickier, since is hard to estimate the visit time for a museum (If you visiting the MUSE probably you just need a couple hour, while visiting the Louvre could take a couple days.) Firstly we relied on the google places API, which we also use for getting data about each museum, but differently from shopping centers or amusement parks most of the museums lack this information (which is both sad and interesting). If we were working with the DC government we could ask them to gather these data, but we decided to tackle the problem from another perspective. We decided a standard time (1h30) and in the application there is a request for the number of museums that the user want to visit. Obviously, changing the standard value with the realistic one won't endanger our infrastructure.

After this part we proceeded familiarizing with our data. We started a "pre-processing phase" where we relied on **Knime software**. In this way we understood which data the bike sharing service offered (in terms of density and docks population) and which ones had to be retrieved in some other way; it also allowed us to adopt a new way of thinking: the one of the pipeline. How to obtain the final result segmenting the whole problem into minor/compartmentalized ones?

In few steps we extracted, cleaned and joined together data we were interested in and visualized them on OSM to have a rough idea of how bike stations are actually distributed on the Washington D.C. territory. We also uploaded in the pipeline the data provided by D.C. Government regarding museums and in this way we had the possibility to have a look to the distribution of the museums as well.

1.2 Design strategy



Resources
1st pipeline layer (Backbone)
2nd pipeline layer (Processor)
1st output layer (Processor - API)
2nd output layer (Third-party App)
USER DATA (position, preferences)
CAPITAL BIKESHARE API (STATUS JSON)
docks.py
status.json (contains the situation of each dock station)
CAPITAL BIKESHARE API (LOCATION JSON)
Google Big Query
mus_clos_stat_filt.csv
stat_clos_mus_filt.csv
pathfinding.py
server.py
API exposure
Museums API
Docks API
Pathfinder API
application (localhost:5000)
MUSEUMS IN DC (JSON)
museums.py
GOOGLE PLACES API (JSON)
museums.json (contains museums and main data about them)
CACHED DATA (JSON)
MAPBOX GL JS API

Figure 1: *Pipeline stages representation*

Having a better knowledge of our data, we planned or tasks. The raw idea of the pipeline was *having two stages*: the first one to build the architecture needed and the second one actually process the users' requests. From now on the two layers will be referred as **backbone** (for the first one) and **processor** (for the second one).

The goal for the **backbone** was a customizable solution. We knew, that the input data were the one from capital bikeshare and from the opendata DC websites, but we tackled the problem as a generic one: with only a few modifications on the data schema the architecture can parse any (geological) data, for example replacing the museums with shopping centers or restaurants, keeping the other "modules" as they are.

The goal for the **processor** was instead the exposition of some custom API for all the data we used. The API are very simple but fully implement the REST paradigm, offering the museums or the bike docks in a standardized format and also a pathfinding API, all provided using the JSON notation. During the design of the processor we also considered a tool for plotting all the data: a webapp using the API we offered acting as a third party application. Processor and webapp run on the same server, but they interact as different systems (offering in our opinion a good degree of simulation of a real case scenario).

2 Implementation and Development

2.1 Backbone implementation

relative files: `museums.py`, `docks.py`, `queries.txt`

Python. The first step was obtaining and understanding better our data. We instantly decided python to be our language and we developed `docks.py` and `museums.py`. Those files contain some functions that can extract data from the streams provided by Washington Bikes and are aggregate them as needed (e.g. station location and bike availability). All the function contained can also save intermediate results in *csv/json* format, following the indication of `utilities.py`. The ease of use of python allowed us to plot the first visualization of the data

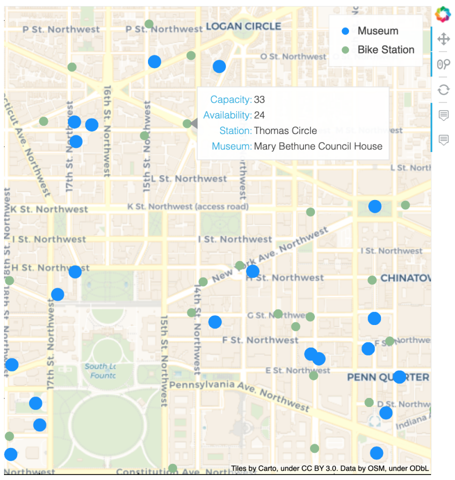


Figure 2: *Visualization of a bike station and related information*

Google Places API. The first version of `museums.py` was effective, but we felt that the dataset provided by Washington Government did not supplied enough information (the name, the address, and the geographical coordinates).

We then decided to use Google Places API (a subsidiary of GMaps API) to perform web scraping and retrieve more information regarding museums. We joined the information provided by the Washington Government with the Google knowledge, obtaining information about opening and closing times, websites and contact info, rating and in some case also ticket costs. These info were really useful and we decided to proceed, but not complete as hoped (as said in the first section). Google has a really expensive pricing policy, so we decided to build a local caching infrastructure: all the obtained data are stored locally and when they have to be retrieved the system decides if getting them through the cache (simulating the API) or download them again (and update the cache). In this way we might use old data without sustaining any cost (however we expect those information to be quite static).

Google BigQuery. After the extraction of the relevant information for our analysis we pushed our data to the Google Cloud Platform via a Google API Authentication and we started creating a database that suited our purposes. Our original data were in *csv* and *json* formats, so we managed to insert them easily (the *json* files schema were trivial). Through **SQL** (the queries are written in `queries.txt`) we obtained some CSV defining the closest stations to each museums and the closest museums to each station. They are the core feature of our pathfinding API,

```
SELECT *
EXCEPT (name, num_bikes_available), name as Station_Name;
ST_DWITHIN(mus.mus_point, stat.stat_point, 300) AS closest;
ST_DISTANCE(mus.mus_point, stat.stat_point)/1609 AS distance;
FROM Washington_Museums mus;
 station_points stat;
WHERE ST_DWITHIN(mus.mus_point, stat.stat_point, 300) = True;
ORDER BY MUSEUM
```

For instance, as can be seen from the code reported above, we solved one of the most important parts of the project with a query that has been performed in less than a second and allowed us to define for each museum at least one bike station in a radius of 300 feet. In the same query we also created a column with the approximated distance between museum and station expressed in miles. With the *csv* files produced by **Google BigQuery** and the other intermediate results we closed the *backbone* development and proceeded with the other tasks.

2.2 Processor implementation

relative files: `server.py`, `pathfinder.py`

The processor layer has been implemented in plain python, and it operates over the intermediate results provided by the backbone layer. The layer exposes the API providing the complete data on museums and docks in real time (or through the museum cache) describing the situation and offers the pathfinding algorithm.

The API exposition in carried using a client server architecture in a RESTful manner. When interrogated the server will answer with a json object containing the information requested. The operation permitted are now limited to the one strictly needed for the accomplishment of the project. The operation available are:

- `GET /api/museums` - collects all the updated data from the museums of Washington DC. The result is the same of the stored in *museums.json*
- `GET /api/docks` - collects all the updated data from the capital bikeshare website. The data are up to date, even if at a first look they seem unchanged after some minutes. The reason is simple (and it took us a bit to get it): due to time shifts, most of the queries are executed during the US night (which as expected presents a low affluence to the bike sharing infrastructure)
- `GET /api/path` - this API expects three parameters: *latitude* and *longitude* of the user (in order to guide him to the closest available station) and number of museums wanted. The request looks like:

```
$ curl -X GET \
     http://localhost:5000/api/path?count=<..>&lat=<..>&lon=<..>
```

The algorithm works using the infrastructure provided by the *backbone* layer about the distances list between museums and docks. After finding the closest station (i.e. a station amongst the closest with a suitable number of available/occupied docks) the algorithm produces a list of station and museums that the user is suggested to visit during the day.

Unluckly, since we found no suitable classification for the museum type, they are considered only by the distance between them, not considering the closest ones and the furthest ones (avoiding a visit without any cycling or a visit that requires many kilometers all over Washington). We decided to follow this approach starting from a brief analysis on the distribution of the museums in the capital, which is sparse but clustered (e.g. *there are some dense groups quite distant one from the other*)

2.3 Third-party web app implementation

A brief paragraph is for the web app that allow us to effectively check the work done by the pipeline. The backend is developed in python (Flask 1.1.2) while the frontend is a standard HTML document enhanced by jQuery and Bootstrap. It is provided by the same server that exposes the API, however it works as a decoupled application that relies on our own API and on the *mapbox API* (map display + visual path).

The development of this "interface" is not strictly bound to the big data project (once exposed the API, theoretically, anyone could access and use them) but it has immediately been one of our priority. Indeed using just raw data on the coordinates can be very tricky and time consuming, thus having a platform that visually provides us a feedback (geodata are intrinsecally designed to be plotted) improved a lot our ability to both detect and solve a problem.

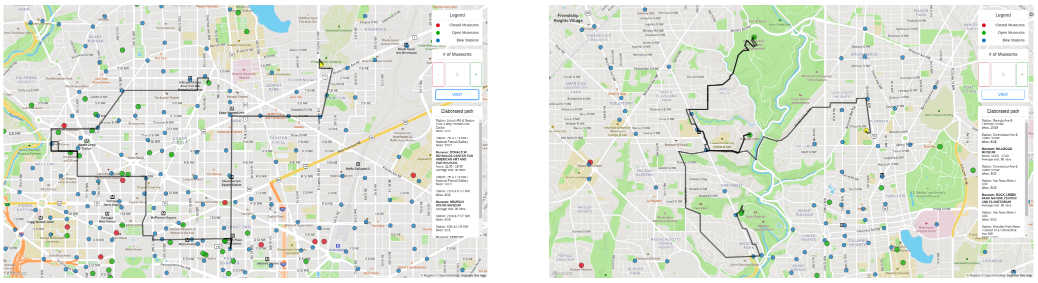


Figure 3: *Two screenshots of the application running*

3 Conclusions

In this report, we presented our work on a big data architecture able to provide an user with a suggestion on a "green trip" around the museums of Washington DC using the capital's bike sharing infrastructure. We took care of each step from the very beginning to the end, providing a maybe crude but, in our view, efficient implementation of the required pipeline.

We defined our unknowns and also provided a realistic set of workarounds on some inherent issues (the need for some expensive services to provide more reliability on the data, the lack of a real correspondence between our GPS and time zone, the lack of various data on some museums, the discrepancy on some data from our sources and the ones in the real world due to the actual sanitary emergency) that allow us to be proud of our work and be more confident on the "big data terrain".