OSM-Report

September 14, 2017

1 Data Wrangling: OpenStreetMap Data

1.1 Project Overview

To choose an area of the world in https://www.openstreetmap.org and use data munging techniques, such as assessing the quality of the data for validity, accuracy, completeness, consistency and uniformity, to clean the OpenStreetMap data for a part of the world that you care about. Finally, use SQL as the data schema to complete your project.

1.2 OSM Dataset

The area selected on the OpenStreetMap for this project is the city of Houston, Texas. Houston is the city where I currently reside. I moved here only about a year ago. So, I wanted to use this as an opportunity to learn more data based facts about this city and get more familiarized with its geography and general demographics. Moreover, I would like to contribute to its improvement on OpenStreetMap.

The data was directly exported as an OSM XML file from the Mapzen Metro extracts: https://mapzen.com/data/metro-extracts/metro/houston_texas/

1.3 Data Audit

Before starting to work with the entire dataset, an initial investigation of the dataset was conducted on a small sample of the dataset, which revealed problems with the data listed below.

1.3.1 Probelms with the data

1. Inconsistency in Abbreviations of street names(FM, Fm, Farm-to-Martket, Farm to Market)
2. Misspelled street names ('Plaze' instead of 'Plaza', 'Westhiemer' spelled as 'Westhimer')
3. Incorrect Postal codes ('Weslayan Street' listed in post code field)
4. Inconsistent Postal codes formats('77042-9998', 'tx 77042', '77478-', '770764', '773867386')

Street Names Although there were no major flaws with the street names, we came across many inconsistent abbreviations. For example Dr. dr, DR for Drive, Fwy or Frwy for Freeway etc. The misspelled words ('Plaze' instead of 'Plaza', 'Westhiemer' spelled as 'Westhimer') in street names were also identified and fixed using the mapping dictionary. There were also abbreviations N, E, S and W for North, East, South and West respectively which needed to be fixed. However, there were entries for 'Avenue E'and 'Avenue S' which needed to remain as is. Additionally, there were street names where multiple abbreviations needed to be fixed such as Hwy 6 S. So, instead of using regular expressions, I opted to iterate over each word of the street name correcting them to their respective mappings in audit.py using the following function:

```
def update_name(name, mapping):
    parts = name.split()
    for i in range(len(parts)):
        if parts[i] in mapping.keys():
            if parts[0] != "Avenue":
                parts[i] = mapping[parts[i]]
                name = " ".join(parts)
    return name
```

This updated all abbreviations in the street names, example Hwy 6 S => Highway 6 South except where the street was an Avenue.

Postcodes In the dataset, our initial review revealed that there were many different formats were used for postcodes. In order to bring the consistency in the format, I chose to convert all the postcodes to 5 digit format eg. 77077 using the below function:

```
#Compiling the reg ex to get 5 digit format from bad postcode format
#find 5 digit pattern in (5 digits) hyphen 4 digits
d5_d4 = re.compile(r'^(7\d{4})-\d{4}$')

#find 5 digit pattern in two alphabets space (5 digit)
str_d5 = re.compile(r'^[a-zA-Z]{2}\s(\d{5})$')

#(5 digits) followed by more numbers or alphabets in continuation
d5_chr  = re.compile(r'^(\d{5}).+$')


def update_postcode(postcode):
    if re.match(d5_d4, postcode):
        correct_postcode = re.findall(d5_d4,postcode)[0]
    elif re.match(str_d5, postcode):
        correct_postcode = re.findall(str_d5,postcode)[0]
    elif re.match(d5_chr, postcode):
        correct_postcode = re.findall(d5_chr, postcode)[0]
    else:
        return postcode
    return correct_postcode
```

After auditing was complete, the data was prepared to be inserted into a SQL database. To do so, the elements in the OSM XML file were parsed, transforming them from document format to tabular format, thus making it possible to write to .csv files. These csv files were then imported to a SQL database as tables using database.py.

When the complete database was queried to look at the postcodes, all postcodes were in the 5 digit format except the three postcodes listed below. 1. postcode =′"Weslayan Street′ 2. postcode = ′88581′ 3. postcode = ′7-′

1.3.2 postcode = 'Weslayan Street'

In order to investigate the postcode = 'Weslayan Street', we queried the database to fetch the complete details from the nodes_tags table for this postcode.

```
SELECT * FROM nodes_tags
WHERE id IN
(SELECT DISTINCT(id) FROM nodes_tags WHERE key='postcode' AND value='Weslayan Street');
```

The results were as below

```
[(u'1812187787', u'atm', u'yes', u'regular'),
 (u'1812187787', u'name', u'Chase', u'regular'),
 (u'1812187787', u'amenity', u'bank', u'regular'),
 (u'1812187787', u'street', u'77027', u'addr'),
 (u'1812187787', u'postcode', u'Weslayan Street', u'addr'),
 (u'1812187787', u'housenumber', u'2900', u'addr')]
```

There must have been a typo error as postcode is in place of street and street name in place of postcode. Since this is just one off query where postcode and street name are switched, I decided to fix it through sql query as below.

```
UPDATE nodes_tags
    SET value = CASE
    WHEN key = 'street'
        THEN 'Weslayan Street'
    WHEN key = 'postcode'
        THEN '77027'
    ELSE value
    END
WHERE  id = '1812187787';
```

1.3.3 postcode = '88581'

The postcodes "88581" stood out as clearly erroneous. This is becuase all of Houston area's postcode begin with '7' as opposed to this one. So, I queried the database to find the complete details from nodes and nodes_tags. Using the latitude and longitude coordinates, we confirmed that the address is in La Porte, which is in Houston Metro area. So, the postcode is definitely incorrect. I queried the database to get the list of cities and the number of tags to make sure if La Porte is indeed in our dataset.

```
SELECT tags.value, COUNT(*) as count
FROM (SELECT * FROM nodes_tags UNION ALL
      SELECT * FROM ways_tags) tags
WHERE tags.key LIKE '%city'
GROUP BY tags.value
ORDER BY count DESC;

SELECT * FROM nodes
WHERE id IN (SELECT DISTINCT(id) FROM nodes_tags WHERE key='postcode' AND value='88581');
```

Result:

```
[(u'1924194015',
  u'29.6646435',
  u'-95.1005174',
  u'Mars Is Waiting',
  u'901643',
  u'1',
  u'13188778',
  u'2012-09-20T21:09:56Z')]
```

Also, I queried the nodes_tags table to get the address details for this ID we got from above.

```
SELECT * FROM nodes_tags WHERE id='1924194015';
```

```
[(u'1924194015', u'name', u'Supercuts', u'regular'),
 (u'1924194015', u'shop', u'hairdresser', u'regular'),
 (u'1924194015', u'website', u'www.supercuts.com', u'regular'),
 (u'1924194015', u'street', u'Spencer Highway', u'addr'),
 (u'1924194015', u'postcode', u'77571', u'addr'),
 (u'1924194015', u'housenumber', u'9001', u'addr')]
```

A quick google search on Supercuts, 9001 Spencer highway revealed that the postcode should be 77571 instead of 88581. So, nodes_tags table was updated to reflect the same.

```
UPDATE nodes_tags
    SET value = '77571'
    WHERE id = '1924194015' AND  key = 'postcode' AND value = '88581';
```

1.3.4 postcode = ′7-′

A similar query as above was done to fetch address details for this postcode. The address was of a mexican restaurant ′Berryhill Tacos Sugarland, 13703 Southwest Freeway. This address was used to find the correct postcode i.e. 77478 and then updated in the database using below query.

```
UPDATE nodes_tags
    SET value = '77478'
    WHERE id = '4265057550' AND  key = 'postcode' AND value = '7-';
```

1.4 Data Overview And Additional Exploration

Before we start the analysis part, lets look at the statistics of our database and its files.

```
houston_texas.osm: 664 MB
OSM_FINAL_DB.db: 580 MB
nodes.csv: 246 MB
nodes_tags.csv:6 MB
ways.csv:21.3 MB
ways_tags.csv:67.1 MB
ways_nodes.csv:86.9 MB
```

```
In [34]: import sqlite3

         db = sqlite3.connect("OSM_FINAL_DB.db")
         cur = db.cursor()
```

1.4.1 Number of Nodes

```
In [17]: cur.execute("SELECT COUNT(*) FROM nodes;").fetchall()

Out[17]: [(3064419,)]
```

1.4.2 Number of Ways

```
In [18]: cur.execute("SELECT COUNT(*) FROM ways;").fetchall()

Out[18]: [(371640,)]
```

1.4.3 Number of unique users

```
In [19]: cur.execute("DROP TABLE IF EXISTS users;")

         cur.execute(""" CREATE TABLE users AS
                         SELECT all_uid.user AS user_name, all_uid.uid AS user_id
                         FROM
                         (SELECT user, uid FROM nodes
                         UNION ALL
                         SELECT user, uid FROM ways) all_uid; """)

         cur.execute(""" SELECT COUNT(DISTINCT(user_id)) FROM users LIMIT 10; """).fetchall()

Out[19]: [(1707,)]
```

1.4.4 Top 10 users

Tabular display in sql isn't very beautiful, so I used pandas to get a better formatted table.

```
In [20]: import pandas as pd

         top_10_df = pd.read_sql_query("""
                         SELECT user_name, user_id, COUNT(user_id) AS contributions
                         FROM users
                         GROUP BY user_id
                         ORDER BY contributions DESC
                         LIMIT 10; """, db)
         top_10_df

Out[20]:          user_name  user_id  contributions
         0  woodpeck_fixbot   147510         565052
         1         TexasNHD   672878         538407
```

```
2         afdreher  1110270         486780
3          scottyc   496606         204136
4          cammace  3119079         193856
5      claysmalley   119881         138030
6        brianboru     9065         116780
7          skquinn   243003          86092
8     RoadGeek_MD99   475877          81261
9          Memoire  2176227          56464
```

1.4.5 Number of users who contributed only once

```
In [21]: cur.execute("""
                     SELECT COUNT(*)
                     FROM
                         (SELECT user_id, COUNT(user_id) as num
                         FROM users
                         GROUP BY user_id
                         HAVING num = 1) one_time_users; """).fetchall()
```

```
Out[21]: [(372,)]
```

1.4.6 Amenities in Houston

```
In [22]: amenities_df = pd.read_sql_query("""SELECT value, COUNT(*) as num
                                          FROM nodes_tags
                                          WHERE key='amenity'
                                          GROUP BY value
                                          ORDER BY num DESC
                                          LIMIT 10;""", db)
         amenities_df
```

```
Out[22]:             value   num
         0  place_of_worship  2220
         1            school   823
         2          fountain   713
         3        restaurant   703
         4         fast_food   641
         5      fire_station   351
         6              fuel   279
         7          pharmacy   177
         8              bank   173
         9            police   161
```

Place of worship, schools and Fountains(Surprise!) takes the top places in above table. Lets look at which religion is practiced the most.

1.4.7 Religion

```
In [23]: cur.execute(""" SELECT nodes_tags.value, COUNT(*) as num
                         FROM nodes_tags
                         JOIN
                         (SELECT DISTINCT(id) FROM nodes_tags WHERE value='place_of_worship') i
                         ON nodes_tags.id=i.id
                         WHERE nodes_tags.key='religion'
                         GROUP BY nodes_tags.value
                         ORDER BY num DESC;""").fetchall()
```

```
Out[23]: [(u'christian', 2151),
          (u'buddhist', 16),
          (u'jewish', 12),
          (u'muslim', 9),
          (u'unitarian_universalist', 5)]
```

Christianity! No Surprise there!

1.4.8 Restaurants - Most popular cuisine

```
In [27]: cur.execute("""
                         SELECT nodes_tags.value, COUNT(*) as num
                         FROM nodes_tags
                         JOIN (SELECT DISTINCT(id) FROM nodes_tags WHERE value='restaurant') i
                         ON nodes_tags.id=i.id
                         WHERE nodes_tags.key='cuisine'
                         GROUP BY nodes_tags.value
                         ORDER BY num DESC
                         LIMIT 10;""").fetchall()
```

```
Out[27]: [(u'mexican', 74),
          (u'american', 36),
          (u'pizza', 33),
          (u'italian', 32),
          (u'chinese', 28),
          (u'seafood', 22),
          (u'burger', 19),
          (u'barbecue', 16),
          (u'sandwich', 13),
          (u'thai', 10)]
```

That was predictable. Let us also look at how many Starbucks are there in Houston.

```
In [28]: cur.execute(""" SELECT COUNT(*) as num
                         FROM nodes_tags WHERE value='Starbucks';""").fetchall()
```

```
Out[28]: [(64,)]
```

So, there are 64 Starbucks in Houston Area.

1.4.9 Schools

```
In [29]: cur.execute("""SELECT COUNT(*) AS num
         FROM nodes_tags
         WHERE nodes_tags.value = 'school';""").fetchall()

Out[29]: [(831,)]
```

There are 831 schools in Houston Area.

1.4.10 Source of Data

In order to get the sources of data, I first created a table of all values for which key is source and their count from both nodes_tags as well as ways_tags. And then query the table to get the number of records for different sources.

```
In [30]: cur.execute("DROP TABLE IF EXISTS allsources;")

         cur.execute(""" CREATE TABLE allsources AS
                         SELECT value, count(*) AS num
                         FROM
                         (SELECT value
                         FROM nodes_tags
                         WHERE nodes_tags.key = 'source'
                         GROUP BY value

                         UNION ALL

                         SELECT value
                         FROM ways_tags
                         WHERE ways_tags.key = 'source'
                         GROUP BY value) source
                         GROUP BY source.value
                         ORDER BY num DESC""")

Out[30]: <sqlite3.Cursor at 0x9563c00>
```

Since the data was not initially cleaned for source tags, so I used %bing% or 'Bing%, etc. to find its matches in the tag fields.

```
In [33]: cur.execute(""" SELECT SUM(num) AS bing
                         FROM
                         (SELECT * FROM allsources
                         WHERE value LIKE 'Bing%' OR value LIKE '%bing%') b""").fetchall()

Out[33]: [(18,)]
```

I queried separately for different sources and numbers are shown below. `Bing - 18 Tiger - 10 Wikipedia - 2 GPS - 7 Yahoo - 7 Google - 1`

1.5 Additional Ideas

1.5.1 Postcode validation

While auditing the file, we came across so many different formats for postcodes being used. Not just the formats of postcode, but we also obseved there was mismatch of city and postcode. In my opinion the city and the postcode should be cross checked for validation at the point of entry by the user. There are many public APIs to retrieve addresses from postcode. If implemented, this will prevent a huge amount of incorrect data inputs and will bring consistency in the format too. In turn, it will help save a lot of time and effort in data cleaning and it will be lot less inconvinience for the people who use the OpenStreet Map.

Another viable solution could be to use location data from mobiles of the users who are logged in. If the location is already in the file, the user will be asked to verify the location details of the place or else input the address details. It is also possible that some users may not want to share their location data, so this can be used as an opt in option for the users. In effect, it may also increase the number of contributions from not so regular contributors if given an opportunity for a quick add/validate option.

1.6 References

1. https://wiki.openstreetmap.org/wiki/Main_Page
2. https://mapzen.com/data/metro-extracts/
3. https://www.dataquest.io/blog/python-pandas-databases/
4. https://discussions.udacity.com/t/creating-db-file-from-csv-files-with-non-ascii-unicode-characters/174958/7