Wrangling OpenStreetMap Data with MongoDB

Data Wrangling Final Project

by Dirk Kalmbach

(May 2015)

Abstract: All statistical analyses (as well as some further explanation) can be found on the corresponding iPython *Notebook Wrangling Open Street Map Data - Additional Information*.
All analyzes are based on the dataset `bonn_osm.xml`.

1 Introduction

In this second course project of the Udacity Nanodegree Data Analyst we were asked to choose a map area from open street map, wrangle it, insert it into MongoDB and do some basic analysis.

I decided to select the city boundary of Bonn, Germany.

Bonn is one of Germanys oldest citys, founded in the first century BC as a Roman settlement. It was the home town of Ludwig von Beethoven and the former capital of (West) Germany during the Cold War Period. With its old university, ... Bonn has a population of 311,287.

The main reason for my choice was that I have been living in Bonn for several years. During this time I got to know and appreciate the city.

2 Problems encountered in the Data

German Umlaute

The german language consists of so called „Umlaute“, special characters which does not exist in other languages and are not part of the ascii-alphabet. Because of this, „umlaute“ are frequently replaced with their two-letter combination (see Table 1).

Umlaut	Substitute	Pronounced
ä, Ä	ae, Ae	[ɛ]
ö	oe, Oe	[œ]
ü, Ü	ue, Ue	[y]
ß[1]	ss	[s]

Table 1: German Umlaute

For reasons of harmonization, all tags which contain Umlaute should be either in its original form (e.g.: „Straße“) or in its subtitute form (e.g.: „Strasse“). For this, the function `findUmlaute()` finds all occurrences of German Umlaut-substitutes (i.e.: tags containing „ae“, „ue“, etc.).[2] The Problem with this approach is, that some german words contain same letter-combinations like the Umlaut-substitutes (e.g.: dass [engl.: that]).

This means, I definetly needed to look over the resulting list and needed to decide manually which of these words does not contain an umlaut.

Unfortunately, the list contained 11055 hits - too many for manually correcting.

As a consequence, I decided to correct all umlaute with its substiute form

1. No capitalized form, as words cannot begin with ß.
2. See *Appendix A.3* in the corresponding iPython Notebook.

(instead of correcting all umlaut-substitutes with its umlaute).[3]

Abbreviations

Like in the U.S. and also described in the class[4] it is common practice in Germany to abbreviate street names. A typical abbreviation for „Straße“ (engl.: street) could be either „Str.“ or „Str“. To receive unique results I wrote a method to convert alle short forms of the german word „Straße“. The challenge here was that (unlike how we did this in class), I needed to look for abbreviations in the beginning or in the middle of a street name, as street names in german can begin or contain the word „Straße“ (e.g.: „Str. der Republik“). I also needed to consider urls and email-adresses (which should not be corrected, of course).

Size of the dataset

One final challenge was the execution time: on a 2.26 GHz MacBook with 4 MB RAM it took about 25 minutes to execute all the above described auditing and cleaning steps.[5]

3 Overview of the Data

Basic Statistics

Table 2 show the size of the different files:

Name	Description	Size
`bonn_osm.xml`	Downloaded file from Open Street Map	181.7 MB
`wrangled_OSM.json`	Wrangled file before import into MongoDB	192.2 MB
`db.Bonn`	Inserted file in MongoDB (size of the collection)	244.0 MB

Table 2: Size of Files

The average size of a document in the collection is 299 bytes. The collection contains of 855049 documents with 717138 number of nodes and 137777 number of ways.

User Statistics

1038 different users contributed to create the map:

```
> db.Bonn.distinct("created.user").length
1038
```

Figure 1: Screenshot of Terminal (mongo)

And the Top Three contributing users are: ARWIE (5555 entries), dschuwa (5504 entries), and dalchimsky (4852 entries).

List of Sightseeing Places

Bonn with its colorful history and large variety of museums is a favourite destination for tourists. The tag tourism in the wrangled dataset provides information for tourists. The study of this tag led to the conclusion that two values, namely *attraction* and *museum* provides sightseeing information.

For example, the following pymongo query provides a list of museums:

```
museums = db.Bonn.aggregate([
        {"$match":
         {"tourism":{"$exists":1}, "tourism":"museum"}},
        {"$match":
         {"name":{"$exists":1}}},
        {"$project":{"name of museum":"$name", "_id":0}}
    ])
```

In total, there are 67 attractions and 35 museums in Bonn.

3. See *3.2.1 Replacing German Umlaute* in the corresponding iPython Notebook.

4. Lesson 6-11 *Improving street names* of the course.

5. I printed out the execution time for most of the code in the corresponding iPython Notebook.

4 Other Ideas about the dataset

For this final task, I decided to create a function which will allow a visitor of Bonn to visit several points within the city boundary starting from one choosen place (e.g.: central station or a hotel address).

For example: a tourist might have a list of touristic attractions he would like to see starting from his hotel. As this tourist does not like to walk long distance he/she prefer to become a map of the attractions and the shortest way between them.

The underlying alghorithm uses the euclidian, i.e. the shortest direct distance between two points. It produces a list of coordinates (value pairs of longitude and latitude) in which the points are stored in order of walking. I.e.: the first entry in the list is the starting point (

I decided to choose telephone boxes, rather than touristic points of interest or pubs, as telephone boxes are widely spread in the city and not agglomerated in the centre. However, this code works simply as a prototype e.g. to create individualized maps for tourists and any kind of places can be easily inserted in the `pymongo` query.

The following image shows the result of the function:

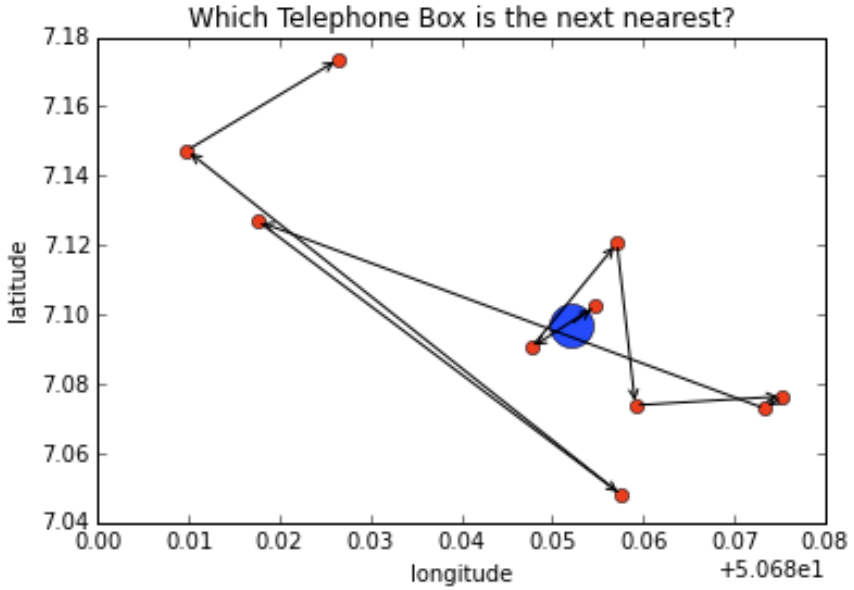

Figure 2: Shortest distance between red points (randomly selected telephone boxes) starting from blue point (Bonn Central Station)

The white space represents the city, the blue point stands for the central station, and the red points are telephone boxes. The black arrows show the way a visitor should walk in order to minimize the walking distance.

However, although the alghorithm seem to find in most cases the next nearest point, in a few cases the alghorithm does not work proberly. For example: instead of pointing from A to B the alghorithm comes to the result that C is the next nearest point to A.

Another drawback of course is that this approach simply uses the directest linear distance between two points. It does not consider ways, streets, or even public transportation. So the actual walking distance would be definetly longer. But as Bonn like most old European cities consists of a lot of small streets, alleys, lanes, ... with a lot of intersections, this probably does not carry weight.

Nevertheless, Bonn is crossed by the river Rhine, i.e. a big improvement of the alghorithm would be to consider bridges.

Another useful implementation would be to calculate the distance from one touristic attraction to another one. For this I implemented a function which computes the euclidian distance between museums and/or tourism attractions in kilometer and miles.[6] The user can choose two museums from a list of all museums in Bonn and receives the distance between these two museums.

For example: the distance between the Arithmeum and the Siebengebirgsmuseum, two popular museums in Bonn is 10,4 km.

The benefits of these implementations are obvious: the city of Bonn could provide customized sightseeing information for tourists. Tourists visiting Bonn could get a list of attractions from which they choose their favourite destinations and the system calculates the shortest „walking path“ together with some nearby-information for every attraction. This could be implemented in a mobile app, web application or a terminal located at the tourist information center.

6. See *5.2.2 Calculate Distances* in the corresponding iPython Notebook.