OTH Regensburg

Spatial Databases

Project Report

Birgit Huber, Florian Fritz, Florian Fußeder

2.1.2017

Inhaltsverzeichnis

Project Summary

The goal of the project was to create a web application that helps a user to find a suitable region for accommodation in a city. By answering a series of questions about preferences and living circumstances we aim to suggest a ranked list of the zones of a city.

Furthermore, the user should be able to enter some main locations. With the given data we are able to calculate a time- and cost-efficient route.

After some research, we chose the city “New York” and discussed the which datasets we might use. Using the datasets, we defined our questions for the user and started with processing the data for an easy integration in our project. Afterwards we were able to start with the development of the web web-application.

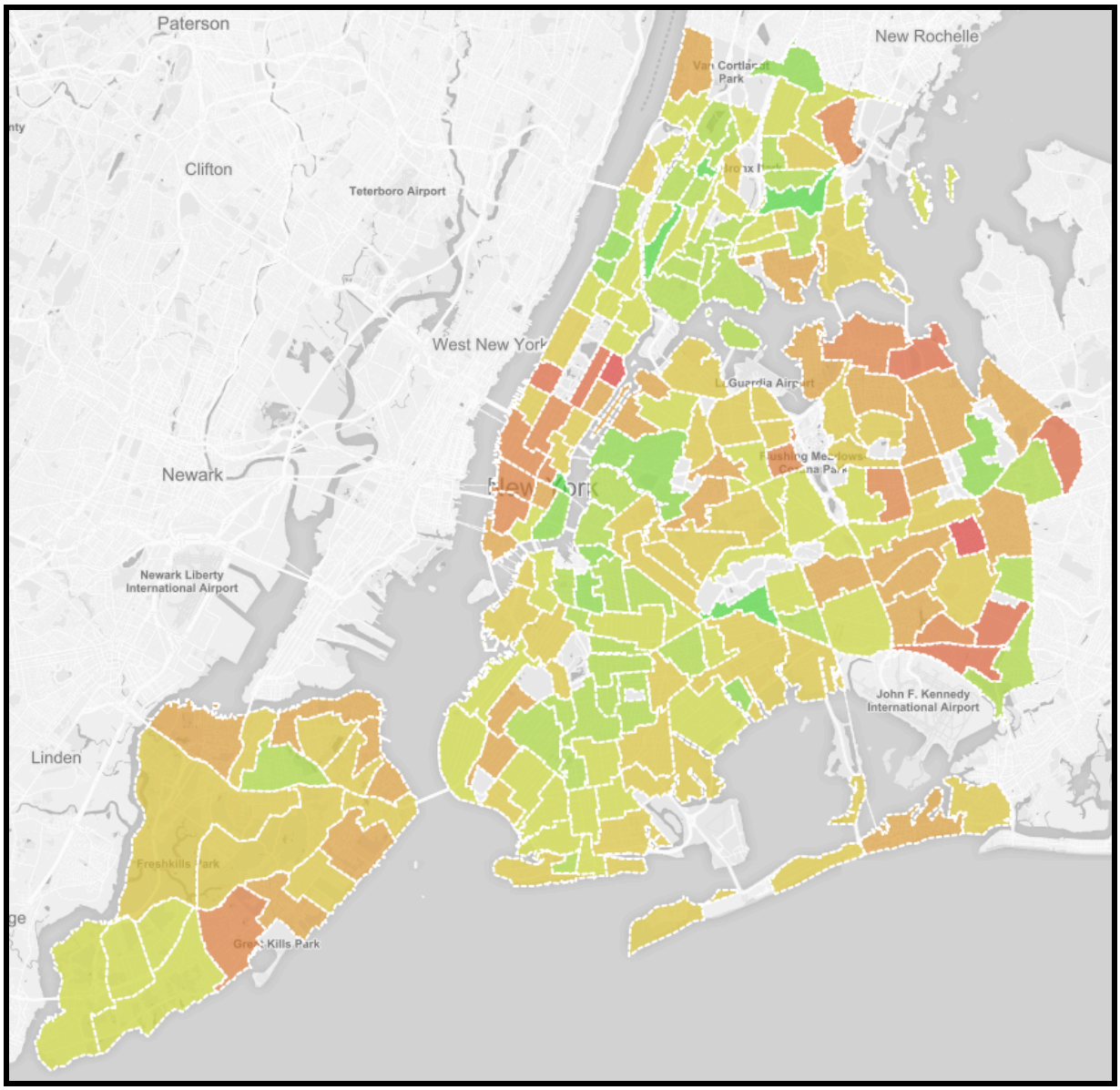

Figure 1: Areas of New York

Decision Making

We started off by considering nine cities, which we thought, might be interesting and could have good datasets available. Each member of the team had a closer look at some cities and their available datasets as following:

- Birgit Huber: San Francisco, Beijing, London
- Florian Fritz: Berlin, New York, Tokyo
- Florian Fußeder: Munich, Seoul, Singapore

After a week, we discussed our results and chose the "City of New York". We discovered that the website *open data census*[1] is very useful to evaluate the amount and quality of available open data. One of the main reasons we chose New York was the fact that the United States have a lot of open geo datasets available, because of an open data initiative[2]. New York City has its own website with geo data[3] which we used for our project. New York is, by default, divided into 5 Zones with total area of approx. 790 km² and 194 Neighborhood-Tabulation-Areas (NTA) with an average size of four km² which perfectly suited the needs of our project. Because of the questions, we wanted to ask the user, we chose to use the following datasets:

- Parks
- Play areas
- Restaurants
- Soccer Fields
- School Points
- Parking lots
- Rental Prices
- Colleges and University's
- Population
- Complaint Data
- Subways

Questions for the User

Resulting of our chosen datasets we picked the following questions to determine a suitable NTA for the user:

- Age range
- Has Children
- Is a student
- Owns a car
- Has a dog
- Does outdoor sports
- Uses subway
- Likes nature
- Prefers vibrant or quiet areas
- Importance of low rental prices
- Prefer to live central
- Favors specific zones

[1] http://census.okfn.org/en/latest/
[2] https://www.data.gov/
[3] https://nycopendata.socrata.com/

Processing the Data

Each Dataset was processed individually by a team member. The result of each dataset was a database view containing a rating value between zero to one for each region. For example, the area with the least parking lot area scores the lowest rating, the area with the most scores a one. This allows us to easily weight the different ratings in the web application, as they are now all within the same range.

Neighborhood-Tabulation-Areas

As mentioned we chose the NTA areas as foundation of our ratings and want to give suggestions of the most fitting ones to our user. The NTA could easily be imported from the existing shapefiles[i]. The Shapefile contained a NTA code, the NTA name, the geometry and some other metadata. Nothing had to be processed in this table.

School Points

The school point data[ii] was imported as a shapefile, containing locations as geometry based on the official address. It includes some basic school information such as name, address, principal, and principal's contact information.

The rating of how good a NTA zone scores in terms of school points, was determined by the amount of school points within the zone. This was done by using basic PostGIS queries[iii], including ST_AREA and ST_CONTAINS. As a final step a normalized view, which only contained NTA code and the rating[iv] was created.

Colleges and Universities

Like the school points table, the data that we used for the colleges was also easy to import, because there was a shapefile[v] available which included some metadata like the name and the street name and the geometry.

Again, the rating of areas containing a college or a university is higher than regions that do not. After viewing the data in the web application we concluded, that because of the low numbers and the bulked locations of the universities only a few areas would have a good rating. So we decided it would be better for our scores to use ST_DWITHIN and add a distance of 200 for our rating view[vi].

Parking lot areas

The parking lot data also already included the geometry of the area[vii]. So we just had to prepare the data for our rating.

In this case, we had two polygon geometries and wanted the area of all parking lots. So, we used ST_INTERSECTS. To get an accurate rating that is not biased towards bigger NTA areas, we calculated the m² of parking space per m² in the area[viii].

Population

The population data consisted of a simple csv-file[ix] that contained the data from the year 2000 and 2010.

After removing the data from 2000 we had usable data that only had to be match by NTA-code[x]. To get a more accurate rating of how crowded the regions are, we calculated the population per m² in the area.

Complaint data

This dataset[xi] includes crimes that were reported to the police in the first three quarters of 2016. The csv-file had the latitude and longitude where the crime took place, as well as some metadata which we cut out to speed up the database queries.

To get a rating of how many crimes have been reported to the police we had to create a point geometry out of the latitude and longitude, by using ST_MAKEPOINT. After that we used ST_CONTAINS to determine the count of crimes within an area[xii].

Rental Data

To determine a approx. rating of the buying/rental data of the specific areas we used the condominium data provided by the NYC department of finances[xiii].

The datasets consist out of five complex Excel worksheets with much data that was not needed for our project. After removing the unwanted data, we converted the file into simple csv-format. Because the data had no spatial data, nor any reliable source to match the csv-entries to a specific NTA-area, but

the address of the building, we decided to use a geocoding[4] API. In our case we used the Python Mapbox API to write a simple script[xiv] consisting of reading the csv-file, sending the address, receiving the latitude longitude and writing this data into new csv-files. After reviewing the locations of the points that we had produced with the locations from the Mapbox geocoder, we discovered that a lot of the produced points were located all over the world. After counting the points, we discovered that approx. 2450 out of the total 23080 entries are clearly out of the boundaries of New York.

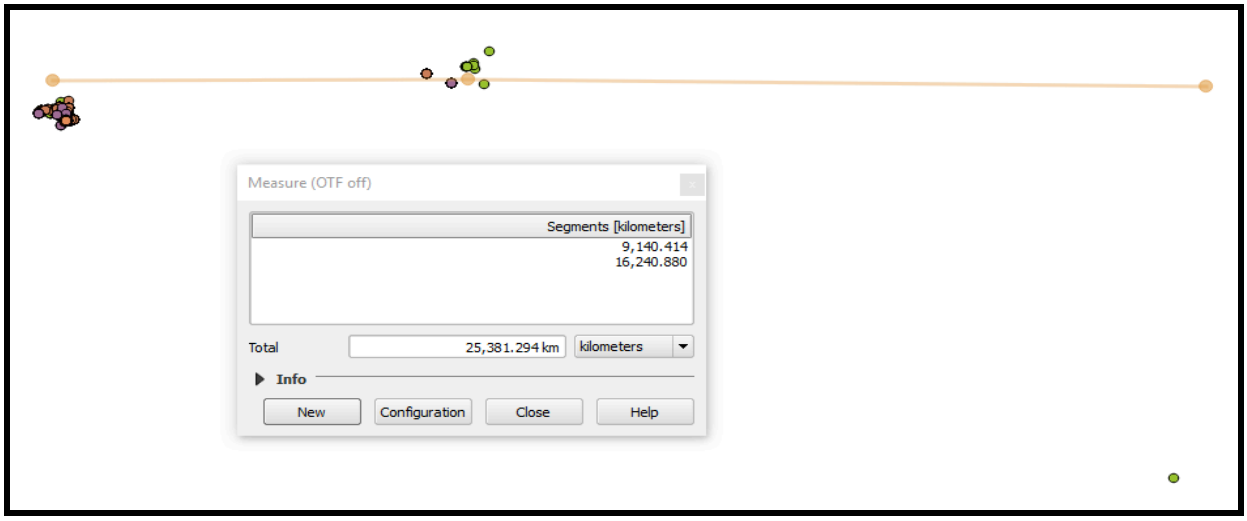

Figure 2: Error Point Distribution 1

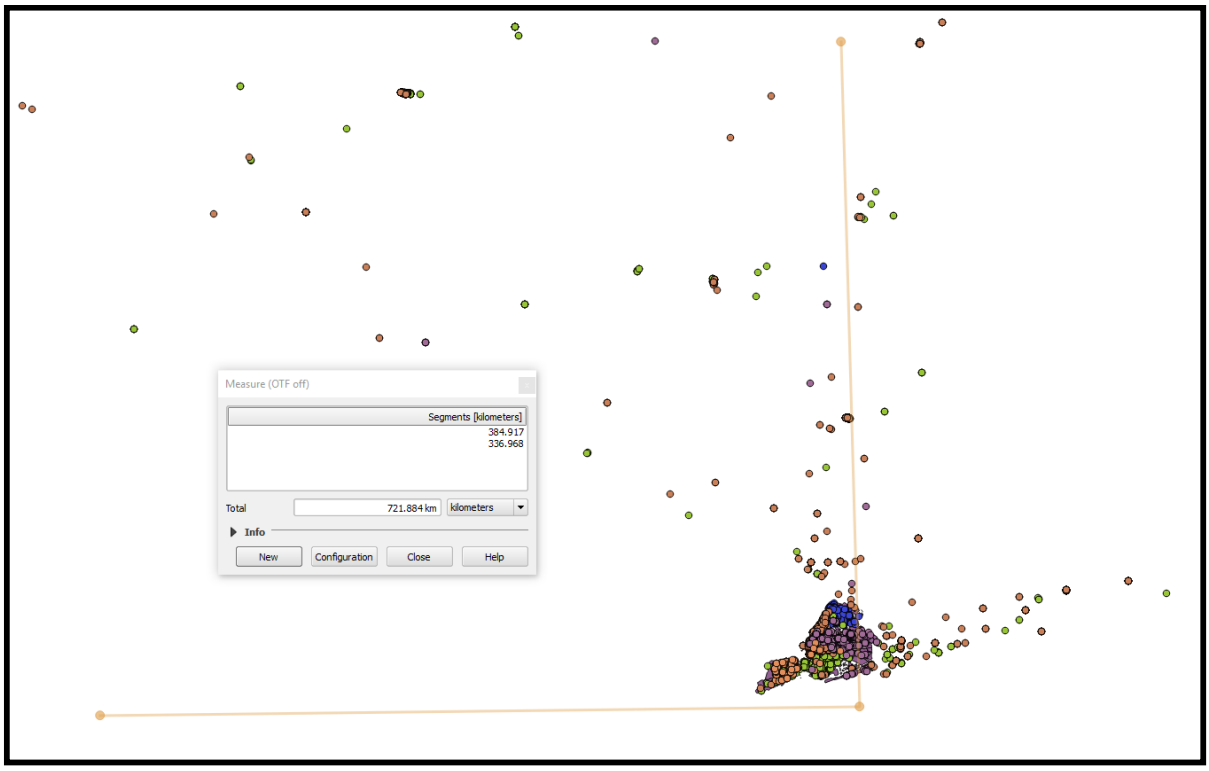

Figure 3: Error Distribution 2

[4] Computational process of transforming a postal address to a location (latitude longitude)

Because we decided that we did not want to use such unreliable data, we wrote another Python script[xv], but this time we did choose the geocoder API from Google Maps. After reviewing the locations of the points that were created out of the Google latitude and longitude data, we discovered that only four points were not located in New York. So we used the Google locations to determine the average market value per square feet for each NTA, which give a basic overview of the price range.

Subways

The data for the subway stations was imported as a shapefile[xvi], which included the name of a certain subway station, its geometry as a point and other meta data.

Then the distance to the closest subway station was calculated for each NTA region by using the PostGIS ST_DISTANCE method[xvii]. Finally, the distances were mapped to a rating value ranging from about zero, for the NTA with the furthest distance to the subway, to one, which means that there is a subway station in that area[xviii]. Therefor the distance of each NTA was divided by the maximum distance. After subtracting one from the result, it was multiplied with -1 to get a result of one for the best rating.

Soccer fields

The Soccer field data set was imported as a shapefile[xix] into the PostGIS database.

The dataset consists out of the geometry, an id and other meta data. Like the previously described data processing of the subway, we did calculate the distance to the closest football field[xx], as there are not that many fields in New York[xxi].

Play areas

The data for the play areas have also been imported as a shapefile from the official geo data website of New York City[xxii]. It included the geometry of the park and other meta data, which were of no use for us.

In the beginning, the number of playgrounds in a certain NTA region was calculated by joining the NTA with the play area table using ST_INTERSECTS. As a further step the same procedure was repeated to find play areas near the zones using ST_DISTANCE within 500 meters, as it seemed reasonable to considerer those close enough to increase the rating of these areas. The rating was put together by

weighting the parks in the NTA area three times as much as the ones in the adjacent regions[xxiii]. Finally, the rating was calculated by considering the total number of playgrounds in and nearby the zones, as well as the zones area[xxiv]. Those steps were all performed successively because of slow database queries.

Parks

The data for the parks and recreational areas[xxv] of New York City was also downloaded as a shapefile and imported into the PostGIS database.

The table consists of several columns, including the geometry, location, name and other attributes. Firstly, the total area of all parks in a particular NTA zone was calculated by joining the previously imported park table with the NTA area table. For this, the PostGIS functions ST_AREA, ST_INTERSECTS and ST_INTERSCETION were used. As in this case, not only the parks in a certain NTA region, but also the ones close by seemed to be relevant. A buffer of 500 meters was added around the zones to determine the adjacent recreational areas. Afterwards the total area of all parks was summed up and divided through the area of the NTA zones. For this calculations, the database functions ST_BUFFER and ST_DIFFERENCE were used[xxvi]. In the end this number was normalised[xxvii] the same way as all datasets have been. These procedures were all executed in small steps.

Restaurants

To get data about the restaurants in the city, a csv- file [xxviii] was downloaded and imported into the database. That table included an address with a street name and zip code.

For further processing, a geometry data entry was needed to determine the exact location of the restaurants. A second data set[xxix], which mapped addresses to GPS points, was imported. Irrelevant information, like building numbers and other meta data, was removed. After eliminating multiple entries to one restaurant in the restaurant table, the two relations were joined using the address. However, this only worked in about half of the cases, because of differences in the street names. For example, there were additionally building numbers in the name or abbreviations were used. As there did not seem to be a better solution, the Google Geocoding API was used again. For that, the not matched restaurants were extracted into another table and exported as csv files.

This time a Java program[xxx] was developed. The program reads the addresses from the csv file and sends a request to the API, which returns the latitude and longitude, then saves the results in another file. A Problem with the Google API is the necessity of using an API key for the request. The number of

requests per key was limited to about 2500 requests per day. To avoid this limit, more keys had to be requested from the Google API site. With that solution, only about three percent of the restaurants could not be matched to a location and must be omitted for further processing.

Afterwards the file with the geo coded data was imported and added to the table with the other restaurants. Finally, the NTA zones table was joined with the restaurants using the ST_INTERSECTS function to get the total number of restaurants for each NTA neighbourhood[xxxi]. Resulting, the id of the NTA zones and the rating of the total number of restaurants in the particular area were used[xxxii].

Developing the web application

Frontend

General Website Structure

We did not use a backend application sever to keep the website setup and structure simple. The whole website is built as a single html page, that executes all code necessary to display the final map on the client's browser. The different steps and sections on the page are implemented by hiding and revealing parts of the web page.

We used various libraries for the website. The main ones are the JavaScript library JQuery for working with the HTML elements on the page and the SemanticUI CSS framework to style the page without much custom CSS code.

One thing we could see, when building the application, is that modern website building tools like webpack[xxxiii] or gulp[xxxiv] and a backend application sever should be used if the project was any bigger or more complex.

GeoServer Setup

We used GeoServer to access our map data. First we joined all individual rating views, that were created, into one table. This table contains all zones with their metadata, geometry and ratings[xxxv]. We decided to use a table and not a view at this point, as this acts as our caching layer. The table is created once, which is time consuming, as all spatial operations to calculate the ratings have to be evaluated. Accessing the combined table from the web application is very fast.

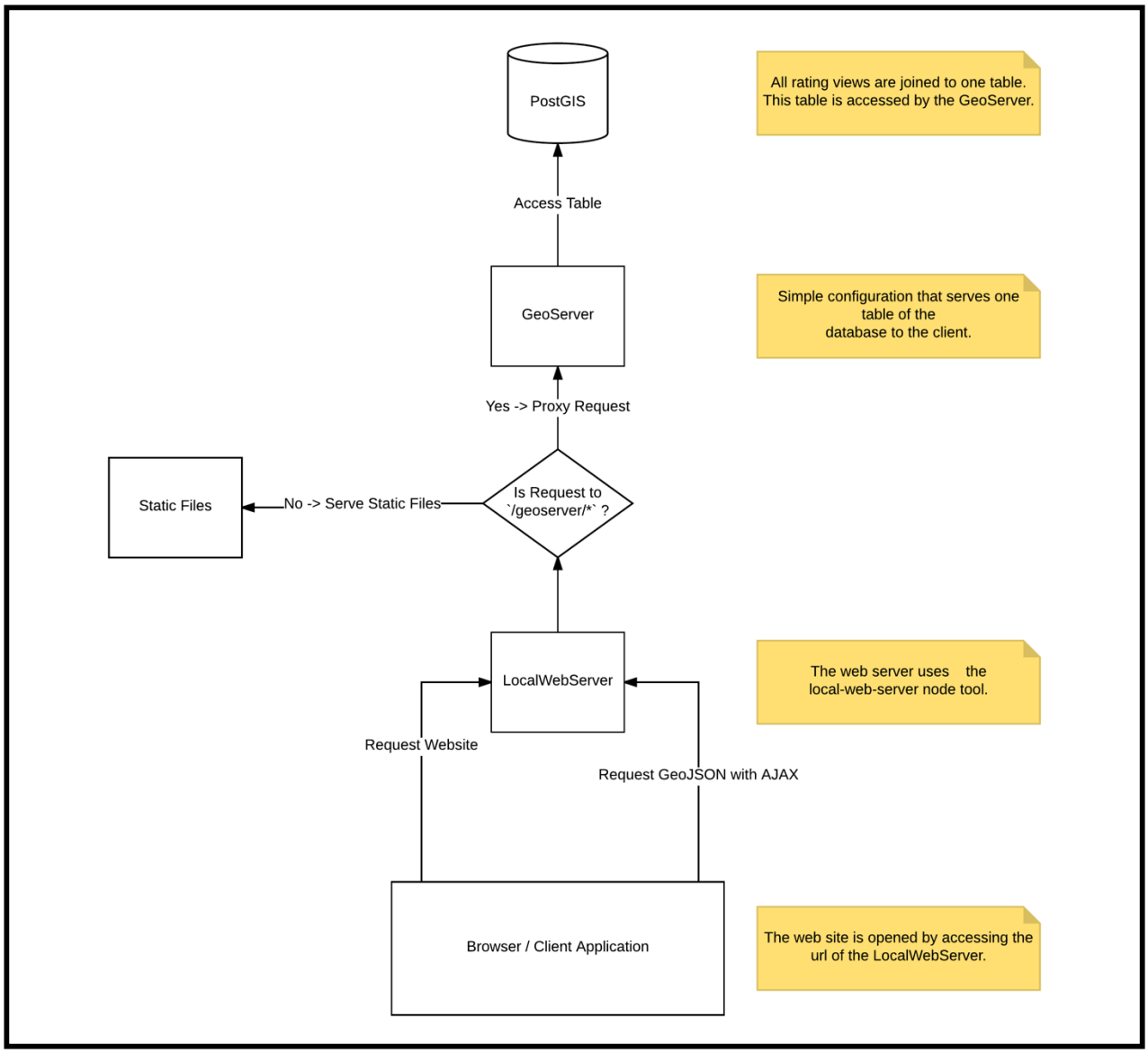

Figure 4: Basic Deployment Structure

After that, we did setup a GeoServer and connected it to the PostGIS database, containing the combined table. Finally, we setup a layer that allows us to access the table from our web page code. One challenge at this point was the same-origin policy[xxxvi], which prevents the clients JavaScript code from loading JSON data from the GeoServer.

To solve the issue one can either setup the GeoServers configuration to allow cross-origin resource sharing (CORS) or make sure that the GeoServer's IP and port are the same as the webpages. We decided to use the second method, as the GeoServer configuration is not trivial and makes it harder to setup the project. To get the websites IP and port to be the same as the GeoServer ones, we setup a webserver that delivers our static website and acts as a proxy for the GeoServer. To do so we used a npm tool called 'local-web-server', as it is very easy to install and use. After installing it, we can start the server using the command `ws --rewrite '/geoserver/* -> http://localhost:8080/geoserver/$1'`.

Note the --rewrite option, as it proxies all requests made to '/geoserver/' to the actual GeoServer[xxxvii]. We recommend to look at the 'local-web-server' tool for any work with small web projects, as it is also very useful and simple to serve some local files to the browser.

Map Display

With the GeoServer setup, we simply used a JQuery ajax request to load the maps data as GeoJSON from the GeoServer. We decided to load the map as GeoJSON, as it allows us to build more interactive content, e.g. style the map dynamically. It also gives us access to all the rating values that are included in the table setup above. To request the map data as GeoJSON instead from the GeoServer we had to add '&outputFormat=application%2Fjson' to the request that accesses the data[xxxviii].

After the data has been loaded from the server, it can be used to display the different zones on the map. Colours and styling of the zones are based on a computed rating value, more on that in the rating chapter.

Besides that, map setup and interaction is mostly cosmetic, like restricting the map area to New York, clicking interactions and hover effects.

Locations of Interest

The user can input a route that he typically follows daily. The route always starts and ends at the centre of a NTA zone. This route is then used to calculate the walking distance for each zone. To implement this, we had to perform two steps. First we had to let the user input his locations of interest, then we had to use these inputs to calculate the walking distances.

To let the user input an arbitrary number of locations we used JQuery, SemanticUI search and some custom data structures. To execute the actual search for a specific location of interest we implemented some of Semantic UI's search methods. In this method, we did use the geocoding API of Mapbox to get a suggestion of locations based on the user's current input. The auto completion is done by SemanticUI. After the user has selected the location he was looking for, we use the coordinates given by the Mapbox geocoding to display a marker on a small map next to the search input. This helps the user to visually see what he selected. We also save all selected places in an array for later usage.

After the user completes the configuration, we need to get the walking distances for the given routes. To do this we need to calculate the distance of the routes starting and ending at each of the 194 zone centres. Because of the amount of zones, we could not use the usual directions API by Mapbox, as this would result in 194 requests, which take too long and also exceed the API limit of Mapbox[xxxix]. To solve

this, we used the Mapbox distance API, which is currently in preview[xl]. The API allows to send an array of up to 100 GPS locations and responds with a matrix containing the walking distances between all the points. The matrix describes a graph with weighted edges based on the walking distances.

In our use case, we sent the places entered by the user and the centres of the NTA zones in batches to Mapbox, as we could not send all 194 zones at once. To coordinate these asynchronous requests, we used promises. We then used this matrix to calculate the routes starting/ending in each individual zone centre and saved the results from the requests into the feature properties of each zone. We then normalise the values, so they are between zero and one. This allows us to use the dynamically calculated walking distances just like the other static ratings from GeoServer.

The distance API works really well for us and we would absolutely recommend it for all problems where walking distances between many points are needed. Note that at the time of writing you must send an email to the support to access the distance API, otherwise all requests will fail with an error code.

Preselection

At the beginning of the application the user is asked to answer a few question that allow us to select an appropriate NTA zone. For that we determine a few peer groups, like parents or students, who may have different preferences for their living environment.

So for instance, we thought that a student might like to live in a vibrant and central area, not too far away from a university, but cannot afford a high rental price. In comparison the family may prefer living in a quieter area, close by schools and playgrounds for the children.

Altogether, depending on the user's selections, we pre-select appropriate values for weighting different NTA zone ratings. These values can be adjusted by the user later on using sliders in the sidebar.

Rating

Whenever the user changes the weighting values using the sliders, we trigger a rating function that calculates a numeric rating value for each zone. We use these ratings to colour the zones and display a sorted list of results in the sidebar. This way, the user can change the rating inputs at any time and the changes are reflected on the user interface.

The user can select serval different weighting values:

- Locations of Interest
- Live central or outside
- Live near a university
- Importance of parking situation
- Live near schools
- Live near many restaurants
- Can afford higher rental costs
- Live in a vibrant or quiet area
- Live near parks
- Live near play areas
- Live near subway stations

Depending on the importance of the criteria, points are added to a NTA zones overall rating. This is calculated by the factor the user selects with the sliding bar, ranging from -1, when a certain point is unwanted, zero, when the question is irrelevant to the user, or one, when it's important for the user. Additionally, we programmatically added weight factors to some points, that we thought are more important. So, for example the short distance to the user's locations of interest and the subway situation are considered very important, so the result is multiplied by a higher factor.

Calculation of Overall Rating

To calculate the overall ratings for the different zones we used the valuation function shown in figure 5.

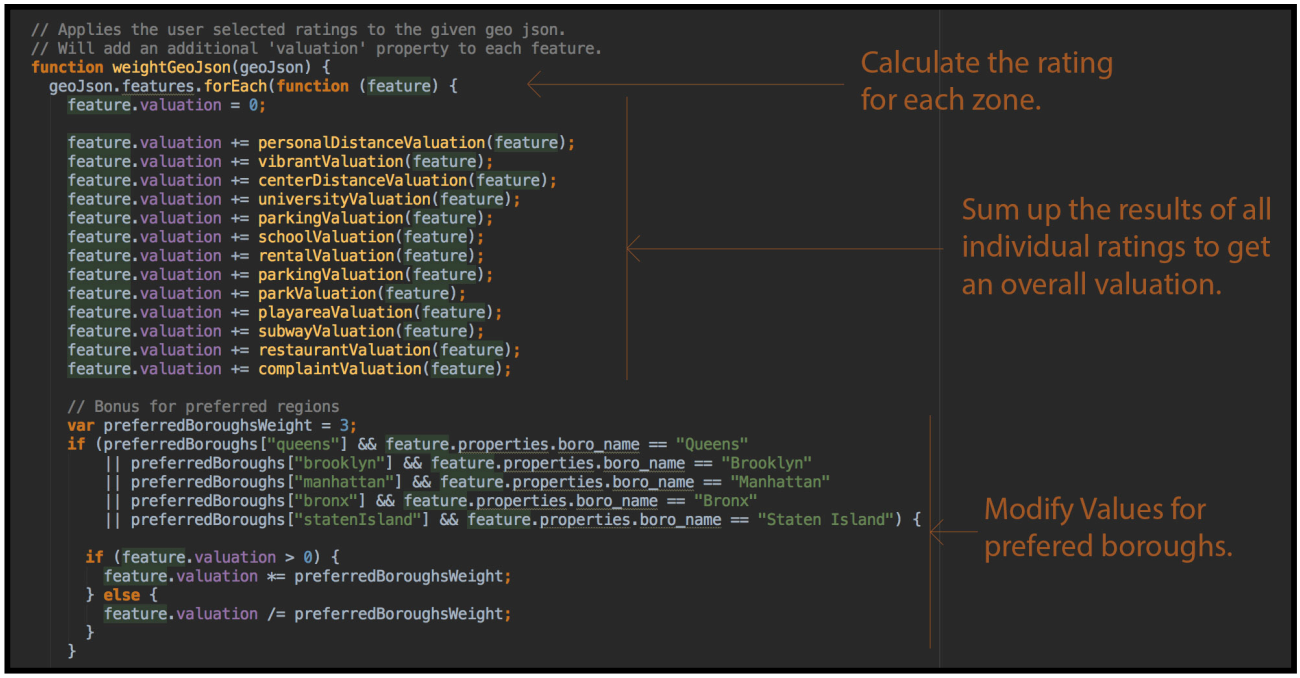

Figure 5: Overall Rating Function

The function iterates over the geoJson of the zones and assigns a new property, called valuation, to each zone. The individual valuations are calculated by adding up the valuations of the factors defined by us.

```
function rentalValuation(feature) {
  // Invert value -> lowest prices should give positive rating
  var rentalRating = 1 - feature.properties.rental_rating;

  // Custom weighting -> how important is this rating?
  var weighting = 5;

  return rentalRating * rentalImportance * rentalImportance * weighting || 0;
}

function schoolValuation(feature) {
  var schoolRating = feature.properties.school_rating;

  // Custom weighting -> how important is this rating?
  var weighting = 1;

  return schoolRating * schoolImportance * weighting || 0;
}
```

Figure 6: Individual Rating Functions

Most ratings could be calculated by applying a weighting factor to the rating value previously calculated in the database. For example, figure 6 shows the school and the rental valuation. The school rating was simply weighted by a factor of one, the rental rating was inverted and weighted by a factor of five, as lower rental costs are considered better and rental prices are in our opinion more important than most other factors.

Most ratings have a similar structure to the functions in figure 6, the main work was to determine good weighting values.

Example Rating: Subways

Some ratings where more complicated than the ones shown in figure 6. One example of this where the subway ratings.

```
function subwayValuation(feature) {
  // Distance to next subway -> 1 is good, 0 is no subay station
  var subwayRating = feature.properties.subway_rating;

  // No subway station in region, special calculation
  if (subwayRating < 1) {
    subwayRating = (subwayRating * subwayRating * subwayRating) / 3;
  }

  // Custom weighting -> how important is this rating?
  var weighting = 1.5;

  return subwayRating * subwayImportance * weighting || 0;
}
```

Figure 7: Subway Rating Function

We discovered that areas without a subway are still rated very positive (figure 8), when simply taking the subway distance as base for the rating value. To determine the ratings for areas with no subway entry, we used the following formula:

$$f(x) = 1 \qquad \text{for areas with subway}$$

$$f(x) = \frac{x^3}{3} \qquad \text{for areas without subway}$$

This left us with the result shown in figure 9.

Note that all input values are between zero (very far away from subway entrance) and one (has a subway entrance).

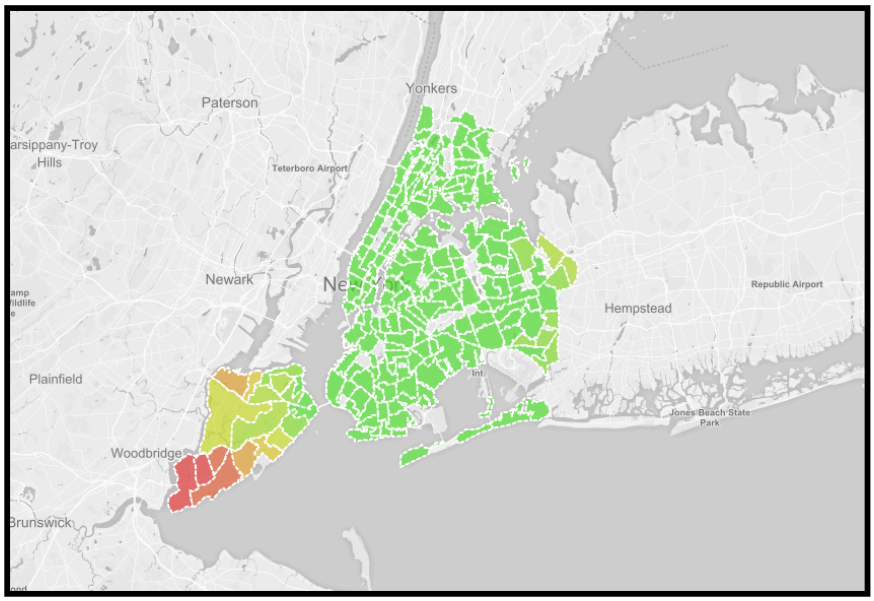

Figure 8: Subway Ratings with Simple Function

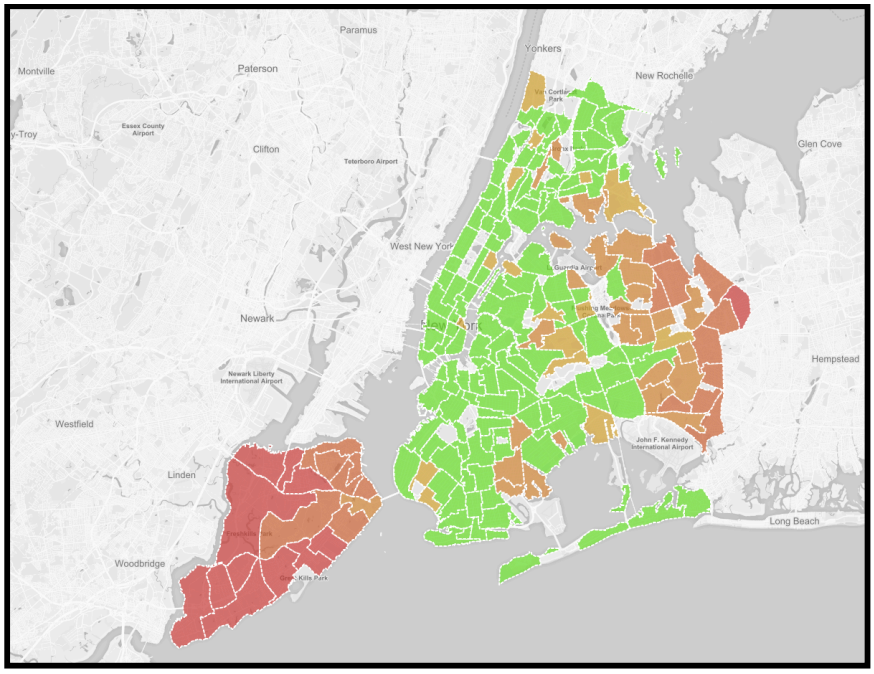

Figure 9: Subway Ratings with Improoved Formula

Conclusion

After testing the final application serval times using different personas, we came to the conclusion that the web app is delivering accurate results. All factors seem quite balanced and affect the results in a meaningful and logical way.

Our prototype shows that the application might be useful to people moving to New York City, but could be improved by more accurate data and more relevant datasets.

We were able to efficiently solve all occurring problems by doing some more research, as we already had acquired a good understanding of the basic techniques in the lecture. It was fun to see the theoretical aspects implemented in an interactive application.

Pageflow of the Web-Application

Start Page

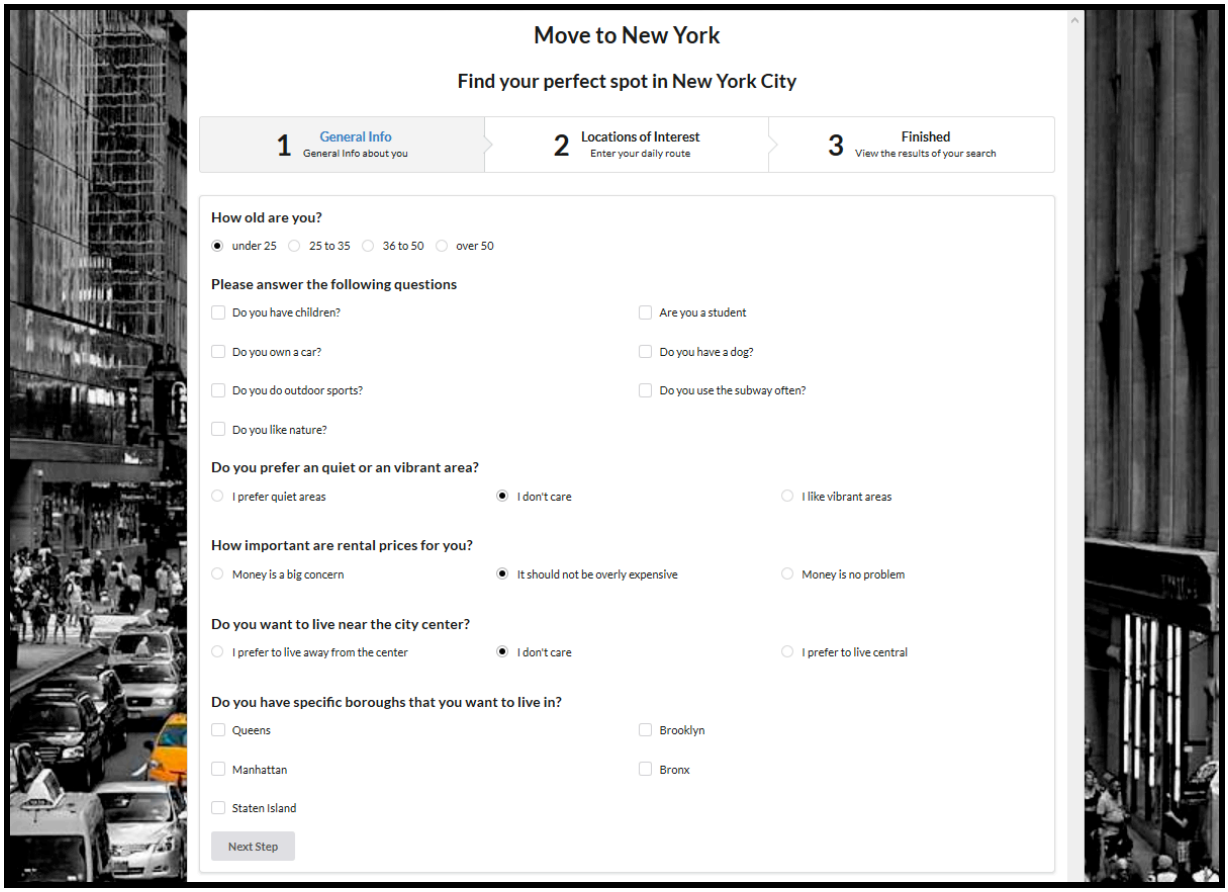

Figure 10: Start page

Define personal preferences

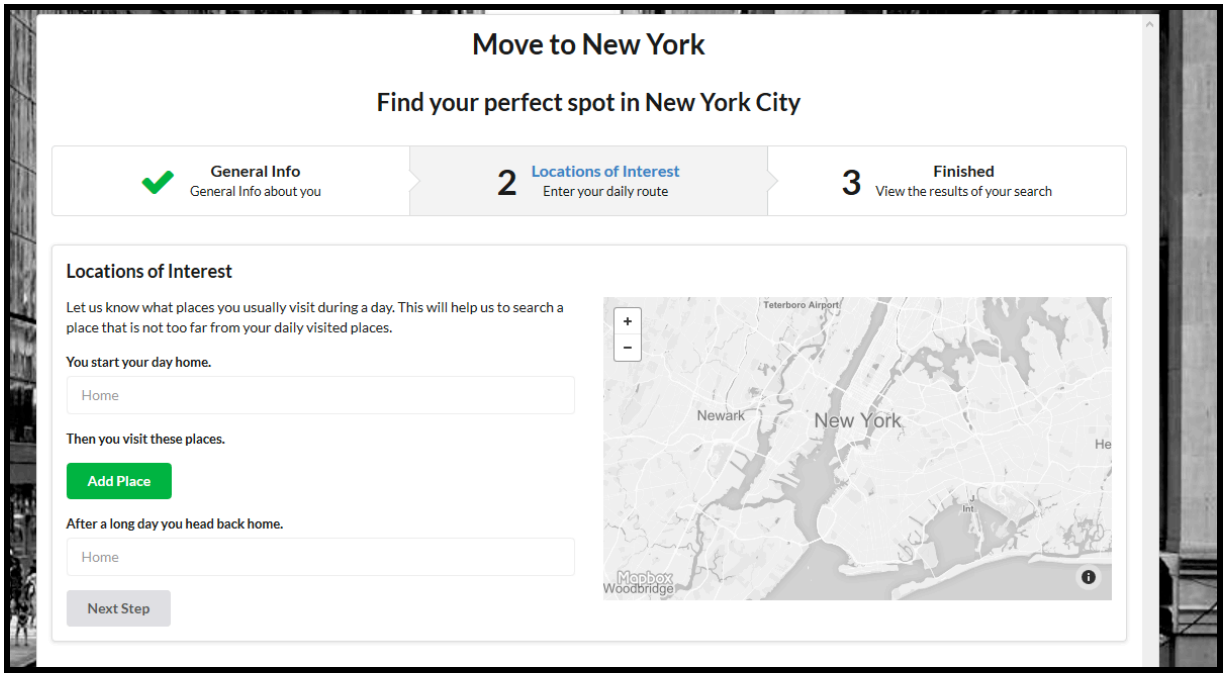

Figure 11: Personal preferences

Set locations of interest

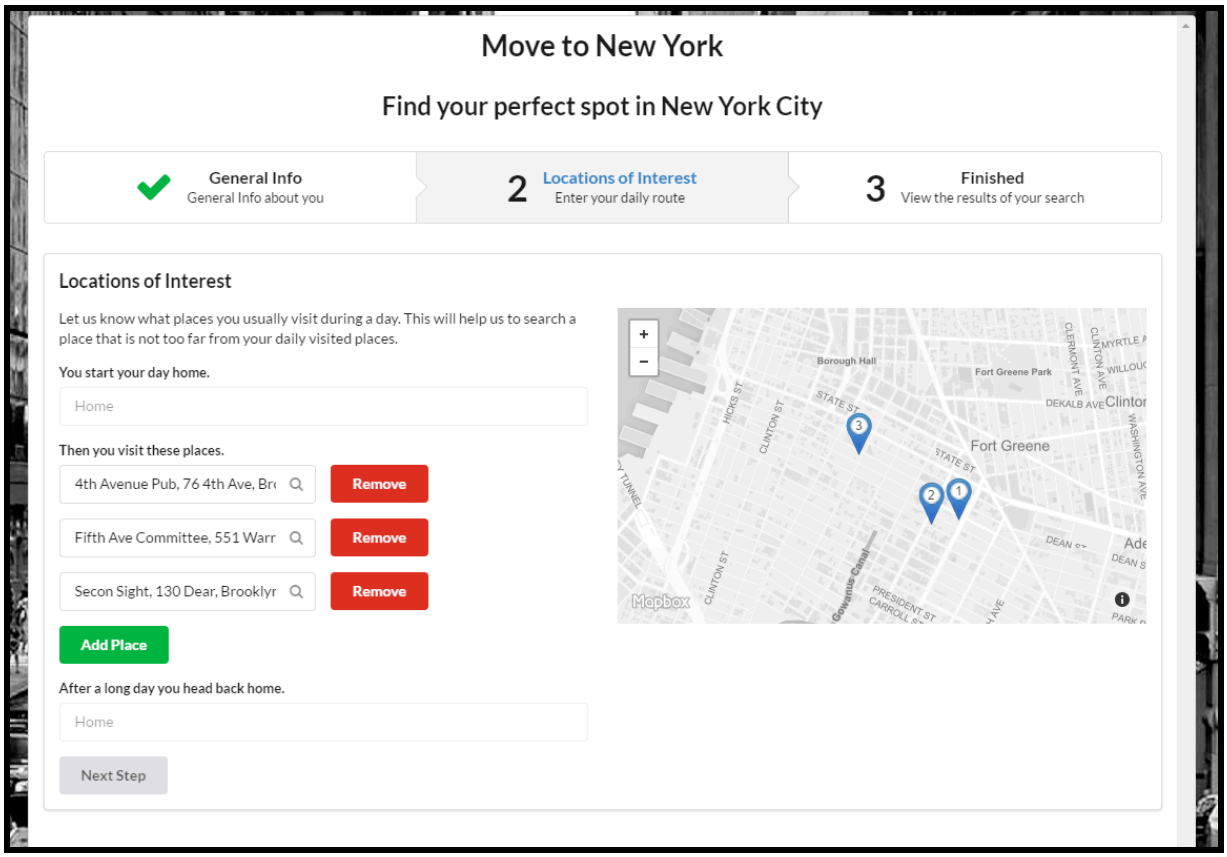

Figure 12: Locations of interest

Result page

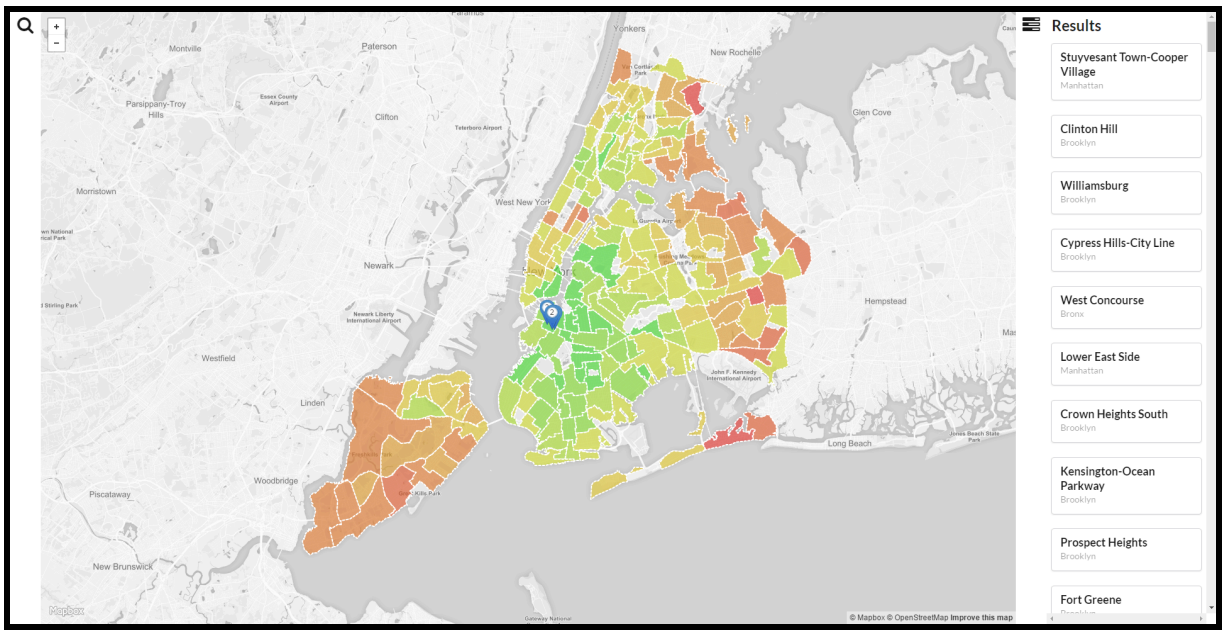

Figure 13: Result Page

Result page with slider to redefine the search

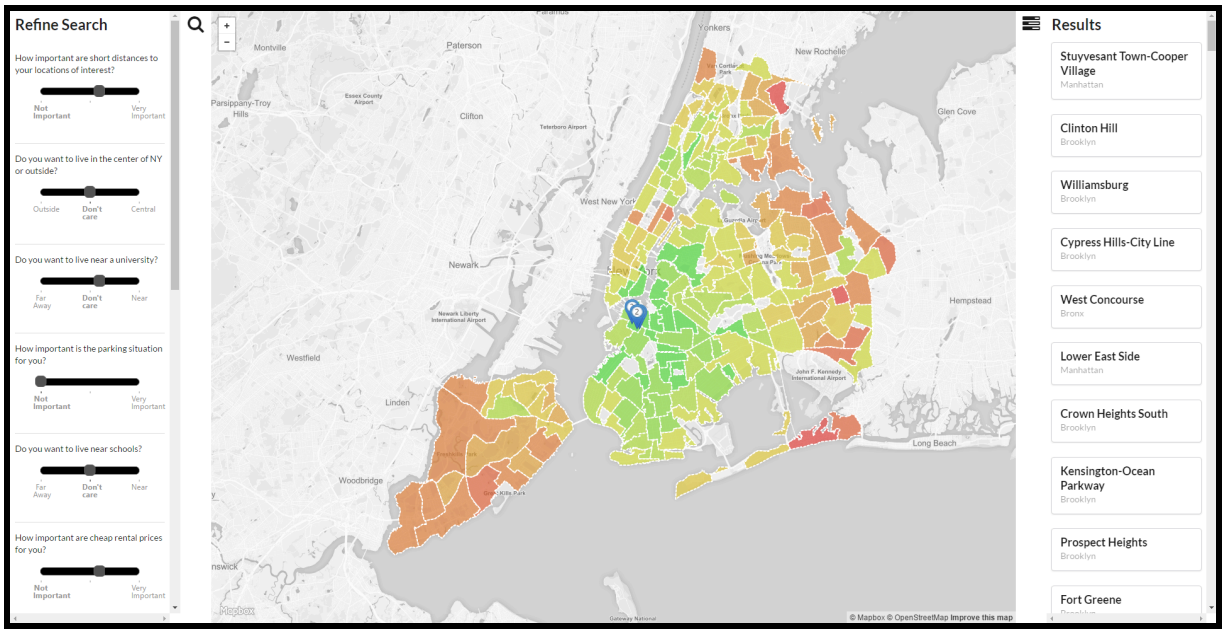

Figure 14: Redefine search

[i] https://data.cityofnewyork.us/City-Government/Neighborhood-Tabulation-Areas/cpf4-rkhq
[ii] https://data.cityofnewyork.us/Education/School-Point-Locations/jfju-ynrr/data
[iii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/FuFlo%20Data/TABLES/4.publicSchoolPoints/SchoolData.sql
[iv] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/FuFlo%20Data/TABLES/4.publicSchoolPoints/view.txt
[v] https://data.cityofnewyork.us/Education/Colleges-and-Universities/4kym-4xw5
[vi] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/FuFlo%20Data/TABLES/2.colleguesAndUniversitys/view.txt
[vii] https://data.cityofnewyork.us/City-Government/Parking-Lot/h7zy-iq3d
[viii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/FuFlo%20Data/TABLES/3.parkintLot/ParkingData.sql
[ix] https://data.cityofnewyork.us/City-Government/New-York-City-Population-By-Neighborhood-Tabulatio/swpk-hqdp
[x] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/FuFlo%20Data/TABLES/6.population/populationData.sql
[xi] https://data.cityofnewyork.us/Public-Safety/NYPD-Complaint-Data-Current-YTD/5uac-w243
[xii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/FuFlo%20Data/TABLES/7.complaint/complaintData.sql
[xiii] http://www1.nyc.gov/site/finance/taxes/property-cooperative-and-condominium-comparables.page
[xiv] https://github.com/FlorianFusseder/SpatialDatabases/tree/master/Geocoder/Geocoder_mapbox
[xv] https://github.com/FlorianFusseder/SpatialDatabases/tree/master/Geocoder/Geocoder_google
[xvi] https://data.cityofnewyork.us/Transportation/Subway-Stations/arq3-7z49
[xvii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/subway/subway.sql
[xviii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/subway/subwaydistance_table.txt
[xix] https://data.cityofnewyork.us/Recreation/Map-of-Soccer-and-Football-Fields/qqsi-vm9f
[xx] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/soccerfields/soccerfields.sql
[xxi] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/soccerfields/soccerfield_data.txt
[xxii] https://data.cityofnewyork.us/City-Government/Play-Areas/8fhn-c4v3
[xxiii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/playgrounds/playgrounds.sql
[xxiv] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/playgrounds/neighbourhood_playground_table2.txt
[xxv] https://data.cityofnewyork.us/City-Government/Parks-Properties/rjaj-zgq7
[xxvi] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/parks/park2.sql
[xxvii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/parks/neighbourhood_parks_table2.txt
[xxviii] https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/xx67-kt59
[xxix] https://data.cityofnewyork.us/City-Government/NYC-Address-Points/g6pj-hd8k
[xxx] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/restaurants/geoCoding.java
[xxxi] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/restaurants/restaurants2.sql
[xxxii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/data/restaurants/restaurants_table2.txt
[xxxiii] https://webpack.github.io/
[xxxiv] http://gulpjs.com/
[xxxv] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/combine_data/combined_view.sql
[xxxvi] https://en.wikipedia.org/wiki/Same-origin_policy
[xxxvii] https://github.com/FlorianFusseder/SpatialDatabases/tree/master/frontend#setup-geoserver
[xxxviii] https://github.com/FlorianFusseder/SpatialDatabases/blob/master/frontend/script.js#L51
[xxxix] https://www.mapbox.com/api-documentation/#directions
[xl] https://www.mapbox.com/api-documentation/#distance