OpenStreetMap Data Wrangling Project

Chosen Area – Biloxi, Mississippi, USA

I have lived next to Biloxi for most of my life in a city called Ocean Springs. I recently began working at a new job in Biloxi and chose this area for my project to see if anything interesting would be revealed about where I am now working.

Problems with Data

My dataset was relatively small (~150MB) so my home computer could handle the processing easily. Because of this, I decided to jump straight into the whole file without working with a smaller sample. Surprisingly, the data I encountered was mostly clean in the areas in which I was interested. However, there were a few outliers that I decided I would clean programmatically to gain practice for when I encountered some very dirty data down the road. I audited and cleaned the following areas:

- ***Street Types:*** Addresses contained differing abbreviations that represented the same street type (E.g. “Rd”, “Rd.”, and “Road”).

- ***Zip Codes:*** There was one outlier zip code that contained additional shipping codes attached to the 5-digit zip code.

- ***City Names:*** There were two spellings of a commonly misspelled city, and an area that is often mistaken as a city.

- ***Cuisine:*** There were differing levels of specificity for places to eat, which would make querying harder. Similar cuisines were grouped into lower detailed categories.

Street Types

When looking over the street names in the dataset, I noticed the abbreviations for street types were not uniform in a few places. To correct this, I used a regular expression and a dictionary with all variations of abbreviations as the keys and the proper full name as the values. The data was programmatically updated to the full road names using the following function:

```
def update_name_street(name, streetmapping):
    street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)
    m = street_type_re.search(name)
    street_type = m.group()

    if street_type in streetmapping.keys():
            name = re.sub(m.group(), streetmapping[m.group()],name)

    return name
```

Zip Codes

I then checked the zip codes to see how many were contained in the map area. I saw that there were 12 unique zip codes but one of them was duplicated with an additional shipping code tacked on the end of the 5-digit zip. I decided I would write the following function to clean it rather than just change the single instance manually:

```
def update_zips(name):
    if len(name) > 5: #take only first 5 numbers of zip
        print name, 'updated to:'

        if re.search('[0-9]{5}', name):
            updated_name = re.findall('[0-9]{5}', name)
            name = updated_name[0]
            print name
            return name

    else:
        return name
```

City Names

When checking the city names, there was a misspelling of the city “D’Iberville” without the apostrophe and uppercase “I”. Also, “Keesler Air Force Base” was listed as a city when it is not, regardless of those on base who would beg to differ. I wrote the following function to update the city names using a dictionary that mapped the proper changes:

```
def update_name_city(name, citymapping):
    if name in citymapping:
        print name, "updated to:"
        name = citymapping[name]
        print name
        return name

    else:
        return name
```

Cuisine

The cuisine tag got a little tricky and I had to make a change that resulted in a loss of detail in the data. A few values were a simple change to spelling “america” as “american” or changing “chicken:american” to simply “american”. However, the hard decision came when deciding how to treat the differing levels of detail between Asian cuisines. There were listings for; Asian, Vietnamese, Chinese, and Japanese. To better show how popular Asian cuisines are when I queried the table later, I decided to group all four listings under “Asian”. I did this with the following function:

```
def update_name_cuisine(name, cuisinemapping):
    if name in cuisinemapping:
        print name, 'updated to:'
        name = cuisinemapping[name]
        print name
        return name

    else:
        return name
```

Biloxi Database Review

File Sizes

```
Biloxi .......................................... 153.57 MB
Biloxi_Sample.osm ............................... 3.11 MB
DAND OSM Data Cleaning.ipynb .................... 0.29 MB
DAND OSM Data Cleaning.py ....................... 0.03 MB
DAND OSM Data Wrangling Project.docx ............ 0.23 MB
nodes.csv ....................................... 58.96 MB
nodes_tags.csv .................................. 0.17 MB
OSM Link.txt .................................... 0.00 MB
OSMData.db ...................................... 78.70 MB
ways.csv ........................................ 4.96 MB
ways_nodes.csv .................................. 19.90 MB
ways_tags.csv ................................... 10.39 MB
~$ND OSM Data Wrangling Project.docx ............ 0.00 MB
~WRL2376.tmp .................................... 0.10 MB
```

Number of Nodes

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

736,587

Number of Ways

```
SELECT COUNT(*) FROM ways;
```

86,674

Number of Unique Contributors to Dataset

```
SELECT Count(DISTINCT both.user)
FROM (SELECT user FROM nodes
UNION ALL SELECT user FROM ways) as both;
```

233

Top 5 City Tags

```
SELECT both.value, COUNT(*) as Total
FROM (SELECT * FROM nodes_tags UNION ALL
SELECT * FROM ways_tags) as both
WHERE both.key == 'city'
GROUP BY both.value
ORDER BY Total DESC
LIMIT 5;
```

```
               0      1
0       Gulfport  14982
1         Biloxi  11952
2     D'Iberville   2480
3  Ocean Springs     30
```

Only 4 cities were contained in dataset

Top 5 Zip Code Tags

```
SELECT both.value, COUNT(*) as Total
FROM (SELECT * FROM nodes_tags UNION ALL
SELECT * FROM ways_tags) as both
WHERE both.key == 'postcode'
GROUP BY both.value
ORDER BY Total DESC
LIMIT 5;
```

```
       0     1
0  39532  8153
1  39503  7420
2  39507  6057
3  39501  5878
4  39531  3440
```

Top 5 Cuisine Tags

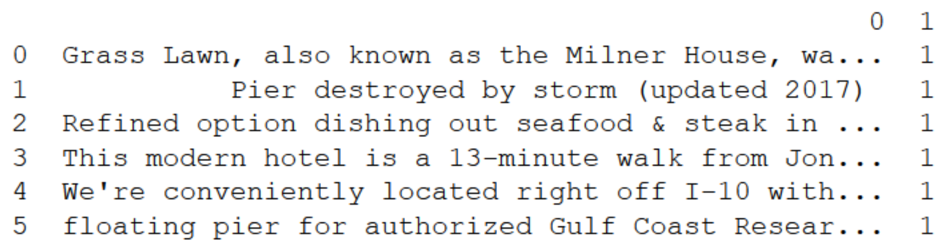
```
SELECT both.value, COUNT(*) as Total
FROM (SELECT * FROM nodes_tags UNION ALL
SELECT * FROM ways_tags)
as both
WHERE both.key == 'cuisine'
GROUP BY both.value
ORDER BY Total DESC
LIMIT 5;
```

```
          0   1
0  american  10
1    burger  10
2     pizza   9
3  sandwich   9
4     asian   7
```

Improvements

My biggest issue with the data was that it showed me what was in the area, but it did not thoroughly explain what something was. I stumbled upon a tag named “description” and to my surprise, there were only 6 descriptions of a node in the entire dataset:

```
SELECT both.value, COUNT(*) as Total
FROM (SELECT * FROM nodes_tags UNION ALL
SELECT * FROM ways_tags)
as both
WHERE both.key == 'description'
GROUP BY both.value
ORDER BY Total DESC;
```



```
                                                    0  1
0  Grass Lawn, also known as the Milner House, wa...  1
1           Pier destroyed by storm (updated 2017)   1
2  Refined option dishing out seafood & steak in ...  1
3  This modern hotel is a 13-minute walk from Jon...  1
4  We're conveniently located right off I-10 with...  1
5  floating pier for authorized Gulf Coast Resear...  1
```

It would have been very useful for someone unfamiliar with the area to have a description of what each node was. However, when I thought more about the dataset, I realized that there may be issues with having a robust “descriptions” tag. First, this dataset is open for anyone to contribute to. It would be very easy for a disgruntled employee to leave a horrible description of their place of work, or for an unhappy customer to leave a rotten review of a restaurant. Second, since having an open sourced description tag could bring out the worst in people, it would require heavy monitoring for accuracy and a policy may have to be written for acceptable use.

Conclusion

Overall, the Biloxi, MS data is a document containing a large amount of useful information. However, it is inherently prone to invalid entries, inconsistency, and non-uniformity since it is open for anyone to contribute. The few keys I audited should be fine to query now, but there are hundreds of unique keys that were not audited and are sure to contain a variety of errors. A comprehensive audit of every key would be incredibly time intensive, but possible if a similar pattern to my programmatic auditing and cleaning was followed.