**Cleaning, Wrangling and Analyzing Seattle Open Street Map**

**Jenny Hung**

1. **Introduction**

We investigated into the Seattle Open Street Map using MongoDB. However, before the analysis of the Seattle OSM can begin, we must first inspect the data contained within the Open Street Map of Seattle, then clean and wrangle the data before we can import the JSON-format data into MongoDB for analysis.

1. **Data Investigation & Wrangling** 
   1. **Technical Challenges Due to Size**

Due to the size of the dataset, we need a way to systematically slice the original dataset for a workable sample to explore. To this end, we use the get\_element(…) function to achieve this. This function takes the input of the source file, and returns the top-level element <node>, <way> and <relation> tags and write the results to SAMPLE\_FILE. The k value is changed from large to small so that the resulting SAMPLE\_FILE ends up at different sizes. To use this function we should try using a larger k, then move on to an intermediate k before processing your whole dataset.

* 1. **Develop a Dictionary for All Tags In the Original Dataset**

Our goal here is to end up with a Python dictionary for all the top-level the tags in the original dataset, so that we know what needs to be wrangled in the data. The count\_tags(…) function achieves this.

Our next step is to explore what is contained within each tag type. To achieve this, we used SAX-style element tree parsing to compile a list of attributes for each of the tag types that we have discovered by using the count\_tags(...) function in the paragraph above. Namely, we compiled a list of attributes for <bound>, <member>, <nd>, <node>, <osm>, <relation>, <tag> and <way>.

* 1. **Observations and Establish Goals of Data Wrangling**

Based on the above, we started to get a sense of that the xml document was indeed used as a data storage tool, in that each of the elements in the above lists of attributes represents an **attribute of an tag**. Our data wrangling goal then, is to transform the information embedded in the OSM xml document into a JSON document with a flexible schema. In general, this schema will reflect the Python data structure of a dictionary, where an **attribute** is used as a **key**, and the **value of the attribute** is the corresponding **value** of the dictionary.

Programmatically, it is important to note that since a JSON document can be arbitrarily complex, it is an excellent fit for MongoDB, whose greatest feature is its flexible schema.

1. **Problems Encountered in Seattle OSM**

After generating several smallish subsets of Metro Seattle open street map xml files, I noticed the following problems with the data:

1. **Inconsistent representation of street name (original name and abbreviated names)**

Examples of this problem are King George Boulevard and also be written as King George Blvd, and Granville Street can also show up as Granville St or Granville St.

1. **Inconsistent representation of addresses under <tag>**

Address can either be represented by a many subtags, such as <tag k='addr:city'>, <tag k='addr:street'>, <tag k='addr:postcode'>, <tag k='addr:housenumber'>. However, at times, an address can also be expressed as <tag k='addr:'>.

1. **Inconsistent zip code format**

Zip codes, which are expressed as postcodes in OSM, are not always shown in a consistent format. Often it is shown in the standard 5-digits such as 98201, but quite often it is shown in the older 9-digit format.

From a visual inspection of the subset of Seattle OSM, we understood that <tag> contains address information. In particular, tags with attribute of k of addr:street contains street names that tend to be described inconsistently in the dataset. Therefore, our next goal is to develop a data audit plan that works specifically on tags with addresses.

1. **Data Cleaning Plan – Street Types**

We now proceed with the address cleaning. At a high level, our audit plan can be described as follows:

1. First we explored the variations and types of street name representation found in the Seattle OSM data. Then we normalized the abbreviated street type representations.

Please see the accompanying clean\_strret.py file for the detailed implementation of address cleaning. At a high level, at the end of this stage, we end up with a Python dictionary showing the entire collection of street types after we have done our initial cleaning. Now we see that a vast majority of street types no longer bears problems. However, some of the street types are obviously wrong. Most notably, whenever Suite number / apartment number is present in the street name, the code has confused it with the name of the street. This needs our attention.

To address this problem, we should make sure that our subsequent code to clean and wrangle the OSM data will shape the raw data in such a way that will avoid confusing the suite number / apartment number with the street name. A convenient way to achieve this is to present an address in the JSON document (namely, the cleaned file) with schema such as this:

"address": {"street": "3401 Evanston Ave N, Suite A"}

1. Second transform the key-value pairs found in <tag> as described earlier.
2. Finally, we will normalize the overall address representation by removing the tags with attribute **addr:**, and make sure that our final street key will show the street name as well as the suite / apartment number.
3. **Data Cleaning Plan – Postcode**

We now proceed with the address cleaning. At a high level, our audit plan can be described as follows:

* 1. First we explored the variations of zip codes found in the OSM. This was done by clean\_postcode.py file. It was then discovered that there exists a problem of inconsistent zip code formats, as described in earlier sections.
  2. Clean\_postcode.py uses regular expression to standardize the two different zip code formats (into the acceptable l 5-digit format).

1. **Inserting into MongoDB Database**

After this step, we then insert into the MongoDB database using the following cmd command:

mongoimport -host 127.0.0.1:27017 --db osm --collection seattle\_osm --drop --file C:\Users\Jenny\Documents\Mathfreak\_Data\School\Data\_Analysis\_ND\Project3\seattle\_washingotn.osm.json

1. **Exploring the Database in MongoDB**
2. File Size

seattle\_washingotn.osm ......... 1,649 MB

seattle\_washingotn.osm.json .... 1,870 MB

1. Overview of Seattle Area Map

from pymongo import MongoClient

def get\_db():

client = MongoClient('localhost:27017')

db = client.osm

return db

db = get\_db()

db.seattle\_osm.find().count()

We see that once cleaned and imported, the Seattle OSM collection has approximately 8.3 million data points.

db.seattle\_osm.find({"type":"node"}).count()

We see that the Seattle OSM collection has roughly 7.5 million nodes.

db.seattle\_osm.find({"type":"way"}).count()

The Seattle OSM has 750175 counts of <way> tags.

len(db.seattle\_osm.distinct("created.user"))

The Seattle OSM has 3280 distinct users.

1. Top 10 Amenities in Metro Seattle

amenity\_result = db.seattle\_osm.aggregate([{"$match":{"amenity":{"$exists":1}}}, {"$group":{"\_id":"$amenity","count":{"$sum":1}}},

{"$sort":{"count":-1}}, {"$limit":10}])

for a in amenity\_result:

print(a)

The following query result shows the top 10 amenities contained in Seattle OSM.

{'count': 10753, '\_id': 'parking'}

{'count': 3318, '\_id': 'bicycle\_parking'}

{'count': 3293, '\_id': 'restaurant'}

{'count': 2848, '\_id': 'bench'}

{'count': 2458, '\_id': 'school'}

{'count': 1645, '\_id': 'place\_of\_worship'}

{'count': 1578, '\_id': 'fast\_food'}

{'count': 1499, '\_id': 'cafe'}

{'count': 1294, '\_id': 'waste\_basket'}

{'count': 1153, '\_id': 'fuel'}

1. Top 10 Place of Worship in Metro Seattle

religion\_result = db.seattle\_osm.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"place\_of\_worship"}},

{"$group":{"\_id":"$religion", "count":{"$sum":1}}},

{"$sort":{"count":-1}}, {"$limit":10}])

for r in religion\_result:

print(r)

The following query result shows the top 10 places of worship contained in Seattle OSM.

{'count': 1500, '\_id': 'christian'}

{'count': 80, '\_id': None}

{'count': 20, '\_id': 'jewish'}

{'count': 17, '\_id': 'buddhist'}

{'count': 7, '\_id': 'muslim'}

{'count': 6, '\_id': 'unitarian\_universalist'}

{'count': 3, '\_id': 'sikh'}

{'count': 2, '\_id': 'bahai'}

{'count': 2, '\_id': 'eckankar'}

{'count': 2, '\_id': 'spiritualist'}

1. Top 10 Dining Choices

dinning\_result = db.seattle\_osm.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"restaurant"}},

{"$group":{"\_id":"$cuisine", "count":{"$sum":1}}},

{"$sort":{"count":-1}}, {"$limit":10}])

for d in dinning\_result:

print(d)

The following query result shows the top dining choices contained in Seattle OSM.

{'count': 817, '\_id': None}

{'count': 265, '\_id': 'mexican'}

{'count': 257, '\_id': 'pizza'}

{'count': 249, '\_id': 'american'}

{'count': 162, '\_id': 'asian'}

{'count': 155, '\_id': 'thai'}

{'count': 150, '\_id': 'chinese'}

{'count': 123, '\_id': 'japanese'}

{'count': 117, '\_id': 'italian'}

{'count': 101, '\_id': 'burger'}

1. Top 10 Types of Coffee Shops

cafe\_result = db.seattle\_osm.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"cafe"}},

{"$group":{"\_id":"$cuisine", "count":{"$sum":1}}},

{"$sort":{"count":-1}}, {"$limit":10}])

for c in cafe\_result:

print(c)

The following query result shows the top dining choices contained in Seattle OSM.

{'count': 663, '\_id': 'coffee\_shop'}

{'count': 567, '\_id': None}

{'count': 48, '\_id': 'ice\_cream'}

{'count': 20, '\_id': 'sandwich'}

{'count': 19, '\_id': 'american'}

{'count': 15, '\_id': 'tea'}

{'count': 14, '\_id': 'donut;coffee\_shop'}

{'count': 10, '\_id': 'vietnamese'}

{'count': 10, '\_id': 'donut'}

{'count': 9, '\_id': 'frozen\_yogurt'}

1. Other Ideas: Number of Distinct Points Contained in Map

Although the above data view showed that there are roughly 8.3 million data points, it does not directly address the question of how many distinct geographical points does Seattle OSM contain. To answer this question, we go about the route of using the geo position of latitude and longitude, together with the code below.

def make\_pipeline():

pipeline = [ ]

group = {'$group':{'\_id':'$pos', 'uniq\_count': { '$sum': 1 }}}

sort = {'$sort':{'count':-1}}

group1 = {'$count':'uniq\_count'}

for e in [group, sort, group1]:

pipeline.append(e)

return pipeline

def aggregate(db, pipeline):

return [doc for doc in db.seattle\_osm.aggregate(pipeline)]

Note that given the size of our returned query result, we have to write the results into a collection, and iterate through each element to get the detailed results contained withitn.

pipeline = make\_pipeline()

pos\_result = db.seattle\_osm.aggregate(pipeline, allowDiskUse=True)

for p in pos\_result:

print(p)

We see that Seattle OSM contained roughly 7.5 million distinct geographical "points".

1. **Other Observed Problems**

I discovered that the Seatlte OSM data had inadvertently contained some points that actually does NOT belong to Seattle at all - in fact, they belong to a neighboring city in a neighboring country, Victoria, Canada. This was probably due to the works for GPS systems that categorized these points based on their vicinity to Seattle. Indeed, Victoria is closer to Seattle than to Vancouver, BC.

not\_result = db.seattle\_osm.aggregate([{"$match":{"address.city":{"$exists":1}}},

{"$group":{"\_id":"$address.city", "count":{"$sum":1}}}, {"$sort":{"count":1}}])

This problem was discovered using this following exploration technique by looking at the name of the cities that were found in the dataset -- sorted in ascending order. While doing this exploration, all of the cities are indeed within Washington State, except for Victoria.

vic\_result = db.seattle\_osm.aggregate([{"$match":{"address.city":{"$eq":"Victoria"}}},

{"$group":{"\_id":"$address.city", "count":{"$sum":1}}}, {"$sort":{"count":1}}])

1. **Conclusions**

After this review, it would appear that the Seattle OSM data set had leveraged some automatic geo-data collection techniques such as GPS devices while a vehicle is in motion. Although some of the data points clearly do not belong to Seattle metro area, I now feel that as a while, the Seattle OSM has been cleaned sufficiently for the purposes of this exercise.

As a further note of possible future research, I find the proper identification of whether a geo- location should belong in the Seattle area can be made better by classification techniques in machine learning, rather than a detailed, hard-coded rule of zip-codes. The argument for machine learning is as follows:

* 1. Washington zip codes all starts with the digit 9.However, other States and areas that also starts with ‘9’ are Alaska, American Samoa, California, Guam, Hawaii, Marshall Islands, Federated States of Micronesia, Northern Mariana Islands, Oregon, Palau, Army Post Office Pacific, and Fleet Post Office Pacific. Given the vast and disperse geographic office, we can imagine a long and complicated if-else statements.
  2. A better approach to allocating an actual Seattle area node is to use longitude-latitude geo-coordinates, which OSM already collects. However, rather than include a node based on the whether a pair of geo-coordinates fall into the map area (again by some if-else rule), probabilistic classification techniques such as logistic regression can be used. Higher probability would translate into higher confidence that a node does belong in the map area.

1. Using machine learning to classify a node using logistic regression can be done relatively quickly. Therefore computational costs is less of a concern than the accuracy, precision and recall rates. There are also other classification methods such as decision tree, SVM, and KMeans with varying levels of computational demands and accuracy rates.
2. Longitude and latitude are the natural features to be used in the machine learning algorithm. Some other candidate can be the postcodes, city name, and nature of amenity. We are expected not to have a large number of features but a large number of training samples, which will aid the training.
3. Using machine learning techniques is congruent with the trend that more and more nodes are being collected and processed by automatic methods such as Apps and GPS that communicate its position to the server on a real-time or near real-time basis. Having an efficient yet accurate approach to process and input the node to OSM is worth considering.