Wrangle OpenStreetMap Data

August 8, 2017

1 Wrangle OpenStreetMap Data

For this map data wrangling project, I selected my home state of Rhode Island. It is about the size of some larger urban areas, so I figured it might pose an interesting choice! The map data I downloaded from Geofabrik, which already has a pre made Rhode Island openstreetmap data file. The area extends into the ocean to include Block Island, an offshore town in the state.

1.1 Data Investigation and Auditing

Several issues were identified in the data: * Inconsistent and abbreviated street names (Point Judith Rd, Cunningham Sq) * Incorrect or inconsistent zip codes (02835, 02903-2996, 029212) * Street names embedded in Tiger GPS blocks in ways

1.1.1 Cleaning Street Data

First, like the case study, I looked into street abbreviations and found that there were many nodes with inconsistencies. Fixing these abbreviations will help standardize the data and make for easier analysis of addresses if needed. The update_street_name code was used in the conversion to csv to fix the street names.

```
def update_street_name(name, mapping):
    """
    Takes in an abnormal street name and returns a corrected one
    """

    # Extract the street type and correct if possible
    street_type = re.search(street_type_re, name).group()
    if street_type in mapping:
        name = name.replace(street_type, mapping[street_type], 1)

    return name
```

This code would take something like Point Judith Rd and replace it with Point Judith Road.

1.1.2 Update Postcodes

Another important facet of the address is the post code - auditing this is not particularly difficult and can easily find some mistakes. Upon searching through the postal codes in the OSM database, it was clear that most had the standard five digit form. However, some included the final four digits after the hyphen and a few were the wrong length entirely, probably due to human input.

The function update_postcodes is used during csv conversion to check for inconsistent postal codes. It will omit records that are likely a result of typos, and only return the first five digits of a postcode found with the hyphen format.

```
def update_postcodes(code):
    """
    Takes a postcode and updates it to a valid 5 digit postcode
    """
    # If the postcode is the proper length, return it
    if len(code) == 5:
        return code
    # Otherwise, if the postcode has a dash only return the first five digits
    elif '-' in code:
        return code[:5]
    # If the postcode does not fall in to these catigories, return None
    else:
        return None
```

For example, the update_postcodes function would make the following changes:

```
02835      => 02835
02903-2996 => 02903
029212     => None, omit record
```

1.1.3 Update Tiger GPS Blocks

While doing a quick through the data, I noticed that there were a lot of ways with their names broken up into segments in a different format. They always had the key start with 'tiger:' which I assumed to mean that they came from an automated upload. Tiger GPS blocks typically have the following format:

```
<tag k="name" v="Commonwealth Avenue" />
<tag k="highway" v="residential" />
<tag k="tiger:cfcc" v="A41" />
<tag k="tiger:county" v="Bristol, RI" />
<tag k="tiger:reviewed" v="no" />
<tag k="tiger:zip_left" v="02806" />
<tag k="tiger:name_base" v="Commonwealth" />
<tag k="tiger:name_type" v="Ave" />
<tag k="tiger:zip_right" v="02806" />
```

You can see that this block has already been collated with a name at the top fo the block in the normal format. During the audit, I decided to check if there were any tiger GPS blocks that had not yet been updated with a name tag. It turns out there were several, and adding the name tag would be useful for standardizing the database. The update_tiger function is used during csv conversion to return the name element. The function also calls the earlier function, update_street_name, to check for abbreviated road types and replace them.

```python
def update_tiger(parent_id, tiger_name, tiger_type):
    """
    Function takes in a parent id, tiger name, and tiger type and returns
    a dictionary with the appropriate name dictionary. Should only be called
    when the tiger name has not yet been compiled.
    """

    # initialize a child dict with the appropriate type, key, and id
    child_dict = {'id':parent_id,
                  'type':'regular',
                  'key':'name'}

    # combine the tiger parts into a full street name
    street = tiger_name + ' ' + tiger_type

    # use the street audit function to fix any abbreviation issues
    street_final = audit_streetnames.update_street_name(street,
                                                        audit_streetnames.mapping)

    # print 'Updated', tiger_name, tiger_type,' => ', street_final
    child_dict['value'] = street_final
    return child_dict
```

For example, this function would take in the following block and add a name element.

```xml
<tag k="tiger:name_base" v="Commonwealth" />
<tag k="tiger:name_type" v="Ave" />
```

This would add the name element:

```xml
<tag k="name" v="Commonwealth Avenue" />
```

1.2 Database Analysis and Queries

We will start by listing the file sizes for some important files:

```
rhode-island-latest.osm   185 MB
ri_osm.db                 98.9 MB
nodes.csv                 71.2 MB
nodes_tags.csv            2.5 MB
ways.csv                  5.3 MB
ways_tags.csv             12.1 MB
ways_nodes.csv            22.8 MB
```

1.2.1 Initial Queries

Next we query to total number of nodes:

```sql
SELECT count(*) FROM nodes;
```

846158

The total number of ways:

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

89540

The number of distinct users, combining both the nodes and ways database:

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

803

The total number of shops in the database:

```
SELECT COUNT(*)
  FROM (SELECT key FROM nodes_tags
  UNION ALL
  SELECT key FROM ways_tags) combined
  WHERE key = "shop";
```

412

1.2.2 Additional Queries

Now we will compute some additional statistics to explore the database. First, we can look at the most common keys in node_tags to see what is available for exploration.

```
SELECT key, count(*) as num
FROM nodes_tags
GROUP BY key
ORDER BY num DESC;
```

```
source      7119
attribution 6462
name        4647
power       4466
ele         3441
feature_id  3116
amenity     2771
created     2564
county_id   2373
state_id    2373
highway     1947
natural     1752
```

The most noticable thing is that so many of the keys are for source and other internal documentation use. Of the remaining common keys, amenity and natural seem to be the most interesting to explore.

To start exploring amenities, we will look at the ten most common amenities:

```
SELECT nodes_tags.value, COUNT(*) as num
  FROM nodes_tags
  WHERE key = 'amenity'
  GROUP BY value
  ORDER BY num DESC
  LIMIT 10;
```

```
school           608
place_of_worship 508
grave_yard       410
restaurant       177
library          116
fire_station     112
parking           88
fast_food         77
bench             72
kindergarten      63
```

For fun, I decided to query the most common fast food resaurants in RI.

```
SELECT nodes_tags.value, count(*) as num
  FROM nodes_tags
  JOIN (SELECT DISTINCT(id)
        FROM nodes_tags
        WHERE value = 'fast_food') as ff
        ON nodes_tags.id = ff.id
  WHERE nodes_tags.key = 'name'
  GROUP BY nodes_tags.value
  ORDER BY num DESC
  LIMIT 4;
```

```
Subway          13
Dunkin' Donuts   9
McDonald's       4
Burger King      3
```

I'm very surprised that Dunkin is not the top. I think most other Rhode Islanders would be too!

Next I looked into the nature tags, and their frequency.

```
SELECT value, count(*) as num
  FROM nodes_tags
  WHERE key = 'natural'
  GROUP BY value
  ORDER BY num DESC;
```

```
tree   1249
peak    236
bay     128
```

```
beach    73
wetland  51
cliff    10
wood      3
cape      1
rock      1
```

Trees are by far the most common nature tags. I'm surpriesed to see a rock, let's look into what that is!

```
SELECT nodes_tags.value
  FROM nodes_tags
  JOIN (SELECT DISTINCT(id)
        FROM nodes_tags
        WHERE value = 'rock') as rock
        ON nodes_tags.id = rock.id
  WHERE nodes_tags.key = 'name';
```

This query returns **Elbow Rock**.

I'm not totally sure what Elbow Rock is, but a quick search seems to show that there is hiking trail near it.

1.3 Conclusions, Ideas, and Issues

This approach for analyzing OpenStreetMap data works well overall, allowing easy creation of the csv files and database. One possible improvement would be to split the users into a separate table with the primary key uid and the user values. It would make it easier to identify distinct users, and you could easily store more information about each user in that table. For example, you could use it to distinguish between bot and human users.

```
SELECT user, count(*) AS num
FROM nodes
GROUP BY uid
ORDER BY num DESC
LIMIT 10;
```

```
woodpeck_fixbot 338103
greggerm        212390
Zirnch           26848
maxerickson      17340
John Wrenn       16771
ZeLonewolf       16466
morganwahl        9693
Roman Guy         9432
GeoStudent        9419
TIGERcnl          8440
```

Looking at the top ten users in nodes, it is clear that some bots are an order of magnitude higher in contribution than some other users. I think it would be very useful to have a table with more information about each user or bot and the kind of data they upload. It might be useful to identify some biases in the data.

One other issued that I had occured during csv import to SQL. I got a 'datatype mismatch' error on the first line for the nodes and ways tables only. I'm not sure what in the header causes this error.

The error: nodes.csv:1: INSERT failed: datatype mismatch