Patrick Coakley
Project 3

Open Street Map Project

Background

In this this project I have chosen to use the San Francisco metropolitan area in Open Street Maps provided by Mapzen. San Francisco is somewhere I've always been curious about from a geographical standpoint, as it has a very distinct terrain setting, as well as from a personal interest standpoint, since it is home to some of the most important businesses in technology. The information used for this project was downloaded using a custom extract using the search **"City of San Francisco, CA, USA"** on MapZen's custom extract tool as there was not one of just San Francisco the city: (**https://s3.amazonaws.com/mapzen.odes/ex_6iHAtrk1kTEfcBGjaopdANtKQ2URZ.osm.bz2**).

Initial Thoughts

In order to get an idea of what I should be checking against before cleaning my data, I used the terminal to poke around my data using **cat, less, grep,** etc. This allowed me to see the **"addr:"** tags without other information getting in the way, making it easier to see what kinds of things to look out for. I also used Python to make a simple script that let me query and organize the information similar to one of the exercises in this course.

One of my initial ideas was to check the zip codes against the official zip codes in the area. I utilized the USPS website and got a copy of all the valid zip codes in San Francisco. I put them in a blank file and ran a regex command (**cat zipcodes | grep -o -E '[0-9]{5}' > fixedzipcodes | sed 's/\(.\{5\}\)/\1,/g' fixedzipcodes | tr -d '\n' > fixedzipcodes**) using cat, grep, sed, and tr to clean out the unnecessary data. Using this list I can check against the output I get from probing the data. In the end the zip codes were ostensibly fine and this was not necessary, but I was able to use it to notice a discrepancy with the cities in the map data in the next section.

Problems In The Data

The three main things I checked for in my audit were the consistency of the street and state values, as well as the validity of the zip codes. I used the regex example from the sample code in our course that covered the street name endings, and adjusted it to allow for zip codes and states by using the wildcard regex expression. This is because these two values are singular and don't require parsing through spaces, unlike the streets. I also used UNIX command line tools such as **grep**, **cat**, and **less**, as they make it easy to find little things while tweaking the Python code.

The two main issues I found for the data were:

- Some of the streets ended with things like numbers or single letters, and there are inconsistencies with things like 'St' instead of 'Street', or lowercase variations such as 'street'
- There were some outliers with regards to states; the vast majority were using the 'CA' designation, but there were a few that were the full name of the state, 'California', as well as other invalid entries such as 'US'
- There were invalid zip codes, including out of area ones such as San Jose or Sausalito

Some differences in data were also cultural. For example, in San Francisco there is an area called The Embarcadero, but that is also what the main street in that area is called the same thing, so it does not have a common street ending. I ended up hardcoding all of the **"states"** entries to be written as "CA" because it was already more than 99% of the other entries and it is easier to make it a one-way operating than to check every entry. For the zip codes, originally I tried doing strict evaluation using **(new_zipcode = re.findall(r'([9][4][1][0-9][0-9])$',zipcode)**, since San Francisco zipcodes are 941XX. However, the issue with this is there will be entries that would have no zip code but would still be included, so it's probably better to keep the out of area zip codes but filter out the invalid ones with **(new_zipcode = re.findall(r'(\d{5})$',zipcode)**.

Overview of the Data

Using **mongoimport**, I imported the resulting JSON file into mongodb. Then I was able to run the following queries on the data:

- **db.map.find({"type": "node"}).count()** lets us know that there are **1671426** nodes in the collection, and **db.map.find({"type": "way"}).count()** shows **175121** ways
- Similarly, running **db.map.find({"amenity": "cafe"}).count()** shows **527** cafes in the city, and changing it to show all the Starbucks shows **52** in the city
- Using **db.map.totalSize()** gives the total size of the collection, which is **150241280 bytes**, which is roughly **150MB**; this is interesting because the resulting JSON file was much bigger at around **515MB**
- I found that there were **1222** unique users in this collection using **db.map.distinct("created.user").length**
- Using **db.map.find({"address.street": "The Embarcadero"}).count()** I found that there were **15** unique entries listed as being on the street called The Embarcadero
- **db.map.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"restaurant"}}, {"$group":{"_id":"$cuisine", "count":{"$sum":1}}},{"$sort":{"count":-1}}])** was used to

find the top cuisines in **“amenity”:”restaurant”**; the top 5 results (not including **“null”** are **Chinese** with **91, Mexican** with **88**, **Italian** with **79**, **Japanese** with **77**, and **pizza** with **68**

- Using a similar query as above, **db.map.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"fast_food"}}, {"$group":{"_id":"$cuisine", "count":{"$sum":1}}},{"$sort":{"count":-1}}])** shows the top fast-food cuisines in the city, which has slightly different but similar results, though this time **burger** restaurants have the most entries at **37**
- **db.map.aggregate([{"$group":{"_id":"$created.user", "count":{"$sum":1}}}, {"$sort":{"count":-1}}, {"$limit":1}])** gives us the number one contributing user, **"ediyes"**, who shows up in **674923** entries

Ways To Improve The Data

Using the **“amenity”** tag to find places that serve food can be confusing due to the various values you could be using, and I think to improve this there should be a boolean meta-tag on the top level to give an idea if something is categorized as serving food or drink. This would help when searching for specific things, as if you were to search for **“pub”** using **db.map.find({"amenity": "pub"}).count()** and not **“bar”** you would get different results (**156 pubs** and **217 bars**), but ostensibly they are the same thing to a user who is looking for a place to drink an alcoholic beverage.

Weakening the distinction between the two by having them categorically listed in **“alcohol”: “True”** or something similar would make it a bit easier for someone who just wants to find a drink. Another example of this problem is **“fast_food”**, which is another **“amenity”** tag value that would better fit in a child tag under restaurants. Why is **“fast_food” (db.map.find({"amenity": "fast_food"}).count()** shows **219** results**)** under **“amenity”,** but **“pizzeria”** or **“pizza”** are not (neither had results using a similar query)? Instead **“pizza”** is listed under the top-level **“cuisine”** tag, which confuses things even further. Semantically organizing every core tag would make the service much easier to use because it would allow you to drop down one level granularly instead of having liberal use of the current tags.

One other example of the disorganization being an issue is the fact that not every restaurant has a **“cuisine”** value, so if you use **db.map.aggregate([{$match:{"amenity":{$exists:1},"amenity":"restaurant"}},{$group:{"_id":"$cuisine", "count":{$sum:1}}},{$sort :{"count": -1 }}, {$limit:11}])**, which sorts all of the restaurants by cuisine frequency, the top results are null. Since there’s no requirement to use the **“cuisine”** tag it can make it very difficult to find exactly how many entries there are of a certain thing. I think the best solution is to have more regulation on the tags, and as noted before, create a strong list of core tags that have to be filled out before data is committed to the mapset.

This might be a Herculean effort on the part of the OpenStreetMap organizers, but it will make the end result much easier to work with for outside parties, such as Apple, who uses this data in their Apple Maps program. By having a more constrained format for tagging information, information purity can be maintained without having a lot of

anomalous information. Maintaining the core mapping information inside of the top-level tags and allowing a sub-category of **"custom-tags"** could be an easy alternative.

On the other hand, the point of OpenStreetMap is to be an open community for any and everyone to work on whatever they choose, so it might go against the spirit of the goals and ideology of the organization to limit the ways people can contribute. In fact, their license mandates that any changes made have to be shared publicly. I feel that when a project of this size is not strongly opinionated, it ends up having quality control issues, and this is why I think that having a core tag set to work with would simply make it easier for newcomers to contribute, as potentially in this setup there would be tag guidelines and style guides to keep the focus on improving the fundamental information of the maps.

Other Thoughts on the Data

Overall, working with this dataset I was surprised with the naming conventions in San Francisco compared to my hometown, as they tend to be more liberal in naming roads and streets instead of always using the traditional street types. The Embarcadero being both a street and location was kind of confusing to me at first, and I thought there were multiple errors with the information, but going back and checking it for myself made it clearer.

My experience with MongoDB was interesting, and I think NoSQL databases are quite useful for certain kinds of data based on this experience. In comparison to my experience with SQL, I found MongoDB to be much easier to setup, import, and query with, and in general working with JSON is much cleaner if you're familiar with it already from using JavaScript. For all of the querying I used **mongoshell** because I prefer REPLs and shells for exploration, and I found it to be quite easy to work with. I would like to incorporate it in future projects, as well as other NoSQL databases such as **Dynamo** or **CouchDB**.

Files

sample.osm: 8.9MB
sample.osm.json: 10.1MB
sanfrancisco.osm: 353.7MB
sanfrancisco.osm.json: 515.9MB

Bibliography

MongoDB. (n.d.). *The MongoDB 3.4 Manual*. Retrieved from MongoDB: https://docs.mongodb.com/manual/

OpenStreetMap. (n.d.). *OpenStreetMap*. Retrieved from OpenStreetMap Wiki: http://wiki.openstreetmap.org/wiki/Main_Page

Python. (n.d.). *Python 2.7.13 documentation*. Retrieved from Python: https://docs.python.org/2/

Udacity. (n.d.). *DAND P3 | Data Wrangling*. Retrieved from Udacity Discussion Forums: https://discussions.udacity.com/c/nd002-p3-data-wrangling