Wrangling Open Street Map Data

Thomas Roscher

https://github.com/ThomasRoscher/Udacity_da_data_wrangling_Python_SQL

Introduction

OpenStreetMap (OSM) is a collaborative project to create a free editable map of the world. For this project I downloaded all available data for Friedrichshain-Kreuzberg which is a district of Berlin. I then audited and cleaned the data, loaded into a SQL database and eventually ran some queries. All code for this project is available on https://github.com/ThomasRoscher/Udacity_da_data_wrangling_Python_SQL

OSM data structure

OSM data can be downloaded as XML file which is build around OSM'S core elements. Elements are the basic components of OpenStreetMap's conceptual data model of the physical world. They consist of:

- nodes (defining points in space),
- ways (defining linear features and area boundaries), and
- relations (which are sometimes used to explain how other elements work together).

All of the above can have one or more associated tags (which describe the meaning of a particular element).

Auditing OSM data

Auditing focused on systematical coding issues not on individual errors such as typos. Since the OSM data is quiet comprehensive (I got 2352 unique tags), I focused one just a handful of tags which are related to address or contact information. More precisely I investigated the following seven tag values:

- addr:country
- addr:city
- addr:suburb
- addr:postcode
- addr:street
- contact:phone
- contact:email

To investigate the quality of the coding, I wrote three audit functions of which one is shown below. Those function do not store the data in memory but rather use iterative parsing to process the map file. The function below is used if one expects an exact match between coded values and one or more expected values. The other two examine if the coded values end or begin with an expected pattern. The general purpose for this approach is to filter "save" values so that one ends up with a much shorter amount of observations which may or may not have systematic errors.

```
def audit_value_isnot_x(filename, vvalue, expected):
    attrib_v = []
    for event, element in ET.iterparse(filename):
        if element.tag == "tag":
            for tag in element.iter("tag"):
                if element.attrib['k'] == vvalue and element.attrib["v"] not in expected:
                    attrib_v.append(element.attrib["v"])
    k = Counter(attrib_v).keys()
    v = Counter(attrib_v).values()
    dic = dict(zip(k, v))
    dic = sorted(dic.items(), key = lambda x:x[1], reverse = True)
    return dic
```

Surprisingly, much of the data was in pretty good shape. The following were pretty much error free tags:

- Country: All but one were coded correctly as “DE”
- City: All but one were coded correctly as “Berlin”, no abbreviations such as “Bln.” were found
- Suburb: All were coded correctly as “Friedrichshain” or “Kreuzberg” IF tags were located in “Freidrichshain-Kreuzberg” (see below for further details)
- Zip-code: Most were coded correctly as one of the 12 valid zip-codes of the district (no special characters, transposed digits or wrong length were found)
- Street name: Mostly coded correctly (no abbreviations such as “Str.”, small letters at the start, or “ss” instead of the German letter “ß” were found)
- Email: All entries end with a valid “.something”

As always there were still some issues, though. First of all, the map covers much more area then the district Friedrichshain-Kreuzberg, as shown in the unique values of suburb and zip-code. Those errors are surely no individual mistakes because we are talking roughly about 20.000 tags. So I guess when downloading the map wrong longitude and latitude values for borough borders are stored. Oddly enough the borders are correctly displayed on the website. Regarding tag values phone number turned out to be a pretty big mess. The correct and expected pattern would be the German prefix (+49) then the Berlin prefix (30) followed by the actual phone number or prefix (+49) followed a mobile number(excluding the zero). I identified about half a dozen different patterns including no prefixes just one prefix or forgetting to drop the zero from the mobile number as well as random white-space and special characters. The cleaning function below corrects many numbers to a unified pattern. Note that, the functions will not turn total rubish input into valid numbers rather it unifies the format and adjusts/adds international and local prefixes if required. Note, that I wrote three additional cleaning functions (city, country, and streetname) which can be founf on github.

```
def update_phone(phone):
    prefixes = ["1", "2","3","4","5","6","7","8","9"]
    phone = ''.join(e for e in phone if e.isalnum())
    if not phone.startswith("49") and phone.startswith("01"):
        phone = phone.replace(phone[0], '')
        phone = ''.join(("49", phone))
    if not phone.startswith("49") and phone.startswith(tuple(prefixes)):
        phone = ''.join(("4903", phone))
    if phone.startswith("030"):
        phone = phone.replace(phone[:3], "4930")
    if phone.startswith("49030"):
        phone = phone.replace(phone[:5], "4930")
    if phone.startswith("4901"):
        phone = phone.replace(phone[:4], "491")
    phone = "+" + phone
    return phone
```

Investigating the database

Next, all cleaning functions were incorporated in the process of transforming the XML file to a csv data, so that I got a couple of nice and clean csv files which the were uploaded into a SQLite database (see the parse_to_csv file one Github). Storing the files in the database was done with the function below.

```
def csv_to_sql(filename, connection, tablename):
    df    = pd.read_csv(filename, encoding = "utf-8")
    con   = sqlite3.connect(connection)
    df.to_sql(tablename, con, index = False)
    del df
    con.close()
```

Once this step was completed I investigated the data with some basic queries. More precisely I checked the:

- number of nodes (935663)
- number of ways (135634)
- number of unique users (3042)
- number of unique users and their contributions (top 3 made about 50% of entries and top contributer seems to be a bot)
- number of rare users (1580 with less then 4 entries)
- top ten amenities (bench, waste basket, bicycle parking)
- number of amenities related to drinks and food (only 99!!)
- entries in nodes over time (increased steadily from 2007 to 2015 then dropped)
- entries in ways over time (increased steadily from 2007 to 2015)

Below you see all corresponding SQL querries that I used.

```
import sqlite3

# check the tables in the database
con = sqlite3.connect("fk_map")
cur = con.cursor()
cur.execute("SELECT name
             FROM sqlite_master
             WHERE type='table'
             ORDER BY name;")
available_table = (cur.fetchall())

# number of nodes
cur.execute("SELECT COUNT(*)
             FROM nodes;")
number_nodes = (cur.fetchall())

# number of ways
cur.execute("SELECT COUNT(*)
             FROM ways;")
number_ways = (cur.fetchall())

# number of unique users
cur.execute("SELECT count(DISTINCT user) as num
             FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways);")
unique_users = (cur.fetchall())
```

```
# number of unique users and their contributions
cur.execute("SELECT user, COUNT(*) as num
             FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways)
             GROUP BY user ORDER BY num DESC;")
unique_users2 = (cur.fetchall())

# number of rare users
cur.execute("SELECT count(user) as num
             FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways)
             GROUP BY user HAVING num < 4;")
rare_users = (cur.fetchall())

# top ten amenities
cur.execute("SELECT value, COUNT(*) as num
             FROM nodes_tags
             WHERE key='amenity'
             GROUP BY value
             ORDER BY num DESC;")
top_amenites = (cur.fetchall())

# drinks and food
cur.execute("SELECT COUNT(*) as num
             FROM nodes_tags
             WHERE value='cafe' OR value='fast_food' OR value='bar' OR value='pub'
             OR value ='restaurant';")
goingout = (cur.fetchall())

# entries in nodes over time
cur.execute("SELECT COUNT(*), strftime('%Y', timestamp) as Year
             FROM nodes
             GROUP BY year
             ORDER bY year;")
over_time_nodes = (cur.fetchall())

# entries in ways over time
cur.execute("SELECT COUNT(*), strftime('%Y', timestamp) as Year
            FROM ways
            GROUP BY year
            ORDER BY year;")
over_time_ways = (cur.fetchall())
```

Conclusions

Obviously, many of the investigated values being correctly coded is a good thing, especially if one considers that many different people enter the values. The reason for the surprisingly good data quality is propably that OSM has the so called Data Working Group (DWG) and offers a variety of data monitoring and error detection tools.

However, arguably there is plenty of room for improvement in other areas. First of all, the OSM people need to fix the borough borders. The other issues are (at least for Berlin) the quantity of the data and the inconsistency of the tags. Regarding the former there simply are not enough entries and many of the ones do not really matter. I mean who actually needs to know where the next bench is? Besides, finding only 99 food and drink locations is surely just a small fraction of the existing places. Regarding the latter one sees that many of the tag names/categories seems to be too similar or do even overlap (2352 unique tags are simply to much).

So what may be options to tackle those issues. For more homogenious tag categories OSM could provide stricter rules rules for tag names/categories or at least ensure that the ones that exists are better implemented. Therefore rules may not just be written down somewhere but rather should be applied directly on the user interface (comparable to valid passwort entries or the user can just choose between pre-defined categories etc.). However such a strategy bares the risk that thinks get to complicated or take too long so that potential contributars get deterred. Thus, keeping the right balance between input standards and user accesability is crucial.

Moreover, data showed that OSM attracted a core group of contributors and amassed a respectable library of content over the last years. Besides, OSM offers things such as "User Groups" to keep people engaged. However, the community seems to be highly skewed. Therefore, getting more people engaged is paramount to improve the quality of the map. Maybe an incentive system helps in this regard. How could such a system be designed? Arguably simple incentives such as running a contest and give out prizes is a bad idea because these kinds of reinforcement schedules produce high rates of behavior that spike just before the reward is given out. Afterwards, activity drops below its original levels and may take time to recover. Besides, incentives can also undermine quality if rules reward quantity over quality. A better approach is to track relevant actions, and then expose the resulting metrics to users. Just letting people know how they're doing can be a powerful incentive. Tell users how many contributions they've made and how long it's been since they made their last contribution. Let them compare their performance with those of other and the community at large. Give top performers the recognition they deserve. For example, clear some space on the home page to list the most prolific contributors. In short, understand their intrinsic motivations, and amplify them (see http://danielbayn.com/authentic-incentive-systems/)

Appendix

- fk.osm: 233 MB
- fk_map.db: 150 MB
- nodes.csv: 76 MB
- nodes_tags.csv: 29 MB
- ways.csv: 8 MB
- ways_tags.csv: 16 MB
- ways_nodes.cv: 30 MB