# Project3_LaimaStinskaite

March 12, 2017

## 1 P3: Wrangle OpenStreetMap Data

### 1.1 Map Area

**New York City, NY, United States**

OpenStreetMap: https://www.openstreetmap.org/relation/175905#map=10/40.6971/-73.9796

MapZen: https://mapzen.com/data/metro-extracts/your-extracts/b0778f38d0cd

I chose to download data of NYC, NY, USA. This city is full of different places: museums, restaurants, cafes, art studios, parks and so on.I think it will be interesting to investigate this data.

### 1.2 Problems encountered in the map

The size of original data file is 274 MB. To easier and faster investigate this data set we are executing **make_sample.py** script. It will make small sample of the original file. The size of a sample file is 5.53 MB (sampleNYC.osm).

To better understand the context of a sample file, let's parse it, find all available tags and their count. To do that we are executing **mapparser.py** script.

```
In [1]: from IPython.display import Image
        Image(filename='count_tags.png')
```

Out[1]:

```
C:\Jupyter\Project3\Project>python mapparser.py
{'member': 253,
 'nd': 31099,
 'node': 21974,
 'osm': 1,
 'relation': 41,
 'tag': 23218,
 'way': 3985}
```

Let's explore data more. Before we process our data and add it into our database, we should check the "k" value for each "tag" and see if there are any potential problems.

First, let's take a quick look of sampleNYC.osm file: we see a lot of tags "tag" with keys "k" which have lower case words and a lot of combinations like "cityracks.small" for amenity key.

```
In [2]: Image(filename='tags.png')
```

Out[2]:

```
<node changeset="1500261" id="419359838" lat="40.7017975" lon="-73.9868206" timestamp="2009-06-12T21:50:33Z" uid="134892" user="cityracks" version="1">
    <tag k="amenity" v="bicycle_parking" />
    <tag k="capacity" v="10" />
    <tag k="cityracks.large" v="2" />
    <tag k="cityracks.small" v="0" />
    <tag k="cityracks.rackid" v="215,4707" />
    <tag k="cityracks.street" v="Jay St" />
    <tag k="cityracks.housenum" v="100" />
</node>
```

Next, let's execute **tags.py** script to check if we have potential problems in "tags" or not. We will use 4 following regular expressions to check for certain patterns in the tags:

- "lower", for tags that contain only lowercase letters and are valid,
- "lower_colon", for otherwise valid tags with a colon in their names,
- "lower_dot", for otherwise valid tags with a dot in their names,
- "problemchars", for tags with problematic characters, and
- "other", for other tags that do not fall into the other three categories.

```
In [3]: Image(filename='potential_problems.png')
```

Out[3]:

```
C:\Jupyter\Project3\Project>python tags.py
{'lower': 9146,
 'lower_colon': 13563,
 'lower_dot': 380,
 'other': 129,
 'problemchars': 0}
```

As we see above, we don't have problems with chars in the keys for tags "tag".

### 1.2.1 Audit data

Now, let's execute **audit.py** script to find some problems with data.

```
In [5]: Image(filename='audit.png')
```

Out[5]:

```
# go through every tag in the file, find nodes and ways tags, find all street, postcode, housenumber, height keys and add all their values to appropriate dictionaries
def audit(filename):
    osm_file = open(filename, "r")
    street_types = defaultdict(set)
    direction_types = defaultdict(set)
    postcode_types = defaultdict(set)
    height_types = defaultdict(set)
    housenumber_types = defaultdict(set)

    for event, elem in ET.iterparse(osm_file, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            for tag in elem.iter("tag"):
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])
                    audit_direction_type(direction_types, tag.attrib['v'])
                elif is_postcode(tag) and (tag.attrib['v'] < '10001' or tag.attrib['v'] > '11102'):
                    audit_postcode_type(postcode_types, tag.attrib['v'], tag.attrib['v'])
                elif is_housenumber(tag):
                    audit_housenumber_type(housenumber_types, tag.attrib['v'])
                elif is_height(tag):
                    audit_height_type(height_types, tag.attrib['v'])
    return street_types, direction_types, postcode_types, housenumber_types, height_types
```

After execution we see that:

1.We have some inconsistencies in the **street types**:

- shortcuts, for example 'Ave': set(['5th Ave'])
- shortcuts for directions, for example 'W': set(['W 14th St'])

2.A lot of **postcodes** outside NYC. The range for NYC postcodes is 10001 - 11102. But we see that there are postcodes like '07302' is in our data (it is postcode for Jersey City, NJ) and '11378' (it is postcode for Maspeth, NY).

3.In the '**housenumber**' column we have values that can consist not only of numbers but characters or symbols like '730A', or '53-20', or '525 REAR', or '408 1/2'.

4.In '**height**' column we see that some numbers like float and some like integers, for example we have either '13' or '13.0'.

### 1.2.2 Clean data

Now, when we discovered problems in our data, let's clean (format) it and after convert it from XML to CSV format. For that purpose we need to execute **data.py** script.

What this script do:

- map shortcuts to the full street names (example before: '5th Ave' after: '5th Avenue')
- map shortcuts to the full direction names (example before: 'W 14th St' after: 'West 14th St')
- change outside NYC postcodes to '10001' (example before: '07302' after: '10001')
- convert all non-numeric house numbers to numeric: delete unnecessary characters, words and symbols (example before: '730A', '53-20', '525 REAR', '408 1/2' after: '730', '53', '525', '408')
- make all height values like float type (example before: '13' after: '13.0').

```
In [8]: Image(filename='shape1.png')
```

Out[8]:

```
# find all problem street types and change their values to correct from the mapping dictionary
def update_streets(name, mapping):
    m = street_type_re.search(name)
    if m:
        street_type = m.group()
        if street_type in mapping_street.keys():
            name = re.sub(street_type, mapping_street[street_type], name)
    return name

# find all problem direction types and change their values to correct from the mapping dictionary
def update_directions(name, mapping):
    m = direction_type_re.search(name)
    if m:
        direction_type = m.group()
        if direction_type in mapping_direction.keys():
            name = re.sub(direction_type, mapping_direction[direction_type], name)
    return name
```

In [10]: Image(filename='shape2.png')

Out[10]:

```
def shape_element(element, node_attr_fields=NODE_FIELDS, way_attr_fields=WAY_FIELDS,
                  problem_chars=PROBLEMCHARS, default_tag_type='regular'):
    """Clean and shape node or way XML element to Python dict"""

    node_attribs = {}
    way_attribs = {}
    way_nodes = []
    tags = []

    for tag in element.iter('tag'):
        key = tag.attrib['k']
        value = tag.attrib['v']

        # if a tag "k" value contains problematic characters, it should be ignored
        if problem_chars.search(key):
            continue

        # cleaning street names
        if is_street_name(tag):
            # updating problem values with defined in mapping dictionary
            value = update_streets(value, mapping_street)

        # cleaning direction names
        if is_street_name(tag):
            value = update_directions(value, mapping_direction)

        # cleaning postcodes
        if is_postcode(tag) and (value < '10001' or value > '11102'):
            value = '10001'

        # cleaning house numbers
        if is_housenumber(tag):
            m = housenumber_type_re.search(value)
            if m:
                if re.search(r"\s", value):
                    value = value.split(" ")[0]
                    if re.search(r"\;", value):
                        value = value.split(";")[0]
                elif re.search(r"\-", value):
                    value = value.split("-")[0]
                elif re.search(r"[^a-zA-Z]", value):
                    value = value[:-1]

        # cleaning height values
        if is_height(tag):
            m = height_type_re.search(value)
            if m:
                value = ''.join((value,'.0'))
```

### Import data to SQL database

Now let's import the cleaned .csv files into a SQL database using schema that is described in schema.py file. To do that we shall execute **create_db.py** file.

In [11]: Image(filename='db1.png')

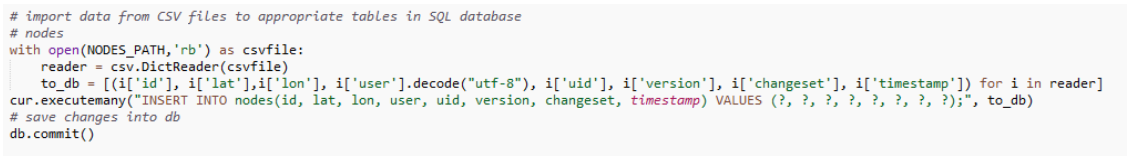
Out[11]:

```
# connect to database
db = sqlite3.connect(filename)
cur = db.cursor()


# create a nodes table
cur.execute("""CREATE TABLE IF NOT EXISTS nodes(
                                    id INTEGER primary key,
                                    lat REAL,
                                    lon REAL,
                                    user INTEGER,
                                    uid TEXT,
                                    version TEXT,
                                    changeset INTEGER,
                                    timestamp TEXT
                                    );""")
```

In [12]: `Image(filename='db2.png')`

Out[12]:




Above you see example of creating *nodes* table and importing data from nodes.csv file to this table. More details about other tables you can find in create_db.py file.

### 1.3 Data Overview

**File sizes** sampleNYC.osm 5.53 MB
nodes.csv 2 MB
nodes_tags.csv 131 KB
ways.csv 262 KB
ways_nodes.csv 750 KB
ways_tags.csv 631 KB
OpenStreetMap_NYC.db 3.4 MB

In [ ]:
```
# Number of nodes

SELECT COUNT(*) AS nodes_number
FROM nodes;
```

```
result: 21974

# Number of ways

SELECT COUNT(*) AS ways_number
FROM ways;

result: 3985

# Number of unique users

SELECT COUNT(DISTINCT(users.uid))
FROM (SELECT uid
      FROM nodes
      UNION ALL
      SELECT uid
      FROM ways) users;

result: 429

# Top 10 contributing users

SELECT posts.user, COUNT(*) as posts_number
FROM (SELECT user
      FROM nodes
      UNION ALL
      SELECT user
      FROM ways) posts
GROUP BY posts.user
ORDER BY posts_number DESC
LIMIT 10;

result:

smlevine|3714

lxbarth_nycbuildings|1797

ediyes_nycbuildings|1520

celosia_nycbuildings|1411

robgeb|1303

minewman|1070

ingalls_nycbuildings|536
```

```
ingalls|356

Korzun|269

# Number of users who have more than 3 posts

SELECT COUNT(*) AS users_number
FROM (SELECT posts.user, COUNT(*) as posts_number
      FROM (SELECT user
            FROM nodes
            UNION ALL
            SELECT user
            FROM ways) posts
      GROUP BY posts.user
      HAVING posts_number >= 3) users_posts;

result: 182

# Users with the most number of nodes and ways

SELECT user, COUNT(*) AS nodes_number
FROM nodes
GROUP BY user
ORDER BY nodes_number DESC
LIMIT 1;

result: Rub21_nycbuildings|8251


SELECT user, COUNT(*) AS ways_number
FROM ways
GROUP BY user
ORDER BY ways_number DESC
LIMIT 1;

result: Rub21_nycbuildings|1308
```

### 1.4 Additional Ideas

The main factor of inconsistent data that it is collected from different resources and specifically by human input. So I have a couple ideas for improvements in this direction:

1.First solution would be to develop and implement a validation system so that when user inserts an invalid value then he (she) will receive detailed message about format for this concrete field.

Benefits:

- it wouldn't be much difficult to implement

- it would be a new standard
- time for cleaning data will decrease

Anticipated Issues:

- because of standartization (data validation before insertion) some useful data can be lost - if, for example user can't add additional information about place because of restrictions
- some people can loose interest to this project because of these rules

2.Also we can look at some standards from other open-source maps projects, for example Google Maps API, and clean this data set using them.

Benefits:

- Google Maps has more standardized and relevant data than OpenStreetMap because Google Maps has special APIs and user interfaces for inserting map data, and most part of world population uses it

Anticipated Issues:

- time for cleaning data will increase because it takes additional time to investigate in what database: either OpenStreetMap or Google Maps there is correct format
- not all fields can be found in Google Maps
- if OpenStreetMap will be using Google Maps API then it needs more time and sources for integration with it (development process)

### 1.5 Additional Exploration

```
In [ ]: # Top 5 highway types

        SELECT value, COUNT(*) AS number_of_highways
        FROM nodes_tags
        WHERE key = 'highway'
        GROUP BY value
        ORDER BY number_of_highways DESC
        LIMIT 5;

        result:

        traffic_signals|49

        crossing|37

        bus_stop|3

        stop|3

        motorway_junction|2

        # What street has the most number of buildings in our dataset
```

```
SELECT value
FROM nodes_tags
WHERE key = 'street'
GROUP BY value
ORDER BY COUNT(*) DESC
LIMIT 1;

result: 1st Avenue

# What road type is most popular: one-way or two-ways?

SELECT value, COUNT(*) AS roads_number
FROM ways_tags
WHERE key = 'oneway'
GROUP BY value
ORDER BY roads_number DESC;

result:

yes|145

no|22
```

### 1.6 Conclusion

NYC area has a lot of data in the OpenStreetMap. I analyzed a small sample of it and it seems that the more data is in the collection the more difficult to clean and analyze it. There are a lot of contributors for this area but this data still has a lot of problems: empty values, differenet formats for streets, house numbers, height of building, etc. Also data for NYC is very rich, there are a lot of properties for buildings, streets, highways and so on. I couldn't clean this data set as full because practically every key has problem value, and to fix all values for one key we need to create a lot of rules .

But I am very excited about this project because it touches every step of working with data: import from XML, audit and cleaning using Python, export to CSV or JSON and after to SQL or MongoDB, analysis. In my opinion, it is a very strong practice for future Data Analyst.