Data Analyst Nano Degree P3 Project

Open Street Map Project

Part 1 - Map Selection

For my open street map project, I selected San Diego, California. This is the city where I grew up and it is the location I feel most comfortable exploring. I procured my .osm data from MapZen using their prepared San Diego Metro Area extract.

- San Diego Metro Extract (https://mapzen.com/data/metro-extracts/metro/san-diego_california/)

Part 2 - Full file overview

The first thing I noticed, even before examining the data within the file was the size of the file. The unzipped OSM XML file was just over 302 MB! That is a lot of elements, tags and children! With that in mind, I wanted to get a better idea of what that meant in terms of data within the file. So the first thing I did was count the top level tags.

Note All the auditing and cleaning function code is included in the Python_Auditing_Code.py file included in the submission. I have kept out the actual code for length purposes.

Counting top level tags in the full San Diego OSM file

{'bounds': 1, 'member': 13328, 'nd': 824464, 'node': 1033820, 'osm': 1, 'relation': 743, 'tag': 2638620, 'way': 93571}

As you can see from the output, there are a huge amount of file elements. There are over 1 million *node* elements, over 93,000 *way* elements and over 2.6 million *tag* element children.

Since the main focus of the project is auditing the 'k' element attributes, I also wanted to get a general scope of their size with the full OSM file. To do this I used code from the Open Street Map Case Study to count tag 'k' values and seperate them into **lower** (those with just lower case letters in the string), **lower colon** (those with lower case letters and a colon, which usually indicated multiple sub-categories, like addr:street), **problem characters** (those with non-alphanumeric characters) and **other** (everything else).

Count all tag 'k' values that fall into the lower, lower_colon and problem character categries.

{'lower': 646269, 'lower_colon': 1954893, 'other': 37453, 'problemchars': 5}

The code further illustrates the size and scope of the full data file with approximately 2.63 million 'k' tags. If I have any hope of accurately understanding and auditing the data, I will need to utilze a much smaller sample.

Part 3 - Creating a smaller sample

To get a smaller file to work with I used the code provided in the Open Street Map Project overview section. In order to ensure there were no major surprises when I did am audit on the full file, I made my sample file take the every 2nd 'k' top level tage element.

Part 4 - Auditing the sample file

Now that I have a smaller sample to work with, I set off getting a better understanding of the actual data. While the Even with a smaller version of the data, I still needed a better understanding of the data content. After reviewing the OSM XML documentation (https://wiki.openstreetmap.org/wiki/OSM_XML) I understood the bulk of the "content" I needed to explore was in the second level attributes of the *node* and *way* elements.

My initial auditing code pulled out the secondary attributes 'k' and 'v' attributes within the element child 'tag'. The print function presents these in a more readable fashion than simply using print or pprint.

I would also like to point out that initiall the extraction code was written using iterparse (as shown in the OSM Case Study). Even with a sample file, this method was nearly crashing my computer. So after consulting the Udacity forum, it was brout to my attention that using enumerate with the get element function would be a more efficient amethod to pull this data from the file.

While these reviewing functions, definately helped me get a better understanding of the types and varieties of 'k' and 'v' values with the OSM file, I was still having a hard time seeing each 'k' attribute and the 'v' values connected. Therefore, I needed a function that not only pulled all 'k' and 'v' child attributes, but also linked the two.

In [ ]:
```
'''Function creates dictionary for the node tags with a set 'k' attribut
es and a set 'v'
values for the associated 'k' attribute.
'''
def tm_unique_k_and_v(osmfile, tagtype):
    osm_file = open(osmfile, "r")
    unique_k_and_v = defaultdict(set)
    for i, element in enumerate(get_element(osm_file)):
        if element.tag == tagtype:
            for tag in element.iter("tag"):
                unique_k_and_v[tag.attrib['k']].add(tag.attrib['v'])
    return unique_k_and_v

print "Unique 'k' and 'v' values for node element"
tm_unique_k_and_v("sample_SD_2.osm", "node")
```

Now I have some information I can work with! This breakdown of the 'k' attributes and associated values really helped me get a better understanding of the data and what elements needed fixing.

Using these outputs, I went through all the information and identified five 'k' attributes that needed fixing and would be helpful in my SQL queries later. Obviously this is not an exhaustive list of problem areas, as there are **A LOT** of areas for auditing and cleaning, but these were the ones with obvious problems that seemed reasonable to fix with the skills provided to date.

Part 5 - Identifying and fixing data problems

As mentioned previously, I identified five 'k' attributes that had inconsistent or problematic 'v' values that would cause problems in an SQL query. The five 'k' attributes are:

1. Phone numbers
2. Zip Codes
3. Cuisine
4. Denomination
5. Street Types

Phone Numbers

Both the node ("contact:phone" and "phone") and way ("phone") elements has 'k' attributes that containted phone numbers, but a closer examination of these shows a large amount of inconsistency in how the phone numbers are formatted.

Examples of phone numbers encountered
'6193351786',
'+1 619 2934450',
'+1 (858) 514-4500',
'(619) 222-7608',
'858.534.9384'

As you can see there are a variety of formatting structures in the phone data includes the use of parentheses, periods, dashes and '+1'. To fix this problem I first stripped away all non-numerical characters, then spaced the numbers in the standard ### ### #### phone structure.

In [ ]:
```
#clean phone numbers
def clean_phone_nums(phone_number):
    num = re.sub('[^0-9]','', phone_number)
    if num[0] == '1':
        cleaned_number = num[1:4] + " " + num[4:7] + " " + num[7:11]
    else:
        cleaned_number = num[0:3] + " " + num[3:6] + " " + num[6:10]
    return cleaned_number
```

Zip Codes

Just like phone numbers, both node ("addr:postcode") and way elements ("addr:postcode", "post_code") have consistency issues in how zip codes are formatted.

The most common issue here is that sometimes the zip code listed is in the standard 5-digit format (92354) and other times it has the full, 9-digit format (92354-7321). Additionally, there are also instances where non-zip code data is found (i.e. full street addresses). To create uniformity with the zip codes, I simply pulled the first 5 numerical characters from the string, as the additional 4-digits are not standard or required when constructing a full address. The next most common issue was that sometimes there was a California state code (CA) abbreviation at the beginning and the 5-digit zip code at the end. In those cases, I pulled the five numberical numbers from the end of the string. Finally, there is one non-zip code related data value ('Scripps Ranch Blvd.'). Knowing the area, I was able to determine that street is relatively small and only exists in a single zip code (92131), so I simply substituted the street name for the zip code where the street is located.

In [ ]:
```
def clean_zipcode(zip_code):
    if re.search(r'^\d{5}', zip_code):
        new_zip = re.search(r'^\d{5}', zip_code).group()
    elif re.search(r'\d{5}$', zip_code):
        new_zip = re.search(r'\d{5}$', zip_code).group()
    else:
        new_zip = '92131'
    return new_zip
```

Cuisine and Denomination

Both the cuisine and denomination 'k' attributes suffer from similar issues, including non-alphanumeric characters, formatting issues and inconsistent capitalization. To address the cuisine issues, I used a regex .search command to isolate just the cuisine names, leaving out the numbers and problematic characters. I also made all the cuisine names lower case. As for the denominations, the biggest issue was around inconsisten capitalization, so I made everything lowercase.

In [ ]:
```
#clean cuisine
def clean_cuisine(cuisine):
    new_cuisine = re.search(r'\b\w+', cuisine)
    cuisine = new_cuisine.group()
    return cuisine.lower()

#clean denomination
def clean_denomination(denomination):
    return denomination.lower()
```

Street Types

My final piece of auditing and cleaning is around street types. As illustrated in the OSM Case Study, there are a lot of inconsistencies in how a stree type is entered in OSM data. This same issue is also present in the San Dieog Metro OSM data.

I fixed the inconsistencies in the stree types the same way as demonstrated in the OSM Case study, by creating a dictionary of alternative street type names ('mapping') and updating the street types if those abbrevations are found in the key of the alternative street type names dictionary. I developed my "mapping" dictionary by going through the entire list of street types.

In [14]:
```
street_type_re = re.compile(r'\S+\.?$', re.IGNORECASE)
mapping = { "St": "Street",
            "St.": "Street",
            "street": "Street",
            "Ave": "Avenue",
            "Ave.": "Avenue",
            "Av": "Avenue",
            "Rd": "Road",
            "Rd.": "Road",
            "Bl": "Boulevard",
            "Blvd": "Boulevard",
            "Ct": "Court",
            "Ctr": "Court",
            "Dr": "Drive",
            "Ln": "Lane",
            "Loreta": "Loretta",
            "Pk": "Parkway",
            "Pkwy": "Parkway"
            }

def update_street_type(street_name, mapping):
    m = street_type_re.search(street_name)
    if m.group() in mapping.keys():
        street_name = re.sub(m.group(), mapping[m.group()], street_name)
    return street_name
```

Part 6 - Integrating the data cleaning and creating a CSV file

Now that I have my cleaning functions ready, I integrated those into the the OSM Case Study code to prepare the data to be inserted into a SQL database. *See the attached Python+Cleaning+Code.py file for the complete cleaning code*

Part 7 - Creating a SQL Database

After the CSV files were created a created a SQL database in the command line

Part 8 - Data Overview and exploring the OSM database with SQL Queries

Now that I have a working SQL database, I can run some queries to get more information about the OSM data. First and foremost it must be stated that the OSM data for San Diego County is very large, both in file size and geographical data. I know this both from growing up in the area and from some basic queries of the data. This section will begin a data overview and then get into specific, exploritory, queries based on *cuisine* and *denomination*

Data Overview

File Size

san-diego_california.osm - 309.2 MB
sd.db 243.4 MB
nodes.csv - 83.4 MB
nodes_tags.csv - 84.9 MB
ways_nodes.csv 19 MB
ways_tags.csv 20 MB
ways.csv 5.4 MB

Number of Nodes

SELECT COUNT(*) FROM nodes;

1033820

Number of Ways

SELECT COUNT(*) FROM ways;

93571

Number of Unique Users

SELECT COUNT(DISTINCT(e.uid))
FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways) e;

1045

As I mentioned before, the San Diego County area represented in this file is very large. Before exploring the data, I want a little more perpsective on how many cities and zipcodes are represented in this area

Unique Cities and Zipcodes in San Diego County OSM

Unique Cities

SELECT COUNT(distinct tags.value) FROM (SELECT *FROM nodes_tags UNION ALL SELECT* FROM ways_tags) tags WHERE tags.key = 'city'

21

Unique zipcodes

SELECT COUNT(distinct tags.value) FROM (SELECT *FROM nodes_tags UNION ALL SELECT* FROM ways_tags) tags WHERE tags.key = 'postcode'

68

As you can this is a VERY large area, with 21 cities and 68 zipcodes.Now lets look at the the tag counts by cities and zipcodes

Count by City

SELECT tags.value, COUNT*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags WHERE tags.key LIKE '%city' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'San Diego', 187651), (u'Chula Vista', 24070), (u'El Cajon', 19185), (u'La Mesa', 16506), (u'Spring Valley', 14371),

Count by Zipcode

SELECT tags.value, COUNT*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags WHERE tags.key = 'postcode' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'92114', 15167), (u'92117', 14110), (u'91977', 12976), (u'92154', 12825), (u'91941', 11620),

Cuisine and Denomination Exploration

Now that we have a general overview of the size and scope of the San Diego set, it is time to explore certain aspects of the data. Since a key portion of my data cleaning in Python was focused on cuisine, denomination and zipcodes, those will be the focus of my further data exploration.

Top 5 Most Common Cuisines

SELECT tags.value, COUNT*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags WHERE tags.key='cuisine' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'burger', 206), (u'mexican', 134), (u'sandwich', 98), (u'pizza', 78), (u'american', 42),

Now that we see the most common cuisines across all of San Diego, let's look closer at where these diverse cuisines are located within the area. First, let's see what zipcodes have the most cuisine types

Most Cuisine Types by Zipcode

SELECT tags.value, COUNT*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT *FROM ways_tags) tags JOIN (SELECT DISTINCT(id) FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) WHERE key = 'cuisine') as subq ON tags.id = subq.id WHERE tags.key = 'postcode' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'92101', 36), (u'92104', 12), (u'92071', 11), (u'92020', 10), (u'92111', 10)]

The 92101 result makes a lot of sense when you look at an acual map (https://www.google.com/maps/place/San+Diego,+CA+92101/@32.7184743,-117.1986796,14z/data=!3m1!4b1!4 117.1647094) of that part of San Diego and see it encompasses the very popular downtown convention center and Gaslamp Quarter area.

Most Cuisine Types by City

SELECT tags.value, COUNT*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT *FROM ways_tags) tags JOIN (SELECT DISTINCT(id) FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) WHERE key = 'cuisine') as subq ON tags.id = subq.id WHERE tags.key like '%city' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'San Diego', 171), (u'El Cajon', 20), (u'Santee', 15), (u'La Mesa', 10), (u'Chula Vista', 8)]

Now that we know where the most cuisine types are located (zipcodes and cities), let's see what is most popular in those specific areas.

Most popular cuisine in 92101

SELECT tags.value, count*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT *FROM ways_tags) tags JOIN (SELECT DISTINCT(id) FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) WHERE key = 'postcode' and value ='92101') as subq ON tags.id = subq.id WHERE tags.key ='cuisine' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'italian', 9), (u'coffee_shop', 4), (u'mexican', 4), (u'regional', 4), (u'sandwich', 3)

Most popular cuisine in San Diego

SELECT tags.value, count*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT *FROM ways_tags) tags JOIN (SELECT DISTINCT(id) FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) WHERE key LIKE '%city' and value ='San Diego') as subq ON tags.id = subq.id WHERE tags.key ='cuisine' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'italian', 9), (u'coffee_shop', 4), (u'mexican', 4), (u'regional', 4), (u'sandwich', 3)

While 'burgers', 'mexican' and 'sandwhich' are the most popular cuisine types across the entire area, we see that in the city and zipcode with the most cuisine types, the most popular cuisines do not match. Interesting, but not unexpected.

From cuisine, it's time to examine religious denominations.

Most common religious denominations

SELECT tags.value, COUNT*() as count FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags WHERE tags.key='denomination' GROUP BY tags.value ORDER BY count DESC LIMIT 5;

(u'baptist', 120), (u'catholic', 48), (u'lutheran', 42), (u'pentecostal', 36), (u'methodist', 35),

Since San Diego is the largest city in the County, I want to conclude my exploration of the denominatin data by looking at the most common denominations in the city and see how they compare to the County overall.

Most common denominations in San Diego

SELECT tags.value, count*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT *FROM ways_tags) tags JOIN (SELECT DISTINCT(id) FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) WHERE key LIKE '%city' and value ='San Diego') as subq ON tags.id = subq.id WHERE tags.key ='denomination' GROUP BY tags.value ORDER BY num DESC LIMIT 5;

(u'mormon', 6), (u'baptist', 2), (u'nazarene', 2), (u'roman_catholic', 2), (u'anglican', 1)

Very interesting! While the most common denomination overall in the County is baptist, the most common in the City of San Diego is Mormon. Which is not even in the top five for the County. But, when you know something about the area this is not all to surprising, as there is a large mormon population in the City of San Diego.

Finally, lets conclude the database exploration by looking at the most common amenities across the entire county.

Top 10 appearing amenities in San Diego County

SELECT tags.value, COUNT*() as num FROM (SELECT* FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags WHERE tags.key='amenity' GROUP BY tags.value ORDER BY num DESC LIMIT 10;

(u'parking', 2060), (u'place_of_worship', 948), (u'school', 597), (u'fast_food', 577), (u'restaurant', 556), (u'bar', 270), (u'cafe', 175), (u'fuel', 139), (u'toilets', 117), (u'bank', 116)

Final Thoughts and Additional Ideas

Overall, I found the exploration of the OSM San Diego Data very interesting. While it was funt to explore the data, there are still a TON of propblems and inconsistencies with how the OSM data is entered. My cleaning efforts, mearly scrattched the surface of the surface of the 100 of data issues.

In terms of additional ideas to explore in the data, I really wish I had the time to take the nodes_tags and ways_tags data and develop some distinct tables for commercial/business/amenity entities in the area. The fact that all the commercial/business data is spread within all the tag keys and values, and not in distinct tables makes a more detailed and through exploration of the data difficult if not impossible it the current format. If the data was pulled into distinct tables (restuarants, fuel, bank etc.), it would be interesting to see the spread and diversity of business in various zipcodes in the county. This would allow citizens, elected officials, developers etc. to see what types of businesses are needed in certain areas of the city and where certain types of businesses are too common.

Of course this taks comes with a lot of potential problems. First, as we have seen most 'value' tags in the OSM have serious formatting and consistency issues. Another problem is that a lot of the business and commercial entity information is spread across a lot of 'key'tags, both in the nodes and ways elements. Simple information about whether something is a business/commercial building can be found in 'building type', 'designation', 'landuse' (just to name a few) and that says nothing about what type of business or even if its closed or open (its very possible information was entered a while ago and the business closed, but the OSM data has not been updated). Even more troubling is the lack of information on how designation decisions are made and if there is consistency across the different 'key' tags. Needless to say you would need to do a lot of cleaning and make some important judgement issues before a good, reliable database of businesses would be useful for city planning purposes.

In [ ]: