Data science project - Guillaume's Part

Nominatim server

When we received the nominatim servers, I created the non-administrator user account for our use and I added the SSH public keys of all the team.

Python and Nominatim

We use `geopy` to use the Nominatim of Python.

Connection to our own Nominatim server

Until now, we were using the default public Nominatim server at https://nominatim.openstreetmap.org/, but we must use our own Nominatim server so I looked into 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 I fixed it. The server can be accessed at : http://i-nominatim-01.informatik.hs-ulm.de/nominatim/ .

Second, **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 :

```
1  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 the **geopy** query timeouts. I had to raise this limit to a bigger number.

Here is what the connection line looks in the end :

```
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)
```

Research

I researched different things to use Python with the Nominatim API. For each, I wrote a **Jupyter notebooks** discribing the steps to achieve it and compared the results with **Google Maps**.

Querying of the centroid

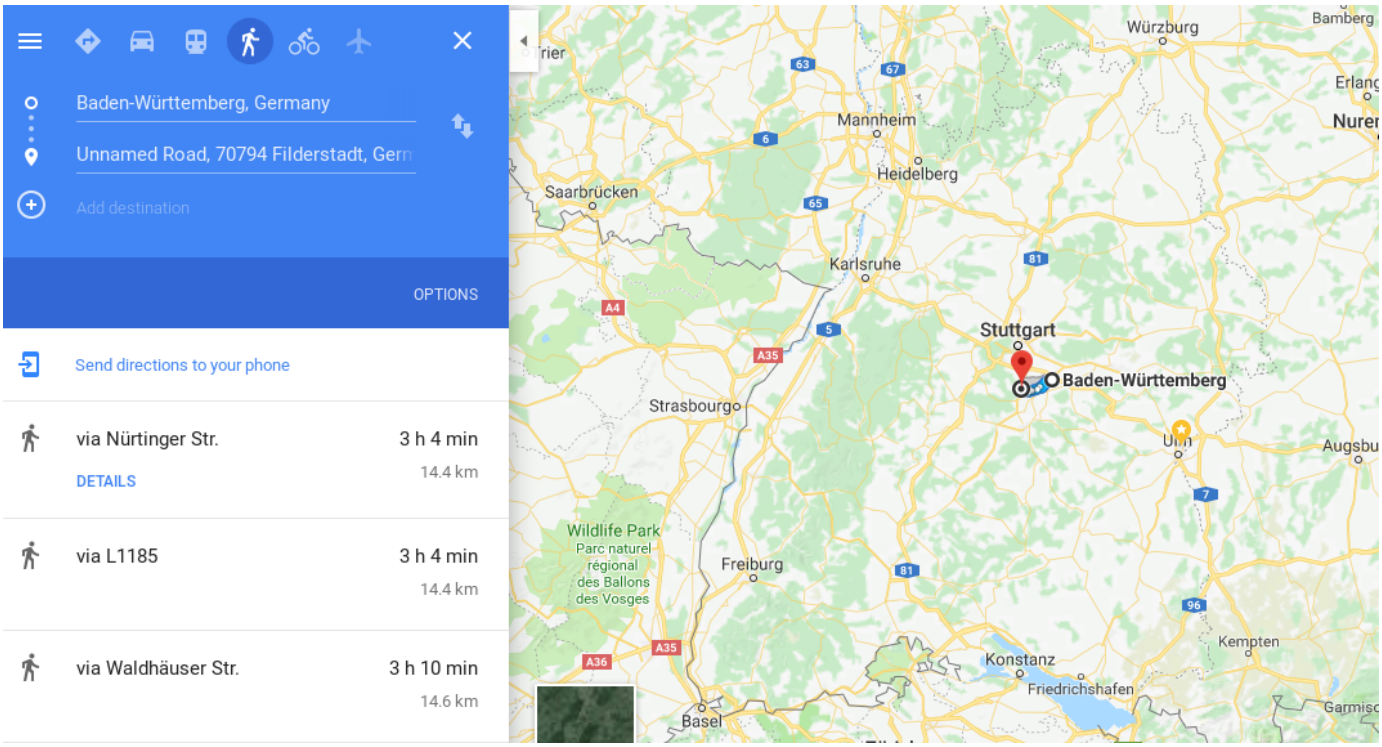
```
location = locator.geocode("Baden-Württemberg, Deutschland")
print(location.address)
print(location.latitude, location.longitude)
```

Result :

```
Baden-Württemberg, Deutschland
48.6296972 9.1949534
```

Comparison with Google Maps :

I entered the given coordinates for the centroid of Baden-Württemberg (**48.6296972**, **9.1949534**) in Google Maps and made an itinary between this centroid and the one given by Google Maps :



We can see that there is only **14.4km** between the two points. If we consider *Baden-Württemberg* 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.

Transforming a coordinate to an address

```
location = locator.reverse((location.latitude, location.longitude))
print(location)
```

Result :

```
Kipfenberg, Landkreis Eichstätt, Obb, Bayern, 85110, Deutschland
```

Distance by road

There is not much library providing a routing algorithm for Python and working with OpenStreeMap.

pyroutlib2 (PyrouteLib — OpenStreetMap Wiki) is the only one I found. It does not seem to be supported and I 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 given below. 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.")
```

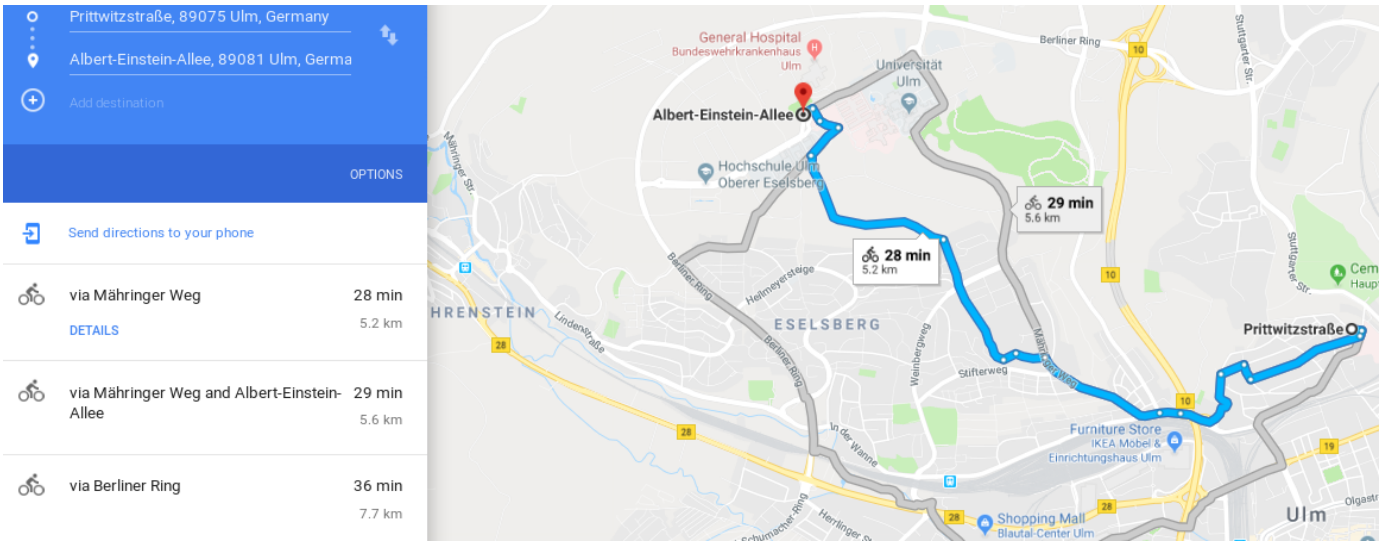
Finally, to get the distance, we compute the distance between each couple of points :

```
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)
```

Comparison with Google Maps

From **Prittwitzstaße** to **Albert-Einstein-Allee**, we get a distance by route of **5.87km**. Google Maps gives a similar result of **5.2km**.



Note on performance

For some in-city routes, the library is pretty fast, but for longer routes (Stuttgart -> München), **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 route for the project.

Grouping all Python code to a library

The different methods we need on the Nominatim API are grouped in a Python library on the project's Github. It can be accessed at : https://github.com/dataBikeHsUlm/NominatimLibrary .

It needs `geopy` as a dependency and `osmapi` for `pyroutelib2` :

```
pip3 install geopy osmapi
```

You can use the library easily :

```
import NominatimLibrary

locator = Locator()
coords = locator.get_coordinates("Prittwitzstaße, Ulm, Germany")
```

Github account

To group our code base and have versionning on it, I created a Github account available at : https://github.com/dataBikeHsUlm .

I also added three projects :

- WebApp : for the web server
- NominatimLibrary : for the library aformentioned
- MapReduce : for the scripts for Hadoop

Django

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

Tutorial

I wrote a recapitulative document for the rest of the team, so they can quickly get ready to work on the server code. This document can be accessed at : https://github.com/dataBikeHsUlm/WebApp/blob/master/django_recap.md .

It is mostly inspired by the *official tutorial of Django* at https://docs.djangoproject.com/en/2.1/intro/tutorial01/ and from the Django documentation.

For further information on Django and how to use it, go to : https://docs.djangoproject.com/en/2.1/.

Postcodes and distance tables in the database

Datamodel for the postcodes and distance

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 https://github.com/dataBikeHsUlm/WebApp/blob/master/geonom/datamodel/models.py .

For further information on working with the database in Django, see : https://github.com/dataBikeHsUlm/WebApp/blob/master/django_recap.md#database .

Initializing the postal code table

To be able to fill the postal code table, we need a list of all postcodes per country. The Nominatim API doesn't directly provide a function to do that, so I looked on the internet and found a list of all postcodes per country at http://download.geonames.org/export/zip .

I implemented a first version of a script using this list and started filling the database it.

Problem was, some countries are missing like Greece and the script had problems finding some cities, so we had to find another way to get the postcodes and it must be more reliable.

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 is :

```
SELECT DISTINCT country_code,postcode \
  FROM location_area \
  WHERE country_code IS NOT NULL \
    AND postcode IS NOT NULL \
  ORDER BY country_code, postcode;
```

Then, for every postcode, we need to query Nominatim to get the centroid coordinates, but querying only with the country code and the postcode is too ambiguous and can lead to wrong centroid. To fix that, we use the `country_name` table that lists countries with names and ISO codes. We join 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;
```

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 :
https://github.com/dataBikeHsUlm/WebApp/blob/master/fill_db_from_osm.py

To run the script, run the accompanying shell script : `fill_db.py` , it will install the dependencies before running the python code.