P3 - Data Wrangling Project

July 13, 2017

1 Wrangling OpenStreetMap Data

Dominic Nguyen

1.1 Introduction

OpenStreetMap data from the city of Austin, TX was parsed and cleaned using Python and MongoDB. This data was a pre-selected metro area by MapZen and can be retrieved using the OSM XML button on this webpage: https://mapzen.com/data/metro-extracts/metro/austin_texas/. The OpenStreetMap data is in XML format so we wrote a python script, *convert_to_json.py*, to parse the data and export it into a JSON file. The *process_map* function in this file was previously provided by a lesson in the Udacity Data Wrangling course.

Austin was selected as the city because I explored it quite a bit during my co-op in Texas last semester. By cleaning the data, I hope to help future tourists that may be using apps that pull from this dataset.

Three fields in the dataset that will be audited include building heights, state address, and zip codes.

PLEASE NOTE:

- The results in this document are from inquiring a sample of the Austin dataset, the *sample_20.osm.json* file. This dataset was obtained by selecting every 20th top node of the entire dataset using the parsing code provided in the Udacity project description.
- Only the results from the 'Data Overview' and 'Additional Dataset Exploration with MongoDB and Ideas for Future Analysis' sections are obtained using the entire Austin dataset, the *austin_texas.osm.json* file.
- An easy way to tell which results came from which dataset is to look into which MongoDB collection the queries are called on.
- For example, `db.sample_20.aggregate(...)` returns results from the sample dataset.

1.2 Problems Encountered in the Map

Here we explore some of the problems in three of the fields we audited.

PLEASE NOTE: Running the code in this section will not return the same sample results in this report as we have added the cleaning functions to the *convert_to_json.py* file which has changed the dataset.

1.2.1 Building Heights

After examining the JSON file, we noticed that the values of building heights field, `height`, were string values instead of numerical values. The units of this height is presumably in meters.

Running aggregation pipelines to sort this value as a string turned out to be problematic. Rather than convert the value during the MongoDB call, we decided to resolve the problem at its source.

Thus, in our *convert_to_json.py* file, we added a condition to convert attribute values into either floats or integers if they could be. This would make sorting for many other fields work correctly. Additionally, it also decreased the size of the *sample_20.osm.json* file by a few MB.

Running the `sample_buildings` MongoDB aggregation pipeline in the *buildings.py* file, we found values with more than 2 decimal places for the building height field. Below, the following code displays a sample of the building heights and illustrates this result. That amount of digits is oddly specific, unnecessary, and may have been an input error.

```
In [2]: # Function displays 'size' number of building addresses and heights
        def sample_buildings(size):
            buildings = db.sample_20.aggregate([
                {"$match": {"building": "yes", "height": {"$exists": 1}, \
                            "address": {"$exists": 1}}},
                {"$project": {"_id": {"$concat": ["$address.housenumber", \
                                                  " ", "$address.street"]},
                            "building_height": "$height"}},
                {"$limit": size}
            ])

            print("\n- Sample of " + str(size) + " Buildings -")
            pprint.pprint([doc for doc in buildings])
```

```
  - Building Heights -
  {'height': 5.04, 'address': {'street': 'East Lisa Drive',
'housenumber': '100'}} {'height': 4.94, 'address': {'street': 'East
Lisa Drive', 'housenumber': '108'}} {'height': 3.98, 'address':
{'street': 'East Lisa Drive', 'housenumber': '211'}} {'height':
4.44, 'address': {'street': 'Miranda Drive', 'housenumber': '6802'}}
{'height': 4.67, 'address': {'street': 'Miranda Drive', 'housenumber':
'6907'}} {'height': 5.21, 'address': {'street': 'Priscilla Drive',
'housenumber': '6911'}} {'height': 4.64, 'address': {'street':
'Priscilla Drive', 'housenumber': '7001'}} {'height': 4.48, 'address':
{'street': 'Miranda Drive', 'housenumber': '6903'}} {'height': 4.41,
'address': {'street': 'Isabelle Drive', 'housenumber': '7005'}}
{'height': 4.430000000000002, 'address': {'street': 'Twin Crest Drive',
'housenumber': '7008'}}
```

Thus, to standardize the data for easier use, we floated the data so each building height has only 2 decimal digits. We did this through defining several auditing functions in the *Cleaning Functions for XML* section of the *buildings.py* file and calling them in the *convert_to_json.py* file.

1.2.2 State Address

Looking into the state field of the address subdocument through the *state.py* file, we noticed that the users had entered many variations of Texas for the state field.

```
In [3]: def find_states():
            states = db.austin_texas.aggregate([
                {"$match": {"address.state": {"$exists": 1}}},
                {"$group": {"_id": "$address.state", "count": {"$sum": 1}}},
                {"$sort": {"count": -1}}
            ])

            print("\n- Sample of Values for State Field -")
            pprint.pprint([doc for doc in states])

   - Sample of Values for State Field -
   [{'_id': 'TX', 'count': 128},  {'_id': 'tx', 'count': 36},  {'_id':
'Tx', 'count': 3},  {'_id': 'Texas', 'count': 2},  {'_id': 'texas',
'count': 1}]
```

To improve searching and sorting in the future, we converted all of these variations to 'TX', the most common one. We did this through defining several auditing functions in the *Cleaning Functions for XML* section of the *states.py* file and calling them in the *convert_to_json.py* file.

In the *states.py* file, one will notice that the `states` list contains more variations than seen in the above result. These are values seen from running the data with the entire dataset.

1.2.3 Zip Codes

Looking into the zip codes, we found a few documents which had a string in the `postcode` field instead of an integer.

Below is the address of one such case. The reason this value was not converted into an integer from a previous implementation of converting fields to integers and floats is because of nonnumerical values such as a hyphen.

```
In [4]: def find_zip_errors():
            str_zips = db.sample_20.aggregate([
                {"$match": {"address.postcode": {"$type": "string"}}},
                {"$project": {"_id": "$address"}}
            ])

            print("\n- Addresses that have Strings in Postcode Field -")
            pprint.pprint([doc for doc in str_zips])

   - Addresses that have Strings in Postcode Field -
   [{'_id': {'housename': 'Suite # 150',           'housenumber':
'9041',           'postcode': '78758-7008',           'street':
'Research Boulevard'}}]
```

To resolve this, we defined an update_zip function in the *zipcodes.py* file which extracts only the 5 digit zip code in a string or longer integer value. This function was then called by the *convert_to_json.py* file to further clean the data before importing it into the MongoDB database.

```
In [5]: # Function to extract 5 digit zipcode
        def update_zip(v_value):
            try:
                # Retrieve first digits as a string
                digits = re.findall(r'\d+', str(v_value))[0]
                # Return first 5 digits of sequence as an integer
                return int(digits[:5])

            except IndexError: # No digits in string
                return v_value
```

1.3 Data Overview

Below is a statistical overview of the dataset using MongoDB queries.

1.3.1 File Sizes

```
Austin_texas.osm - 1.4 GB
  Austin_texas.osm.json - 1.5 GB
  Sample_20.osm - 72 MB
  Sample_20.osm.json - 79MB
```

1.3.2 Number of Documents

```
> db.austin_texas.find().count()
  6846953
```

1.3.3 Number of Nodes

```
> db.austin_texas.find({"type":"node"}).count()
  64043231
```

1.3.4 Number of Ways

```
> db.austin_texas.find({"type":"way"}).count()
  442465
```

1.3.5 Number of Unique Users

```
> db.austin_texas.distinct("created.user").length
  1326
```

1.3.6 Number of Bicycle Parking Areas

```
> db.austin_texas.find({"amenity":"bicycle_parking"}).count()
  85
```

1.3.7 Number of Trees

```
> db.austin_texas.find({"natural":"tree"}).count()
  2638
```

1.4 Additional Dataset Exploration and Ideas for Future Analysis

Here we explored the dataset further using multiple MongoDB aggregation queries and pose ideas for further analysis. We will display the most interesting of these MongoDB pipelines and the sample results they generated. The other pipelines not shown can be found in the *buildings.py*, *states.py*, *zipcodes.py*, and *explore.py* files.

Tallest and Shortest Buildings

```
In [6]: # Function displays 'size' number of the tallest and shortest buildings
        def tallest_shortest(size):
            tallest_buildings = db.austin_texas.aggregate([
                {"$match":{"building": "yes", "height": {"$exists": 1}, \
                           "address": {"$exists": 1}}},
                {"$project": {"_id": {"$concat": ["$address.housenumber", \
                                                  " ", "$address.street"]},
                              "building_height": "$height"}},
                {"$sort": {"building_height": -1}},
                {"$limit": size}
            ])

            shortest_buildings = db.austin_texas.aggregate([
                {"$match": {"building": "yes", "height": {"$exists": 1}, \
                            "address": {"$exists": 1}}},
                {"$project": {"_id": {"$concat": ["$address.housenumber", \
                                                  " ", "$address.street"]},
                              "building_height": "$height"}},
                {"$sort": {"building_height": 1}},
                {"$limit": size}
            ])

            print("\n- Top " + str(size) + " Tallest Building Heights -")
            pprint.pprint([doc for doc in tallest_buildings])

            # Need to find smallest value to determine appropriate unit
            print("\n- Top " + str(size) + " Shortest Building Heights -")
            pprint.pprint([doc for doc in shortest_buildings])
```

```
  - Top 5 Tallest Building Heights -
  [{'_id': '10102 Aircraft Lane', 'building_height': 69.5},  {'_id':
'3900 West Howard Lane', 'building_height': 62.8},  {'_id': '303 West
15th Street', 'building_height': 57.7},  {'_id': '3823 Avenue F',
'building_height': 48.1},  {'_id': '3000 North Interstate Highway 35
Service Road',   'building_height': 44.4}]
```

```
  - Top 5 Shortest Building Heights -
  [{'_id': '2801 Singlefoot Lane', 'building_height': 1.97},  {'_id':
'4809 Eilers Avenue', 'building_height': 2.01},  {'_id': '4713 East
Yager Lane', 'building_height': 2.3},  {'_id': '7001 Ranch to Market
Road 2222', 'building_height': 2.3},  {'_id': '14702 Deaf Smith
Boulevard', 'building_height': 2.4}]
```

Surprisingly, the shortest buildings are only around 2 meters tall. Pulling these addresses on Google Maps, we can see the buildings are presumably taller than that. Thus, it is possible the value inputted into the 'height' field may not have been for the building in these cases. Further exploration on this abnormality is needed to confirm this hypotheses.

Another interesting area for exploration would be to gather the building heights data and group them by location. The user then would be able to use this data to help predict population density by location.

1.4.1 Empty Zip Code Fields

In an interesting discovery during the auditing of the zip code field, we found **129849** documents that had an address but no postcode field. Comparing this to the **86640** documents that do have a zip code, we find that **60%** of the documents with addresses don't have postcodes. Some examples are below.

```
In [7]: # Return 'limit' number of documents that have an address but no zip code
        def find_no_zips(limit):
            no_zips = db.austin_texas.aggregate([
                {"$match": {"address.housenumber": {"$exists": 1},
                            "address.postcode": {"$exists": 0}}},
                {"$project": {"_id": "$address"}},
                {"$limit": limit}
                ])

            print("\n- Sample of " + str(limit) + \
                  " Addresses with no Postcode Field -")
            pprint.pprint([doc for doc in no_zips])

        # Function returns number of addresses with no postcode field
        def count_no_zips():
            num_no_zips = db.austin_texas.aggregate([
                {"$match": {"address.housenumber": {"$exists": 1},
                            "address.postcode": {"$exists": 0}}},
                {"$group": {"_id": "num_no_zips", "count": {"$sum": 1}}}
                ])

            print("\n- Number of Addresses with no Postcode Field -")
            pprint.pprint(list(num_no_zips))
```

```
  - Sample of 5 Addresses with no Postcode Field -
  [{'_id': {'housenumber': '900', 'street': 'W. Slaughter Lane'}},
{'_id': {'housenumber': '2703', 'street': 'Pecan St.'}},  {'_id':
```

```
{'housenumber': '9801'}},  {'_id': {'housenumber': '10910', 'street':
'Domain Drive'}},  {'_id': {'housenumber': '2808', 'street': 'N
IH-35'}}]
   - Number of Addresses with no Postcode Field -
   [{'_id': 'num_no_zips', 'count': 129849}]
```

A potential solution, though presumably time consuming, would be creating a document that maps ranges of coordinates, latitude and longigtude values in the the `pos` field to a zip code. Then, running our collection against this document, we could programmatically populate zip codes for those addresses that do not have one.

Though not important to find the location on the OpenStreetMap map, the zip code information will be important for mailing services as mail is sorted by zip.

1.4.2 Bar Hours

One last area for future data cleaning would be hours of operation of businesses. To achieve a small sample, we selected all the bar locations with listed opening hours. The code is pasted below and can be found in the *explore.py* file.

```
In [8]: # Find all bars with opening hours
        def find_bar_hours():
            bar_hours = db.austin_texas.aggregate([
                {"$match": {"amenity": "bar", "opening_hours": {"$exists": 1}}},
                {"$project": {"_id": "$name", "hours": "$opening_hours"}}
            ])

            print("\n- Bar Hours in Austin -")
            pprint.pprint([doc for doc in bar_hours])
```

```
   - Bar Hours in Austin -
   [{'_id': 'Coyote Ugly', 'hours': 'Mo-Th 17:00-2:00;Fr-Su
12:00-2:00'},  {'_id': 'HandleBar', 'hours': 'Mo-Sa 14:00-02:00;
Su 12:00-02:00'},  {'_id': 'Workhorse Bar', 'hours': '11am to
12am open 7 days a week'},  {'_id': 'Dive', 'hours': '4pm-2am'},
{'_id': 'Dark Horse Tavern',   'hours': 'Tu-Fr 16:00-24:00;Sa
16:00-24:00;Su 0:00-1:00;Su 16:00-24:00'},  {'_id': 'The Jackalope',
'hours': '11:00-2:00'},  {'_id': 'MugShots',   'hours': 'Su
18:00-2:00;Mo 19:30-2:00;Tu,We 20:00-2:00;Th,Fr 17:00-2:00;Sa
'           '19:00-2:00'},  {'_id': 'Stompin Grounds', 'hours':
'11AM - 2AM'},  {'_id': "Dirty Bill's", 'hours': 'Mo-Su 16:00-02:00'},
{'_id': 'Butterfly Bar Austin',   'hours': 'Mo-We 17:00-24:00;
Th-Fi 16:00-24:00; Sat 17:00-24:00; Su '            '00:00-01:00,
17:00-24:00'},  {'_id': 'Rio Rita', 'hours': '10:00AM-02:00AM'},
{'_id': 'King Bee', 'hours': '05:00PM-02:00AM'},  {'_id': 'The
Grackle', 'hours': '12pm-2am'}]
```

As can be seen, the values of the hours are of all sorts of formats based on what the user who submitted the data is familiar with. Additionally, some of the bars don't list which days they are open either. Standardizing this data for future comparisons would be ideal. However, parsing this data and converting it into a single format would be very complex. Thus, we suggest tackling the problem at its source. OpenStreetMap could provide a standardized format to input hours.

1.5 Conclusion

In summary, implementation of the above auditing scripts will allow Open Street Map users to have access to cleaner and more complete addresses. Furthermore, further analysis of building heights will be quicker and cleaner with standardized data.

Some problems that may arise by implementing this improvement are potential additional conversions of float and integer fields to strings for manipulation (i.e. zip code extraction). If this improvement is implemented, the scripts would be needed to run again as more data as added to ensure consistency.

Another problem that may arise would be future zip code entries that contain no digits. Our current implementation would accept this string data and display it. Many changes will need to be made to include this data in another field if it is related or exclude it completely.

1.6 Sources

https://stackoverflow.com/questions/4289331/python-extract-numbers-from-a-string

1.7 Description of Attached Files

- *convert_to_json.py* - parses XML formatted OSM file and converts it to JSON format. Imports cleaning functions from other files.
- *buildings.py* - script file that explores `height` field of buildings and provides cleaning functions for the conversion file.
- *states.py* - script file that explores `state` field of the `address` subdocument and provides cleaning functions for the conversion file.
- *zipcodes.py* - script file that explores `postcode` field of the `address` subdocument and provides cleaning functions for the conversion file.
- *explore.py* - file that explores other fields
- *create_sample_osm.py* - creates sample from entire Austin dataset by picking selecting every kth top node.