OpenStreetMap+Project.md

Udacity OpenStreetMap Project: New Orleans, LA

For this project I chose to study New Orleans, LA and some of the surrounding municipalities.

I obtained a raw .osm map from Mapzen: https://mapzen.com/data/metro-extracts/metro/new-orleans_louisiana/

This raw .osm file was ~1.2GB uncompressed. In order to reduce processing times I used the sample osm code from the project outline with a k value = 10 to obtain a subset of the larger database. This code can be found with my project in the file sample_osm.py.

Project Purpose

This project serves to demonstrate my understanding of skills learned in the Udacity Data Analysis Nanodegree Lesson: Data Wrangling. In this lesson we studied the process of auditing a dataset and ways to correct/standardize a dataset to make it easer to process. Using OpenStreetMap data can be problematic due to many types of human error when entering the data into the database. In order to gain confidence in our dataset we need to assess the data quality, audit the data based off of conditions we discover, define a set of operations necessary to correct the data and programatically clean the data up to our standards.

Assessing Data Quality

It is necessary to first assess the quality of the data we are working with, to determine what we might need to do clean it up. The best ways to do this is to come up with some tests that programatically parse the data and display relevant information to us about the dataset.

I started by taking a look at the number of unique users that have contributed to this dataset. In order to accomplish this I adapted the code used in our lessons to work with my dataset as users.py.

Problem 1: Out of Memory!

The first major problem that I encountered is that my computer was quickly running out of memory while parsing through the dataset, even with smaller dataset samples. After some research on the Udacity forums I found a post from Myles Here! that greatly reduced the memory usage of my code. Code snippet can be seen below for reference!

```
def get_element(osm_file, tags=('node', 'way')):
    """Yield element if it is the right type of tag"""
    """This code handles a single parse event at a time
        and then clears it from memory after it has returned it!"""
    context = ET.iterparse(osm_file, events=('start', 'end'))
    _, root = next(context)
    for event, elem in context:
        if event == 'end' and elem.tag in tags:
            yield elem
            root.clear()
```

After integrating this into my users.py code, I was able to run through my 128MB dataset with no memory issues. I had to implement this logic in each unique piece of code used in this project in order to keep memory usage manageable.

Unique Users

After running users.py on my dataset I noticed of couple of interesting things.

First, there are 563 unique users that have contributed to this dataset. Thats alot of users! Second, it looks like there are a couple of bots that contribute to the dataset e.g. 'woodpeck_fixbot' and 'woodpeck_repair'. I wonder how many contributions these bots are responsible for compared to other unique users?

Tags Assessment

In the data wrangling lesson we took a look at all of the tags in the dataset to get an idea as to how the related "k" values are formatted. This will help us in auditing our datset later. I adapted the tags.py code from the lesson to work with this dataset. Running this code against this dataset should give us a count of the following:

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

After running tags.py I got the following output:

```
{'lower': 65769, 'lower_colon': 68988, 'other': 31490, 'problemchars': 1}
```

And looking at some samples of the categories:

```
lower:
railway
name
place
railway
highway

lower_colon:
addr:street
addr:housenumber
addr:city
addr:housenumber
addr:postcode
addr:state
addr:street

other:
gnis:ST_num
gnis:ST_alpha
gnis:County_num
gnis:Class
gnis:County

problemchars:
payment method
```

This gives us a good idea of different tags formatting we are up against when we go to audit our data.

Auditing Data

Now that we have a general idea of what our data looks like, we need validate the data with a structured audit. In our lessons we learned the following elements to look at when auditing a dataset:

- Validity: Conforms to a Schema
- Accuracy: Conforms to Gold Standard - need a set of trusted data for comparison
- Completeness: All records present
- Consistency: Matches other data in dataset
- Uniformity: All data use same units?

Since we know that we want to import this data into a SQL database, we should focus first on **Data Validity**. In particular we need to ensure that our data conforms to a common schema that allows it to play nice with SQL. Second, we will need to focus on **Consistency** and **Uniformity**. Since our data is heavily modified by human hands, it is very likely that it is riddled with abbreviations, typos and formatting errors. This will prevent us from getting accurate aggregations once the data is imported into a SQL database.

Street Names, In Particular

Following along with the data cleaning lesson, lets take a look at the street names component of the dataset. This can be done by adapting the audit.py code from our lessons, which takes a look at expected street types (e.g. Street/Avenue) and flags when it finds street types that are not epected. Additionally we can create a mapping to keep track of expected street types that may be abbreviated.

Running the audit.py script with the expected street types as left from the lesson revealed another issue.

Problem 2: Louisiana Expectations

Here is a snippet from the initial output:

```
Output:
(udacity_dev) C:\Users\Andy\Documents\OpenStreetMap>python audit.py
{'134': set(['Road 134']),
 '259': set(['Road 259']),
 '266': set(['Road 266']),
 '308': set(['Road 308']),
 '312': set(['Road 312']),
 '361': set(['Road 361']),
 '433': set(['Road 433']),
 '503': set(['Road 503']),
 '53': set(['Highway 53']),
 '530': set(['Road 530']),
 '603': set(['Highway 603']),
 '604': set(['Highway 604']),
 '82': set(['Rt 2 Box 82']),
 '90': set(['Highway 90']),
 'A': set(['Avenue A']),
 'Bayou': set(['Rotten Bayou', 'Turkey Bayou']),
 'Bend': set(['Giveans Bend']),
 'Circle': set(['Ahekolo Circle',
                'Bayou Circle',
                'Boggs Circle',
                'Clearwater Circle',
                'Dogwood Circle',
                'Donna Circle',
                'East Wheaton Circle',
                'Five Oaks Circle',
                'Hanalei Circle',
                'Harbor Circle',
                'James Circle',
                'Lynn Circle',
                'Meadowwood Circle',
                'Pecan Circle',
                'Sheepshead Circle',
                'Summerhaven Circle',
                'Tyler Circle',
                'West Rockton Circle',
                'West St Andrews Circle']),
 'Colette': set(['Rue Colette']),
 'Combel': set(['Combel']),
 'Dogwood': set(['Dogwood']),
 'Driveway': set(['Pine Driveway']),
 'Duthu': set(['Duthu']),
 'E': set(['Avenue E']),
 'East': set(['Diamondhead Drive East']),
 'Highway': set(['Behrman Highway', 'Chef Menteur Highway']),
 'Hollow': set(['Possum Hollow']),
 'Hwy': set(['Shortcut Hwy']),
```

```
'John': set(['Grand Route St John']),
'Knee': set(['Cypress Knee']),
'Lasalle': set(['Rue De Lasalle']),
'Loop': set(['Youngswood Loop']),
'Mignon': set(['Rue Mignon']),
'Nadine': set(['Rue Nadine']),
'Nichole': set(['Rue Nichole']),
'North': set(['Diamondhead Drive North']),
'Point': set(['Cashew Point', 'Poplar Point']),
'Run': set(['Wilderness Run']),
'Sycamore': set(['Sycamore']),
'Taylor': set(['General Taylor']),
'Trace': set(['Camellia Trace']),
'Walk': set(['Magic Walk']),
'Washington': set(['Washington']),
'Way': set(['Alafia Way', 'Linohau Way', 'Manini Way', 'Opla Way']),
'Wood': set(['Wood'])}
```

The following street types are perfectly acceptable down here and should be added to the expected street types.

```
"Bayou", "Circle", "Bend", "Highway", "Point", "Way", "Hollow", "Trace", "Run", "Loop", "Knee"
```

Additionally, Street names in this area are a blend of French and English representations (carried over from wayyy back before the Louisiana Purchase), So streets such as "Rue Mingnon" and "Rue Nichole" should be treated as valid. This will require some changes to the audit/cleaning code, more than just adding a street type to the expected street type list.

Problem 3: Abbreviations

Another problem that is plaguing the street types is inconsistent abbreviations. Taking a look at the 'St' key our mapping dict generated with audit.py:

```
'St': set(['543 5th St',
           'Banks St',
           'Bourbon St',
           'Canal St',
           'Chartres St',
           'Dauphine St',
           'Decatur St',
           'E Rutland St',
           'East St',
           'Johnny Dufrene St',
           'Lasalle St',
           'Laurel St',
           'Magazine St',
           'Magazine Street;Magazine St',
           'Marais St',
           'N Rendon St',
           'North St',
           'Octavia St',
           'Ponce De Leon St',
           'Prytania St',
           'Royal St',
           'S Broad St',
           'S Dupre St',
           'South St',
           'South St Patrick St',
           'Tchoupitoulas St',
           'Toulouse St',
           'Whitney St']),
```

All of these abbreviations should be cleaned, replacing St with Street. This will result in a more consistent dataset. Likewise, abbreviations for Avenue, Road, Loop, Boulevard and Highway should also be cleaned.

Cleaning Data

Now that we have performed our audit, we can begin cleaning the data and importing it into out SQL database!

As we covered in our lessons, I will need to parse the OSM XML map file and transform it into tabular format. This will allow the file to be written to .csv files for importation into a SQL database.

The process for this transformation is as follows:

- Use iterparse to iteratively step through each top level element in the XML
- Shape each element into several data structures using a custom function
- Utilize a schema
- Clean the Data
- Write each data structure to the appropriate .csv files

Using the data.py file provided in our lessons as a starting point, I was able to clean and shape the data all in one step. The cleaning done

Cleaning Street Names

The first cleaning operation I chose was to ensure that all of the street type abbreviations (e.g Ave. Av. St.) were changed to the full street type (e.g. Avenue, Street).

Problem 3: (Pot) Holy Abbreviations

After cleaning the data using data.py the first time I noticed that the 'St' regex key also changed all of the abbreviations for "Saint" (St.) to Street! The state of Louisiana is divided into "Parishes" that take the place of what would be "Counties" in most other states. The roman catholic origins of this system result in many of the Parishes being named after Saints (e.g. Saint Tammany Parish, Saint Charles Parish etc).

The solution for this issue was to implement checks in my cleaning routines that ensure only street names are attempted to be cleaned. Previously I was passing all tags to my cleaning function. This worked!

Cleaning Postal Codes

The next cleaning operation I implemented is related to postal codes. I implemented code that checks if the postal code is > 5 characters and contains some non numeric digits. If this returns true, then we remove the non numeric characters and return only the numeric ones. This should clean up any data that includes states in the zip (e.g. LA70123).

The following two Stack Overflow posts helped me accomplish this:

- REGEX Check for Only Numbers
- Strip Non Numeric Characters with REGEX

Importing Data into SQL Databaase

Following the creation of my cleaned, shaped .csv files I needed to import them into a SQL database. I used a discussion here on the Udacity forums as a reference to manually import the csv files into the database.

```
C:\Users\Andy\Documents\OpenStreetMap>sqlite3 nodes
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import nodes.csv nodes
sqlite> .schema nodes
CREATE TABLE nodes(
  "id" TEXT,
  "lat" TEXT,
  "lon" TEXT,
  "user" TEXT,
  "uid" TEXT,
  "version" TEXT,
  "changeset" TEXT,
" TEXTestamp
);
sqlite> .tables
```

```
nodes       nodes_tags  ways        ways_nodes  ways_tags
sqlite> .schema nodes_tags
CREATE TABLE nodes_tags(
  "id" TEXT,
  "key" TEXT,
  "value" TEXT,
" TEXTe
);
sqlite> .schema ways
CREATE TABLE ways(
  "id" TEXT,
  "user" TEXT,
  "uid" TEXT,
  "version" TEXT,
  "changeset" TEXT,
" TEXTestamp
);
sqlite> .schema ways_nodes
CREATE TABLE ways_nodes(
  "id" TEXT,
  "node_id" TEXT,
" TEXTition
);
sqlite> .schema ways_tags
CREATE TABLE ways_tags(
  "id" TEXT,
  "key" TEXT,
  "value" TEXT,
" TEXTe
);
```

Problem 4: Last Column in .csv file not imported correctly

For all of the tables , the key for the last column did not import correctly. I followed the tutorial here and was able to manually create the tables before importing and then import the .csv files without headers. This fixed my issue!

e.g.

```
sqlite> CREATE TABLE nodes_tags(
   ...> id TEXT NOT NULL,
   ...> key TEXT NOT NULL,
   ...> value TEXT NOT NULL,
   ...> type TEXT NOT NULL);
```

Querying the Database

Now that we have a functional SQL database, we can start querying our data and computing statistics/answering questions!

Here are some basic statistics about the dataset:

Size of Files

```
nodes...........46.4MB
nodes_tags.......637KB
ways............2.2MB
ways_nodes.......1KB
ways_tags........4.9MB
```

Number of Nodes

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

1048573

Number of Ways

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

37725

Number of Unique Users

Since our nodes and ways tables are separate we will want to do our aggregation on a join of two tables.

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

495

Previously this query was 563, so it is interesting that it differs from ealier. I did some investigating and found that opening/saving the file generated from my data.py code in Excel resulted in the whole file not being loaded/saved. This must have trimmed off some users.

Next up I would like to look at the users that have the most number of entries in the dataset. This is related to my question in the data quality assessment concerning both users and how their contributions stack up against other users.

Lets look at the nodes table:

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

Matt Toups|337791
ELadner|110752
wvdp|75801
coleman_nolaimport|30934
ELadnerImp|21872
woodpeck_fixbot|21214
Matt Toups_nolaimport|4755
Minh Nguyen_nolaimport|3700
ceseifert_nolaimport|3344
TIGERcnl|2557
```

It seems that there are other users with lots of entries besides the bots! Some of these users have 'import' in the names so they were likely part of some batch change to the map.

Now lets look at an aggregation of the different tag types in the dataset.

nodes_tags:

```
SELECT count(*),nodes_tags.type
FROM nodes_tags
GROUP BY nodes_tags.type
ORDER BY count(*)desc
LIMIT 10;

19902|addr
11024|regular
3426|gnis
28|name
10|contact
8|ref
6|is_in
6|payment
6|shop
4|disused
```

ways_tags:

```
SELECT count(*),ways_tags.type
FROM ways_tags
GROUP BY ways_tags.type
ORDER BY count(*)desc
LIMIT 10;

58972|regular
29876|NHD
29619|addr
27879|tiger
290|gnis
42|note
23|building
15|hgv
15|source
14|destination
```

Improving the Dataset

I think that this dataset could be improved further with some additional orginazation of the data into more discreet types. For both the nodes_tags and ways_tags tables, the tag type 'regular' makes up the majority of the entries. This does not really tell us a whole lot and limits our opportunities to learn from the data.

A good way to implement this would be to audit the data using additional regex keys to get a clear picture of the relation between 'key' and 'value' in the tags. If patterns begin to emerge, for example there are a number of occurences of 'key'=highway, 'value'=turning_circle, 'type'=regular, then we could assign a new type to tags that fit that pattern.

Additionally, if OpenStreetMap included some sort of like/dislike consensus system, the data quality could be more easily accessed. If there was a tag associated with each element that allowed for users to vote on the quality, of a particular entry that would provide a great initial guide of what to clean up.

In order for this to be implemented well, OpenStreetMap would likely need to overhaul its GUI for it to be effective. This could provide some hurdles if there is not a large core team of OpenStreetMap developers.

Conclusion

After the completion of this lesson and project, the importance of data auditing and cleaning is very clear to me. Without a good test plan and cleaning blueprint it becomes very difficult to fully understand messy data. Formatting errors can skew statistics, and vague categories can limit visibility into how the data is organized. I look forward to applying the skills learned in this lesson to future data analysis opportunities!