OpenStreetMap Data Wrangling Study: Tampa, FL

1. Map Area

Tampa, FLORIDA, USA.

- https://mapzen.com/data/metro-extracts/metro/tampa_florida/ (https://mapzen.com/data/metro-extracts/metro/tampa_florida/)

I have lived in Florida for years and studied my Master and PhD degree there. I was lost in Tampa a few times so I would like to contribute to its improvement on OpenStreetMap.org.

2. Problems Encountered in the Map

Since the size of the datafile is more than 300 MB, I defined *k*= 40 instead of 10 for sampling. The output sample data file is ready now.

First, I have explored small sample datsets using k = 40 and k = 30 to investigate potential problems and solution strategies. The main probmlems were:

2.1. Gnis data:

The data imported from GNIS database seems to have a different format. I investigated the issue as below with a helper function:

In [ ]:

```
def investigate_specific_key(filename, word, tag):
    tag_list = []
    for number, element in ET.iterparse(filename):
        if element.tag == tag:
            for item in element:
                    item_dict = {}
                    key = item.attrib['k']
                    if key.startswith(word):
                        item_dict[key] = item.attrib['v']
                        tag_list.append(item_dict)
    return tag_list

investigate_specific_key(SAMPLE_FILE, 'gnis', 'node')
```

- *Gnis Data problemmatic issues:*

  1 Second level “k” tags with the value "type"

  k="gnis:created" v="10/19/1979"

  k="gnis:state_id" v="12"

  k="gnis:county_id" v="057"

  k="gnis:feature_id" v="293110"

  The 'gnis' part should be assigned a new style as below:

  type = 'gnis'

  2) 'feature_id' vs 'id'

  From openstreetmap wiki page (https://wiki.openstreetmap.org/wiki/USGS_GNIS), it is seen that 'id' and 'feature_id' refers feature-id. Therefore, I should convert ids to feature_id.

  "The Feature ID uniquely identifies a feature in the GNIS database and is thus the most important thing to tag when relating an OSM feature to a GNIS feature. The tag gnis:feature_id is by far the most commonly used for this purpose. Other tags for GNIS Feature IDs include gnis:id and tiger:PLACENS."

  3) 'ST_num' and 'state_id'

  Both state_id and ST_num refers to state_id, which is 12. Therefore, I should convert ST_num to state_id.

  4) 'county_name and 'County'

  Both 'county_name' and 'County' refers to county_name. Therefore, I should convert County to county_name.

  5) 'County_num and county_id'

  Both 'County_num' and 'county_id' refers to county_id. Therefore, I should convert County_num to county_id.

2.2. Street Names:

I have checked the abbreviations of the streets and made them the same format. First, I have audited them. I have used the same investigate_specific_key helper function to check the street names.

- *Street Names problemmatic issues:*

---

1) Overabbreviated Street Names

Below are some examples to the mentioned street types. All the strrets names should be more readable.

{'addr:street': '4th St N'} : '4th Street North'

{'addr:street': 'S Belcher Rd'} : 'South Belcher Road'

2) Unstandardized Abbreviations

Some street names are not over abbreviated but have unstandardized names. Those should be standardized for further analyses.

'Boulevard' vs 'Blvd', 'South' vs 'W' (for west) and 'Avenue' vs 'Ave'

{'addr:street': 'Gulf to Bay Boulevard'} vs {'addr:street': 'Park Blvd'}

{'addr:street': 'South Macdill Avenue'} vs {'addr:street': 'W Waters Ave'}

2.3. Postal Codes:

I have checked the postal codes for potential problems. Because, postal codes are critical and is a primary id for most analyses.

The only problemmatic issue is unstandardized type of postcodes as below:

{'addr:postcode': '33781-5034'}

I can either ignore the after-dash part or create a new key. In the sample, there exist only one this type. So, ignoring such types and deleting the after-dash part would be a beter idea.

2.3. Housenumbers:

There exist suite numbers and they should be excluded and put into the unit part. An example is below:

{'addr:housenumber': '2663 Suite 27'}

2.4. Amenities:

In amenity there exist no big problem. However, some amenities are also given as keys. Examples are below:

'bicycle_parking'

Those will be investigated in the additional ideas part.

2.5. Tiger Data:

The data imported from tiger database seems to have a different format. I investigate the issue as below with a helper function:

- *Tiger Data problemmatic issues:*

  Second level “k” tags with the value "type"

  ```
      <tag k="name" v="West Charter Street" />
      <tag k="highway" v="residential" />
      <tag k="tiger:cfcc" v="A41" />
      <tag k="tiger:county" v="Hillsborough, FL" />
      <tag k="tiger:reviewed" v="no" />
      <tag k="tiger:zip_left" v="33602" />
      <tag k="tiger:name_base" v="Charter" />
      <tag k="tiger:name_type" v="St" />
      <tag k="tiger:zip_right" v="33602" />
      <tag k="tiger:name_direction_prefix" v="W" />
  ```

  Tiger part should be a new 'type' key and remaining part should be other keys. The format should be as below:

  ```
       type  = 'tiger'
  ```

  There seems no other problem in the data came from tiger database.

3. Cleaning and Updating Process

The below updates should be applied before preparing for database. I will write appropriate functions to clean and shape the sample data, test them and apply for the main dataset.

3.1. Gnis data updates

Second level key issues are investigated at the end. So, I dealt with

- converting ids to feature_id
- converting ST_num to state_id
- converting Country to country_name
- converting Country_num to country_id

I have created mappings and change_name function for this.

In [4]:

```
map_gnis = {'gnis:id': 'gnis:feature_id', 'gnis:ST_num': 'gnis:state_id',
            'gnis:County': 'gnis:county_name', 'gnis:County_num': 'gnis:county_i
d'}
```

In [2]:

```
def change_name(name, mapping):
    for key in mapping:
        if name == key:
            return mapping[key]
        else:
            return name
```

3.2. Postal Code Updates

I only took first 5 character of postal codes. A simple helper function was sufficient.

In [3]:

```
def postal_code_cleaner(postcode):
    return postcode[:5]
```

3.3. House Number Updates

I have cleared the extra part after the house_numbers. There exist exeptions that housenumbers may end with a letter

In [ ]:

```
def house_number_cleaner(housenumber):
    a = housenumber.find(' ')
    if a >=1 and len(housenumber[a:]) >1 :
        return housenumber[:a]
    else:
        return housenumber.strip()
```

3.4. Street Name Updates : First, I have audited street names to fill the mappings for abbreviations. The functions are adopted from the Case Study Part. After auditing, I have defined the below mapping and the below function. I did not use the regular expressions, rather I have used the function from sample case to make character by character analysis.

In [5]:

```
street_mapping = { "St": "Street",
            "St.": "Street",
            "Ave": "Avenue",
            "Rd.": "Road",
            "Rd" : "Road",
            "Blvd": "Boulevard",
            "Blvd.": "Boulevard",
            "BLVD": "Boulevard",
            "Cir": "Circle",
            "Dr" : "Drive",
            "Hwy": "Highway",
            "Pl": "Plaza",
            "S": "South",
            "N": "North",
            "N.": "North",
            "E" : "East",
            "S.": "South",
            "E.": "East",
            "W": "West",
            "W.": "West",
            "SE": "Southeast",
            "NE": "Northeast"}
```

In [6]:

```
# Update street names without using regular expressions. Adopted from sample Sql
project.
def update(name, mapping):
    words = name.split()
    for w in range(len(words)):
        if words[w] in mapping:
            if words[w].lower() not in ['suite', 'ste.', 'ste']:
                # For example, don't update 'Suite E' to 'Suite East'
                words[w] = mapping[words[w]]
    name = " ".join(words)
    return name
```

3.5. Tiger Data Updates :

The updates (assigning the 'tiger:' value to the 'type' attribute) will be done when I prepare the data for the database)

4. Prepare the Data for DATABASE

I have combined all the cleaning functions into a single clean_all function to process the data only once. And, I have cleaned the data and processed adopting the shape element functions and prepared the data for the database. All the sample data is validated and then the full osm data was processed.

In [9]:

```
# Clean all the data combining all the functions
def clean_all(element):
    ''' Clean and update gnis keys, street names, postcodes, housenumbers using predefined functions'''
    if element.tag == 'way' or element.tag == 'node':
        for item in element:
            if item.tag == 'tag':
                key = item.attrib['k']
                if key.startswith('gnis'):
                    item.attrib['k'] = change_name(key, map_gnis)
                if key =='addr:postcode':
                    value = item.attrib['v']
                    item.attrib['v'] = postal_code_cleaner(value)
                if key =='addr:housenumber':
                    value = item.attrib['v']
                    item.attrib['v'] = house_number_cleaner(value)
                if key == 'addr:street':
                    value = item.attrib['v']
                    item.attrib['v'] = update(value, street_mapping)
```

5. Export the CSV Files to the Sqlite Database

I have used DB-API to export the CSV files to the database. I imported 5 csv files (nodes, nodes_tags, ways, ways_tags, ways_nodes) into the SQLite database. A sample code is as below:

In [ ]:

```
sqlite_file = 'osmfile.db'
conn = sqlite3.connect(sqlite_file)
cur = conn.cursor()

cur.execute('''DROP TABLE IF EXISTS nodes_tags''')
conn.commit()
```

In [ ]:

```
cur.execute('''CREATE TABLE nodes_tags(id INTEGER, key TEXT, value TEXT, type TE
XT, FOREIGN KEY (id) REFERENCES nodes(id))''')
conn.commit()
```

In [ ]:

```
with open('nodes_tags.csv','rb') as f:
    dr = csv.DictReader(f) # comma is default delimiter
    to_db = [(i['id'], i['key'],i['value'].decode("utf-8"), i['type']) for i in 
dr]
```

In [ ]:

```
cur.executemany("INSERT INTO nodes_tags(id, key, value,type) VALUES (?, ?, ?, ?)
;", to_db)
# commit the changes
conn.commit()
conn.close()
```

6. Overview of Data

In this section, I have overviewed all dataset using sql queries.

This section comntains

- Size of the files
- Number of unique users
- Number of nodes and ways
- Number of chosen type of nodes

6.1. File Sizes

- tampa_florida.osm: .....**373.1 MB** (The main osm file)
- sample_tampa.osm: .....**9.4 MB**
- tampaosm.db: .....**206.5 MB**
- nodes.csv : .....**137.2 MB**
- nodes_tags.csv: .....**6.7 MB**
- ways_nodes.csv: .....**46.2 MB**
- ways_tags.csv: .....**33.1 MB**
- ways.csv: .....**10.8 MB**

6.2. Number of Nodes and Ways

I have used below queries to explore the numbers as below:

In [ ]:

```
conn = sqlite3.connect(sqlite_file)
cur = conn.cursor()

# Count of nodes
cur.execute('''SELECT COUNT(*) FROM nodes''')
rows = cur.fetchall()
print(rows)
```

In [ ]:

```
# Count of ways
cur.execute(''' SELECT COUNT(*) FROM ways''')
rows = cur.fetchall()
print(rows)
```

The below results were obtained from the above queries:

- Number of rows: **1663833**
- Number of ways: **184110**

6.3. Number of Unique Users

The below queries were used to find unique users both in nodes and ways together.

In [ ]:

```
# Number of unique users
cur.execute('''SELECT COUNT(*) FROM (SELECT uid FROM nodes UNION SELECT uid FROM 
ways)''')
rows = cur.fetchall()
```

- Number of total unique users: **1452**

6.4. Number of Chosen Type of Nodes

Number and type of amenities:

In [ ]:

```
cur.execute('''SELECT value, COUNT(*) as num FROM nodes_tags WHERE key = 'amenit
y' GROUP BY value ORDER BY num DESC''')
rows = cur.fetchall()
```

The query showed that the top numbers were as below:

'restaurant': 852

'place_of_worship' : 771

'school' :554

'fast_food' : 396

'bicycle_parking' : 353

There were restaurants and place of worships mostly as amenities in Tampa, FL.

Number and type of keys in node_tags, way_tags:

```
In [ ]:
```

```
#Nodes tags
cur.execute('''SELECT key, COUNT(*) as num FROM nodes_tags GROUP BY key ORDER BY
num DESC''')
rows = cur.fetchall()
```

The query showed that the most common nodes_tags keys were as below:

'highway': 25586

'power', 18786

'name', 18197

'operator', 9607

'route_ref', 9157

```
In [ ]:
```

```
#Ways tags
cur.execute('''SELECT key, COUNT(*) as num FROM ways_tags GROUP BY key ORDER BY
num DESC''')
rows = cur.fetchall()
```

The query showed that the most common ways_tags keys were as below:

'highway', 117490

'name', 82528

u'county', 68990

u'cfcc', 68944

u'name_base', 64235

7. Other Ideas About the Dataset

A problem I have seen in the dataset is duplication in key categories. These duplications reduces the validity of search and analysis. Moreover, the decisions made based on those data would be biased. To illustrate, below 'bicycle_parking' is assigned as a **key**.

```
In [ ]:
```

```
cur.execute('''SELECT  key, COUNT(*) as num FROM nodes_tags WHERE key = 'bicycle
_parking' GROUP BY key  ''')
rows = cur.fetchall()
```

The query gives:

(u'bicycle_parking', 45)

There exist 45 bicycle_parking keys in the dataset. However, bicycle parking is also assigned as an **amenity**.

```
In [ ]:
```

```
cur.execute('''SELECT  key, value, COUNT(*) FROM nodes_tags WHERE value = "bicyc
le_parking" GROUP BY key''')
rows = cur.fetchall()
print rows
```

The query gives:

[(u'amenity', u'bicycle_parking', 353)]

There exists 353 bicycle parkings also as amenities.

We can write a script to find those duplications for further investigation.

In [ ]:

```
cur.execute('''SELECT DISTINCT(key) FROM nodes_tags''')
keys = cur.fetchall()

cur.execute('''SELECT DISTINCT(value) FROM nodes_tags WHERE key = "amenity" ''')
amenities = cur.fetchall()

duplications = []
for key in keys:
    if key in amenities:
        duplications.append(key)
```

The query and script gives **duplications** :

[(u'atm',),

(u'fuel',),

(u'social_facility',),

(u'shelter',),

(u'bench',),

(u'bicycle_parking',),

(u'waste_basket',),

(u'parking',),

(u'toilets',),

(u'car_wash',),

(u'studio',)]

As seen, there exist several duplications to be fixed in the dataset. To fix those, we can assign them as amenities. This will increase efficiency of the search and analysis in the dataset.

However, **a potential problem** would be to transfer the other field related with those duplications such as name and value.

8. References

I have only used Udacity Data Wrangling Case Study and Sample_Submission (https://gist.github.com/carlward/54ec1c91b62a5f911c42#file-sample_project-md) for this study. I have referred if I used those resources.

In [ ]:

```

```