Nanodegree: Data Science

Project #2 – Data Wrangling with MondoDB

Tim Tse

4/15/2015

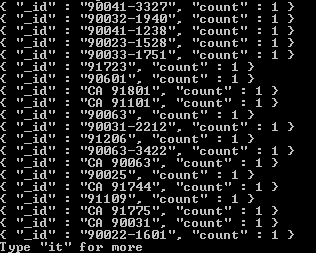
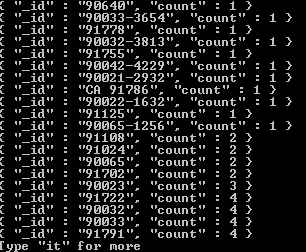
**Project Report**

1. Problems encountered in my map

After downloading the map of the San Garbiel Valley area in California and running it against the data.py file created in lesson 6, I noticed three main problems with the dataset. I will discuss the three problems in this session.

1. **The dataset consists of different format of postal codes.** From the output, we can see there are three different formats of postal codes. They are “CA 91744”, “91744”, and “90033-1751”. We could do auditing to clean up the zip code so the data is uniform.

* db.map.aggregate([{"$match":{"address.postcode":{"$exists":1}}}, {"$group":{"\_id":"$address.postcode", "count":{"$sum":1}}}, {"$sort":{"count":­1}}])



1. **Missing data to perform accurate analysis.** The following code shows the top cuisine in the dataset. We can see that there are 117 data with missing data. There are 25 cuisine entries that are tagged as “Chinese”. However, we cannot conclude that Chinese cuisine is the most popular cuisine due to the high number of missing data.

* db.map.aggregate([{"$match":{"amenity":{"$exists":1},”amenity”:”restaurant”}}, {"$group":{"\_id":"$cuisine", "count":{"$sum":1}}}, {"$sort":{"count":­-1}}])



1. **Address type not uniform with abbreviations.** In my auditing the data session, I went into more details on this problem. I included the code with the 9 steps that I took to resolve the data with address that has abbreviations and improper format. In my solution, I choose to remove the abbreviations and change the dataset into the format below:

La Vida Ln => La Vida Lane

Center St. => Center Street

Jackson St. => Jackson Street

Judge John Aiso St. => Judge John Aiso Street

N. Alameda St. => N. Alameda Street

E. Temple St. => E. Temple Street

South Almansor St. => South Almansor Street

E. 1st St. => E. 1st Street

1. DataOverview

This section contains basic statistics for the temple city map dataset and the MongoDB queries used to obtain the information.

The data.py from lesson 6 prepared the dataset “example.osm” to “example.osm.json”

Mongoimport –db map –collection map –file example.osm.json

File size

* db.map.dataSize()
* 104659232 : 104.6 MB

# Number of nodes

* db.map.find({“type”:”node”}).count()
* 295587

# Number of ways

* db.map.find({“type”:”way”}).count()
* 43693

# Number of unique users

* db.map.distinct(“created.user”).length
* 503

#Number of address

* db.map.find({“address”:{$exists:true}}).count()
* 57953

# Number of unique version

* db.map.distinct(“created.version”).length
* 94

1. Additional Ideas

# Popular amenity

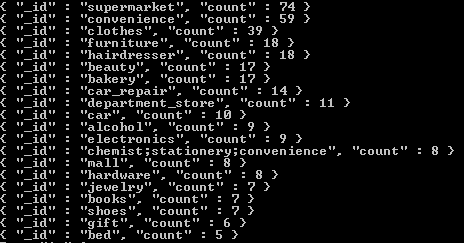
* db.map.aggregate([{"$match":{"amenity":{"$exists":1}}}, {"$group":{"\_id":"$amenity ", "count":{"$sum":1}}}, {"$sort":{"count":­-1}}])



* I am surprise to find that “place to worship” is the top 3 amenity in the San Garbiel Valley area.

#Popular shop

* db.map.aggregate([{"$match":{"shop":{"$exists":1}}}, {"$group":{"\_id":"$shop", "count":{"$sum":1}}}, {"$sort":{"count":­-1}}])



* SGV has a decent number of supermarket and convenience store. However, since I have been living in the area for more than 20 years, based on my observation, this number is smaller than the actual number. Therefore, I believe the data is incomplete.

1. Auditing the Data (code stored in clean.py)

Step 1: Import library and file. Parse through the file with ElementTree to find the counts of each element.

*#Import library*

**import** xml.etree.cElementTree **as** ET

**from** collections **import** defaultdict

**import** re

**import** pprint

*#Import file*

OSMFILE = "example.osm"

tags = {}

**for** event, elem **in** ET.iterparse(OSMFILE):

**if** elem.tag **in** tags:

tags[elem.tag] += 1

**else**:

tags[elem.tag] = 1

pprint.pprint(tags)

{'bounds': 1,

'member': 16372,

'meta': 1,

'nd': 320045,

'node': 249298,

'note': 1,

'osm': 1,

'relation': 561,

'tag': 220126,

'way': 36774}

Step 2: From lesson 6, we defined lower, lower\_colon, and problemchars. The code below is to show the counts of each of the definition below.

Lower = Strings containing lower case characters

Lower\_colon = Strings containing lower cases characters and a single colon within the string

Problemchars = characters that cannot be used within keys in MongoDB

lower = re.compile(r'^([a-z]|\_)\*$')

lower\_colon = re.compile(r'^([a-z]|\_)\*:([a-z]|\_)\*$')

problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')

**def** key\_type(element, keys):

**if** element.tag **==** "tag":

**if** problemchars.search(element.attrib['k']):

keys['problemchars'] += 1

**elif** lower.search(element.attrib['k']):

keys['lower'] += 1

**elif** lower\_colon.search(element.attrib['k']):

keys['lower\_colon'] += 1

**else**:

keys['other'] += 1

**return** keys

**def** process\_map(OSMFILE):

keys = {"lower": 0, "lower\_colon": 0, "problemchars": 0, "other": 0}

**for** \_, element **in** ET.iterparse(OSMFILE):

keys = key\_type(element, keys)

**return** keys

 keys = process\_map(OSMFILE)

pprint.pprint(keys)

{'lower': 116548, 'lower\_colon': 100586, 'other': 2990, 'problemchars': 2}

Step 3: Build an expression to match the token in a sting ending with a period and the expected clean street type.

street\_type\_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)

expected\_street\_types = ["Avenue", "Boulevard", "Commons", "Court", "Drive", "Lane", "Parkway",

"Place", "Road", "Square", "Street", "Trail"]

map\_street\_types = \

{

"Ave" : "Avenue",

"BLVD" : "Boulevard",

"Blvd" : "Boulevard",

"Blvd." : "Boulevard",

"Cir" : "Circle",

"Dr" : "Drive",

"Ln" : "Lane",

"Pkwy" : "Parkway",

"Rd" : "Road",

"Rd." : "Road",

"St" : "Street",

"St." : "Street"

}

Step 4: The audit\_string function will take in the dictionary of the regex and street types in step 3 to match against that string and the list of expected street types. If there is a match and the match is not in our list, then it will add the match as a key to the dictionary and the string to the set.

def audit\_string(match\_set\_dict, string\_to\_audit, regex, expected\_matches):

m = regex.search(string\_to\_audit)

if m:

match\_string = m.group()

if match\_string not in expected\_matches:

match\_set\_dict[match\_string].add(string\_to\_audit)

Step 5: The audit function will do the parsing and auditing of the street names in our database.

def audit(osmfile, tag\_filter, regex, expected\_matches = []):

osm\_file = open(osmfile, "r")

match\_sets = defaultdict(set)

# iteratively parse

for event, elem in ET.iterparse(osm\_file, events=("start",)):

# iterate the tags within node or way

if elem.tag == "node" or elem.tag == "way":

for tag in elem.iter("tag"):

if tag\_filter(tag):

audit\_string(match\_sets, tag.attrib['v'], regex, expected\_matches)

return match\_sets

Step 6: The is\_street\_name function will determine if the element has “addr:street” attribute. Also, we can see a print of the audit output.

def is\_street\_name(elem):

return (elem.attrib['k'] == "addr:street")

street\_types = audit(OSMFILE, tag\_filter = is\_street\_name, regex = street\_type\_re,

expected\_matches = expected\_street\_types)

pprint.pprint(dict(street\_types))

{'106': set(['South Lake Ave #106']),

'21': set(['South Avenue 21']),

'3rd': set(['E. 3rd']),

'52': set(['Avenue 52']),

'53': set(['N Avenue 53']),

'56': set(['N Avenue 56']),

'61': set(['North Avenue 61']),

'64': set(['North Avenue 64']),

'Alley': set(['Martin Alley', 'Miller Alley']),

Step 7: The update function will replace the abbreviated street types. The string to update function will take a string to update, mapping the dictionary, and a regex to search.

def update(string\_to\_update, mapping, regex):

m = regex.search(string\_to\_update)

if m:

match = m.group()

if match in mapping:

string\_to\_update = re.sub(regex, mapping[match], string\_to\_update)

return string\_to\_update

Step 8: Take the keys from the map to create a string joined by “|”. This will cause the regex to search for any of the keys to match the first it finds.

bad\_streets = "|".join(map\_street\_types.keys()).replace('.', '')

street\_type\_updater\_re = re.compile(r'\b(' + bad\_streets + r')\b\.?', re.IGNORECASE)

Step 9: Traverse the street\_type dictionary to see the abbreviations to clean representations.

for street\_type, ways in street\_types.iteritems():

if street\_type in map\_street\_types:

for name in ways:

new\_name = update(name, map\_street\_types, street\_type\_updater\_re)

print name, "=>", new\_name

La Vida Ln => La Vida Lane

Center St. => Center Street

Jackson St. => Jackson Street

Judge John Aiso St. => Judge John Aiso Street

N. Alameda St. => N. Alameda Street

E. Temple St. => E. Temple Street

South Almansor St. => South Almansor Street

E. 1st St. => E. 1st Street

1. Conclusion

Our goal is to assess the dataset to test assumptions about the values, data types, and shape. Also, we need to identify errors or outliers in the data and finding the missing values. Based on the analysis of this dataset, I can conclude that the San Garbiel Valley map dataset is incomplete and still need a lot of work. In my “audit the data” exercise, my 9 steps cleaned up the abbreviation in the address names so they are more uniform. However, there are much more dirty data that needs to clean up, such as the postal code, cardinal directions of the address, and amenity names. Beyond the scope of this course, we can analyze the inputs from each user and understand their error pattern from their entry so we can set up a coding standard for the dataset.

I am glad that openstreetmap open their data source and allow users to make updates to improve the dataset. This exercise taught me how to do queries and analysis big data using MongoDB and pylon. Users can use this dataset to solve problems such as choosing a business marketing strategy to target a specific race or religion in an area or choosing a location that is suitable to open a restaurant without intense competition.