OpenStreetMap Data Wrangling Project

Udacity Data Analyst Nanodegree
Kaleem Zia Khwaja, June 1 2015

Map: Portland Metropolitan Area, USA
https://www.openstreetmap.org/relation/186579
https://mapzen.com/metro-extracts/

Introduction

I decided to investigate and clean up a map of the city of Portland. The area was too big to download directly from openstreetmaps.org, but was available from mapzen.com. I realized later during my analysis that this area labeled as simply "Portland" on mapzen was in fact the entire Portland metropolitan area, home to approximately 2.3 million people spread over 6,684 square miles; this was fine as the file was still small enough to work with within reasonable time limits.

Dataset Overview

The dataset took up 632MB in uncompressed osm format, and later ~2 GB as a mongodb database. Given the size, I developed my python data-cleaning script on a 1% subsample using a modified version of the script provided in the project assignment (I included this script in the project folder as: 'makeSmallerOSM.py'). I changed the script to return a random sample of elements rather than every 10th, with the sample size specified as a command line argument. I scaled up to the full dataset once everything was running smoothly.

I started by computing several file statistics in python to get acquainted with the data: The full dataset contained a total of 7,986,818 tags, of which 3,005,298 were nodes or ways. There were 1024 unique contributors to the map. No osm keys had problematic characters that would preclude their use as dictionary keys in python, making data structure manipulation pretty straightforward.

Cleaning

I chose to clean up the dataset in two ways before converting osm to json for loading into mongodb:

First, I cleaned up the zip codes, which were the values under keys "addr:postcode" within nodes.

The city of Portland proper has only 24 zip codes, but my map returned 121 different zip codes (ignoring 4-digit zip extensions). After mapping a couple dozen of these by hand, I noticed they were generally all within the greater Portland Metropolitan area, which encompasses the nearby urban areas of Beaverton, Hillsboro and Vancouver, Washington just north across the Columbia River. A couple zip codes were outside the Portland metro area, but still within Northern Oregon or Southern Washington, so I decided not to exclude any data based on location.

One zip I found, 98214, was not an actual US code, so I decided it worthwhile to do an additional cleaning step by loading in a csv file of valid Oregon and Washington zip codes as a reference set. I checked each zip code in my map for validity against this set of zips, and only output the zip code field to json if valid. In the end, only 8 unique invalid zip codes existed in the data. 6 of these entries were comprised of other address information with or without a valid zip code, such as 'OR 97006' and 'NE 37th Ave'. The remaining 2 invalid entries were non-existent 5-digit zips: 97003 and the aforementioned 98214. I removed the 98214 entries (kept the nodes but removed the zip code field), but there were so many 97003s in the dataset that I had to do some double-checking. It turned out my csv file of Oregon zips was from 2012, and 97003 was added to a Portland suburb in 2014. I added 97003 to my zip code csv by hand and re-ran my code.

To deal with the cases where a postal code entry included a valid code plus additional address info, I built a regex to search invalid entries for 5-digit numbers or 5+4-digit numbers and extract these zip codes without additional state, city or street information attached. Here's some of the command line output from this function for example:

```
invalid zip codes in my map:

'98214', 'Milwaukie, OR 97267', 'NE 37th Ave', 'OR', 'OR 97006', 'OR 97214', 'Portland, OR 97209'

running regex on zip code: Milwaukie, OR 97267

Milwaukie, OR 97267 ===> 97267

running regex on zip code: NE 37th Ave

no match for: NE 37th Ave
```

After cleaning up the zip codes, I standardized street name endings, such as converting 'Blvd' to 'Boulevard'. I created a mapping of 16 different street names, which I built by reading all the street name endings that were not included in a list of 14 common and expected names. I did not change the names of highways ending in numbers, as in "State road 240". The cleaning up of street names was pretty straightforward.

After these two cleanup steps, I converted the osm data to a list of dictionaries and wrote it to a json file. I included only elements tagged as “node” or “way”, ignoring relations. The dataset was then ready to load into mongodb.

Analysis

Once loaded into mongoDB, I was able to investigate the data further:

```
> db.pdx.find().count()
3005298
```

Sanity check. This is exactly the sum of the number of nodes and ways we had in our original osm file, meaning we didn’t lose any node or way entries along the way.

```
> db.pdx.distinct("created.uid").length
1012
```

This is the number of unique users: it is 12 shy of the 1024 we had in our osm file, but we left a lot of data behind by selecting only nodes and ways. I’m actually surprised we still have 1012 contributors after cutting our elements down by almost two thirds.

```
> db.pdx.aggregate([{"$group":{"_id":"$created.user", "count":{"$sum":1}}},
{"$group":{"_id":"$count","num_users":{"$sum":1}}}, {"$sort":{"_id":1}}, {"$limit":1}])

{ "_id" : 1, "num_users" : 205 }
```

This is the number of users with only one entry in the database. Most people add multiple entries to the map.

The four top posters can be seen here:

```
> db.pdx.aggregate([{"$group":{"_id":"$created.user", "count":{"$sum":1}}}, {"$sort":{"count”:-
1}},{"$limit”:4}])

{ "_id" : "Mele Sax-Barnett", "count" : 631889 }
{ "_id" : "Peter Dobratz_pdxbuildings", "count" : 603543 }
{ "_id" : "Grant Humphries", "count" : 332514 }
{ "_id" : "Darrell_pdxbuildings", "count" : 325209 }
```

By their usernames, it looks like contributors 2 and 4 in the list might work for some city buildings department.

There were a very few entries which, despite being tagged node or way in the osm file, had a type field that was not node or way:

```
> db.pdx2.aggregate([{"$match":{"type":{"$exists":1}}}, { $group : {_id: "$type", count: {$sum: 1}}}, {$sort: {count: -1}}])

{ "_id" : "node", "count" : 2701996 }
{ "_id" : "way", "count" : 303273 }
{ "_id" : "public", "count" : 10 }
{ "_id" : "broad_leaved", "count" : 6 }
{ "_id" : "Public", "count" : 6 }
{ "_id" : "audio", "count" : 4 }
{ "_id" : "video", "count" : 2 }
{ "_id" : "conifer", "count" : 1 }
```

6 entries were of type “broad-leaved” and one was of type “conifer”. Perhaps an amateur botanist decided to start marking trees? I looked up these entries with the following command:

```
>db.pdx2.find({ $or: [{"type": "conifer”}, {“type”: “broad_leaved"}]})

{ "_id" : ObjectId("556bc2ca5e9269db1e6aa398"), "pos" : [ 45.4616392, -122.6880805 ], "created" : { "user" : "mz_thump", "version" : "2", "uid" : "182554", "timestamp" : "2015-01-20T23:35:56Z", "changeset" : "28296697" }, "type" : "broad_leaved", "id" : "3254620815", "name" : "Cherry" }
…
```

I then entered the gps positions into openstreetmaps and saw that the seven trees were on two different residential lots in Portland. Using google street view, I was able to confirm the existence and proper identification of 6 of the 7 trees (tree types were identified via the “name” field). The dogwood in particular was quite an impressive specimen.
Moving on:

```
> db.pdx2.find({"pos":{"$exists":1}}).count()
2702015
```

2.7 million entries are geo-tagged.

```
> db.pdx2.find({"pos":{"$exists":0}, "type": "node"}).count()
0
```

All entries of type “node" are geotagged.

```
> db.pdx2.find({"pos":{"$exists”:1}, "type": "node"}).count()
2701996
```

Only 19 geotagged locations are not of type “node”. What type are those 19 items?

```
>db.pdx2.find({"pos":{"$exists":1}, "type": {"$ne": "node"}})
```

I found that 7 of the 19 were our beloved trees already mentioned, 11 were associated with an airport and labeled as type “public”, and the last was type “video” with name “FFAKE llc”. I suppose an open source map is bound to have a few mis-labeled fields here and there. All told, I think 19 “type”-field mistakes out of 2.7 million is pretty impressive.

```
> db.pdx.find({"address":{"$exists":1}}).count()
112777
> db.pdx.find({"address.housenumber":{"$exists":1}}).count()
107215
> db.pdx.find({"address.street":{"$exists":1}}).count()
107319
> db.pdx.find({"address.postcode":{"$exists":1}}).count()
107476
```

Only a few percent of the 2.7 million nodes have addresses, but of those most have complete address information.

Which amenities are most commonly added to the map?

```
> db.pdx2.aggregate([{"$match":{"amenity":{"$exists":1}}}, { $group : {_id: "$amenity", count: {$sum: 1}}}, {$sort: {count: -1}}])
{ "_id" : "parking", "count" : 2362 }
{ "_id" : "place_of_worship", "count" : 1174 }
{ "_id" : "restaurant", "count" : 1033 }
{ "_id" : "school", "count" : 913 }
{ "_id" : "fast_food", "count" : 504 }
{ "_id" : "cafe", "count" : 376 }  …
```

Parking, apparently, followed by places of worship, restaurants, and schools. Grouping by cuisine type, we can get a sense of what kinds of food are most common in Portland:

```
> db.pdx2.aggregate([{"$match":{"amenity":{"$exists":1}}}, { $group : {_id: "$cuisine", count: {$sum:
1}}}, {$sort: {count: -1}}])
{ "_id" : null, "count" : 9061 }
{ "_id" : "burger", "count" : 173 }
{ "_id" : "mexican", "count" : 153 }
{ "_id" : "coffee_shop", "count" : 139 }
{ "_id" : "pizza", "count" : 139 }
{ "_id" : "american", "count" : 84 }
{ "_id" : "sandwich", "count" : 81 }
{ "_id" : "chinese", "count" : 58 }
{ "_id" : "thai", "count" : 51 } …
```

It looks like burgers win out, followed by Mexican, coffee, and pizza.

Ideas for improvement

For an area the size of greater Portland, relatively few amenities are marked in openstreetmaps: we found 1033 entries marked as ‘ “amenity”: “restaurant” ’ in our map data, while a search for Portland restaurants at yelp.com returns 6555 results. Yelp has a free open API with a daily call limit of 25,000, and increased call limits upon request. I checked the format of Yelp API responses using jsonlint.com, and responses are valid json, so they would be easy to parse. I also looked at the API documentation and saw that searches can be made using latitude/longitude bounding boxes. It would be straightforward to write a script that performed a gridwise search of yelp entries covering the entire Portland area, perhaps iterating through a list of business categories in each cell of the grid, making an API call on each business type in each area.

Results could then be converted to osm format and added to openstreetmaps using their editing API. I imagine the most difficult part of this process would be deciding which Yelp entries were valid/current/otherwise worthy of inclusion in the map. One of the fields returned by Yelp is a boolean ‘is_closed’, helpful because it would allow us to easily skip over establishments no longer in business. Another potential issue would be double-marking of a business already on the map. This process would undoubtedly present challenges in these kinds of fine details, but it also has potential to increase the number of amenity entries in the map several-fold.