David Pankiewicz
dwpankiewicz@gmail.com
Data Analyst Nanodegree
5/28/15

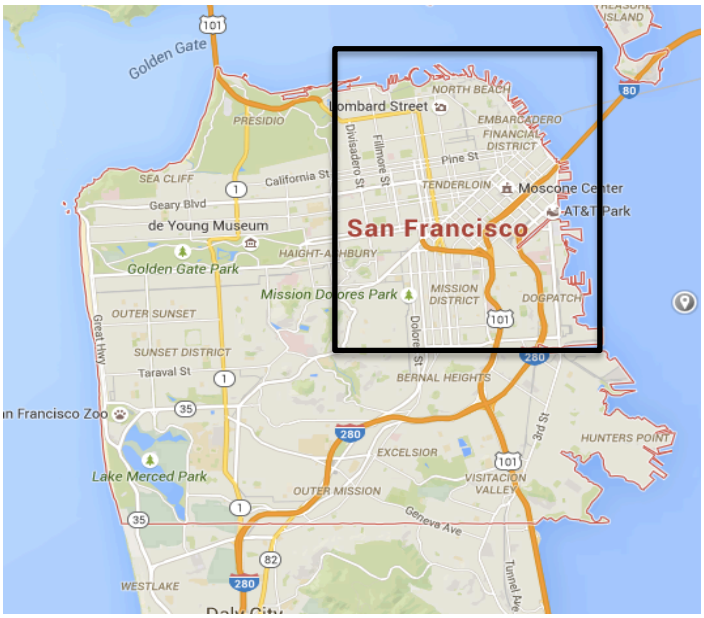
OpenstreetMaps Project
San Francisco, CA

1. The Area

I chose to work with a subset of data on the city of San Francisco. Initially, I attempted to work with the full dataset of the city, but at approximately 640 MB, the file was too large for my laptop to run the implementation of my analysis. Instead, I decided to work with a set of data that's roughly depicted in the black box on the map of San Francisco below.

The exact call to the Overpass API is as follows:

```
(node(37.746299, -122.457169, 37.814679,-122.378548);<;);out meta;
```



David Pankiewicz
dwpankiewicz@gmail.com
Data Analyst Nanodegree
5/28/15

2. Problems with the Data and Cleaning

General Data Sprawl

The first challenge in analyzing this dataset was understanding its composition. In order to get a finger tip feel for the data, I wrote some functions in `audit.py` that create a dictionary of the categories and the distinct values that appear in the dataset. This was then sent to a text file for easy viewing ("tag_key_value.txt").

Screenshot of "tag_key_value.txt"

```
KEY: map_size   VALUES: ['site']
KEY: map_type   VALUES: ['street']
KEY: material   VALUES: ['wood']
KEY: maxheight  VALUES: ['12\'6"', '14\' 0"', '5']
KEY: maxlength  VALUES: ['12 feet', '50 feet']
KEY: maxspeed   VALUES: ['30 mph', '15', '40 mph', '40mph', '5 mph', '20 mph', '35 mph', '25 mph',
'50 mph', '30mph', '45 mph', '15 mph']
KEY: maxweight  VALUES: ['3']
KEY: maxwidth   VALUES: ['3']
KEY: microbrewery       VALUES: ['yes', 'no']
KEY: monitoring:bicycle VALUES: ['yes']
KEY: mooring    VALUES: ['no']
KEY: motor_vehicle      VALUES: ['forestry', 'yes', 'private', 'no']
KEY: motorcar   VALUES: ['yes', 'no']
KEY: motorcycle VALUES: ['yes', 'designated', 'no']
KEY: muni_route_ref     VALUES: ['10']
KEY: myspace    VALUES: ['toronadosf']
KEY: name       VALUES: ['Iglesia Del Pacto Evangelico', 'PacBell Building', 'The Mix', 'Strauss
Playground', 'Tsar Nicoulai Caviar Cafe', 'Phelan Hall', 'McEvoy Ranch Olive Oil', 'Indiana Street',
'Islamic Society of San Francisco', 'Samasource', 'TransAmerica', 'First Covenant Church', 'Genray
Hair Salon', 'Nob Hill Motor Inn', 'Meacham Place', 'Sushi Delight', "Suites at Fisherman's Wharf",
'Embarcadero North Street', 'corner store', "Tarantino's", 'Hong Kong Clay Pot Restaurant', 'Marriott
Marquis', 'Woods Lot', 'Edward Street', 'Lush Lounge', 'Berry & 7th', 'BevMo!', 'Tailored Salon',
'DSW Shoe Warehouse', 'Mission Plaza Cleaners', 'Alamo Square Cafe', '211 Gough Street Professional
Building', 'Union Street Apothecary', 'Nordstrom Rack', '24th & Valencia (Union 76)', 'Guy Place',
'Velvet da Vinci', 'Noe Valley Nails', 'Margaret S. Hayward Playground', 'AT&T', 'Army Veritable
Vegetable', 'Saint Edward the Confessor Roman Catholic Church', 'Fire Station Number Two',
'Transamerica Pyramid', 'Castle Street', 'Safeway 2606', 'Nike Store', 'Flax Art Supply', 'Fortuna
Avenue', '3rd Street Grill', 'Hyde Street Pier', 'child care center- Lutheran Church', 'A.P. Giannini
Plaza', 'Digital Media Building', 'Wallgreens', 'Travelodge at the Presidio', 'SF County Jail', 'Osha
```

After doing this, I had a better sense of the underlying structures of the data as well as some starting points for where and how I might go about cleaning any dirty data.

Implicit Sub Tags

There were many categories that were comprised of a category and a subcategory formatted with a colon in between (e.g. "diet:vegan", "diet: vegetarian").
Given the standard form, it was easy to programmatically create a dictionary as the entry for a category and use the subcategories as keys. For example:

Diet: {
    Vegan: Yes
    Vegetarian: Yes
    }

"City"

```
KEY: addr:city  VALUES: ['San Francisco, CA 94102', 'San Francscio', 'San Francicsco', 'san
Francisco', 'SAN FRANCISCO', 'San Francisco', 'san francisco', 'San Francisco, CA']
```

All entries were clearly San Francisco. However, entries varied in capitalization, included the state and zip code, or included typos. Standardizing all "city" entries to "San Francisco" was a simple fix.

David Pankiewicz
dwpankiewicz@gmail.com
Data Analyst Nanodegree
5/28/15

"Route_ref"

```
KEY: route_ref  VALUES: ['1;2', '5L', '14;14L;49;67', '5;6', '41;80X;82X', '24', '28;76;91 Owl',
'14;14L;14X;N Owl', '27', '21', '22', '14;14L;14X', '47;49;82x;90 Owl;91 Owl', '47;49;90 Owl', '9AX;
9BX;9X;30;45;79X;91 Owl', '1; 41', '82X;108', '27;38;38L', '6;7;9;71;71L;F;L Owl;N Owl', '22;53',
```

Route_ref holds data on which bus and trolley lines are associated with nodes and ways. While some entries contained only one item ("42 Owl"), others held multiple entries separated by semicolons. ("35; 83;92"). To solve this, cleaning was done programmatically by splitting the entries on semicolons and then storing the results in a list.

"Street Names"

Street names were either messy because 1) they lacked a common form of the type of street (e.g., Van Ness Ave., Van Ness Avenue), or 2) they lacked a type of street completely (e.g. Van Ness). While the first problem type is easy to map to Avenue, the second type of entry is difficult to figure out unless you can match the full street name elsewhere in your data. I chose to map these cases manually in a dictionary because the majority comprised of a few major streets in San Francisco. In this particular case, a manual solution was simpler than a programmatic one given the small size and scope of the cleaning

"Amenity" and "Cuisine"

While the entries to amenity data are clean, the format may be difficult to query. Entries longer than a word have an underscore between them (e.g. "bus_station"). To make querying intuitive, I replaced underscores with spaces. I performed the same actions for cuisine data.

3. Data Overview

File Sizes

large-sf-square.osm.json 144.5 MB
large-sf-square.osm 129 MB

Number of documents: 665,127

```
db.sanfrancisco.find().count()
```

Number of nodes: 604,620

```
db.sanfrancisco.find({"type":"node"}).count()
```

Number of Ways: 60,430

```
db.sanfrancisco.find({"type":"way"}).count()
```

Number of Unique Users: 734

```
{"$group":{"_id": "$created.user"}},
{"$group": {"_id": "Unique users", "count": {"$sum":1}}}
```

Accepts Bitcoin as payment : yes: 77, no:4

```
{"$match": {"payment": {"$exists":1}}},
{"$group": {"_id": "$payment.bitcoin", "count": {"$sum":1}}}
```

Num of Café's: 264

```
{"$match":{"amenity": "cafe"}},
{"$group": {"_id": "Num. of Cafe", "count": {"$sum":1}}}
```

Top Cuisines:

```
{u'_id': u'mexican', u'count': 67}
{u'_id': u'coffee shop', u'count': 66}
{u'_id': u'italian', u'count': 47}
```

Anecdotally, this makes sense. The Mission district has a ton of Mexican restaurants, San Francisco is known for its coffee culture, and the North Beach neighborhood is San Francisco's Little Italy and is fairly prominent.

```
{"$match":{"cuisine": {"$exists":1}}},
{"$group": {"_id": "$cuisine", "count": {"$sum":1}}},
{"$sort": {"count":-1}},
{"$limit":3}
```

4. Future Work, Analyzation, and Use Cases

A potential future improvement is to decide what specific categories to include an entries in the database. This can be done manually or programmatically.

In using a manual solution, one would only include data that is relevant to the purpose of the database. For example, if this data were to be used to power a "Yelp"-like application, categories like "address", "cuisine", and "URL" are relevant, but others like "ship:type" don't make sense to include.

One implementation of a programmatic solution would be to set a threshold for the number of appearances of a category to merit inclusion in the database. For example, the category "name" appears on hundred of documents and would be highly likely to be included. On the other hand, the category "aeroway" appears only on one document in the entire dataset. Thus, it may not need inclusion because it gives us very little descriptive information about the dataset as a whole. By deciding category inclusion in this matter, we can ensure that any categories in our data are robust enough to support analysis representative of the whole dataset.

Final Thoughts

Most of the data itself is clean and required little to no munging, which was somewhat surprising. Creating commom structures was critical in organizing and standardizing the structure and made analysis easier. What was more challenging and lacks a clear solution is data that is incomplete or missing. While this is not an explicit problem in the dataset (i.e., you don't have to work with "NULL" values), it can lead to underestimation in the values in aggregate analyses. For example, I suspect that more than 77 shops accept bitcoin in San Francisco, but only 81 have been tagged regarding their policy on bitcoing. If this dataset becomes more complete, it becomes possible to start tracking trends and changes over time (e.g., bitcoin adoption, the average number of levels in a building as SF begins to build up, etc.). With complete data, this map becomes a powerful and immensely useful tool.

Appendix: Python File Descriptions

`data_audit.py`
Takes in a raw .osm file, parses it, and prints various outputs that can serve as strating points to cleaning and munging data. Additionally, a text file is created that contains all key values pairs that appear in the tag elements.

`clean.py`
Cleans various predefined categories of data. Clean.py is imported into create_json.py and called from there.

`create_json.py`
Create_json takes in the raw .osm file, parses it, cleans it as necessary with clean.py, and creates the final json file for import into a Monogo database.

`insert_into_mongo.py`
Inserts a given json file into a Mongo Database

`pipeline_sfmap.py`
Runs queries against a Mongo Database using an aggregation framework.