Project 3 Wrangle Open Street Map Data

By Kathleen Cravotta, November 2016

1. Converting the OSM XML to structured JSON

The New York, NY data was downloaded from the MapZen MetroExtract maps. I used the provided code from Udacity to extract 1/1000 to a small sample file to do my initial work with. While processing this data, I did the following:

- Converted abbreviated street names as per 'Improving Street Names' exercise.
- Create a data structure for compound tags when the second level tag keys had colons. For this approach, needed to handle tags that had both value and children
- Checked for duplicate tag names in multiple parts of the document
- Checked for data types per field

Abbreviated Street Names

I audited the street types to check any abbreviated values and verified that the street types are valid. In my chosen region, sometimes Avenue is designated at the beginning of a street name instead of the end. And in some streets, a suffix for direction is included. After auditing this data, I prepared a function to improve the street name by fixing the type.

Sample of Mapping of abbreviations or typos to street type:

```
 mapping = { "Rd" : "Road",
    "Steet" : "Street",
    "STREET" : "Street",
    "avenue" : "Avenue",
    "ROAD" : "Road",
    "Pkwy" : "Parkway",
    "Cir" : "Circle",
    "AVenue" : "Avenue",
    "Tunrpike" : "Turnpike",
    ...

      Sample of correction:
      Valley Rd => Valley Road
      West Merrick Rd => West Merrick Road
      MINE STREET => MINE Street
      Bedford avenue => Bedford Avenue
      Cross Island Pkwy => Cross Island Parkway
      Barron Cir => Barron Circle
      Wantagh AVenue => Wantagh Avenue
      ...
```

Compound Tags

I chose to create a data structure when the second level tag keys had colons. To achieve this, I created a dictionary for the first part of the key, and used recursion to pass the rest of the key along with the dictionary to the same method.

```
...
levels = name.split(':')
top = levels[0]
...
#add to a dict if there are multiple levels
elif top!='address' and len(levels) > 1:
      if top not in tags: tags[top]={}
      process_tag (name.split(':', 1)[1], value, tags[top])
```

Tags With Both Value and Children

When processing the data, a few data elements generated exceptions. I investigated to find this happened when a second-level tag had both a value and was the parent of a compound tag.

```
Output: (sample)
      maxspeed:hgv 50 mph  simple value already found as  55 mph
      maxspeed:goods 50 mph  simple value already found as  55 mph
      building:levels 2  simple value already found as  terrace
      railway:track_ref 10  simple value already found as  subway
      building:levels 2  simple value already found as  terrace
      lanes:forward 4  simple value already found as  7
      building:levels 6  simple value already found as  yes
      railway:track_ref 3  simple value already found as  subway
```

I modified the code so that if there was compound tag and the root label was on the designated list, then rename the dictionary of children values by suffixing it with _data. (The drawback of my solution is that I only covered this situation when it is within any individual record.)

```
            #add to a dict if there are multiple levels
            elif top!='address' and len(levels) > 1:
                  if top in has_value_and_children: top = top + "_data"
                  elif top in tags and not isinstance(tags[top],dict):
                     #a root value was already found for this tag.
                  print name, value, ' unexpected - simple value already 
found as ', top, tags[top]
                  return
                  if top not in tags: tags[top]={}
                 process_tag (name.split(':', 1)[1], value, tags[top])
```

Duplicate Tags

For checking the quality of the format of the data, I checked if the same tag name appeared multiple nested parts of the documents. I did see that some contact tags and address tags match tags at the top layer of the records. Redundancy could warrant a further analysis of the data. This particular attempt didn't help identify that tiger and [other] sources store address data with different tags.

```
Duplicate tags (sample):
... u'colour': set([u'building_data.colour', u'roof.colour']),
  u'goods': set([u'goods', u'maxspeed_data.goods']),
  u'height': set([u'height', u'roof.height']),
  u'hgv': set([u'hgv', u'maxspeed_data.hgv', u'source.hgv']),
  u'id': set([u'gnis.id', u'id']), ...
```

2. Descriptions and Auditing in MongoDB

I used the mongo shell to import the json file, then continued to audit the data by using pymongo.

Here are some basic statistics about the dataset and the MongoDB queries used to gather them. Then I share a few more observations and improvements on the quality of the data.

Descriptions of the data

Source file and count of tags:

```
2,355,723,043 new-york_new-york.osm
{'bounds': 1,
 'member': 100531,
 'nd': 12155810,
 'node': 9412590,
 'osm': 1,
 'relation': 8351,
 'tag': 8709038,
 'way': 1538707}

2,621,841,121 new-york_new-york.osm.json
```

Number of documents

```
db.nyny.count()
10951297
```

Number of Nodes and Ways

```
distinct_with_count("type")
{u'count': 1538707, u'_id': u'way'},
{u'count': 9412590, u'_id': u'node'}
```

Number of distinct values

```
def make_countdistinct_pipeline( by_field ):
    pipeline = [
        { "$match": { by_field : { "$exists" : True } } },
        { "$group" : { "_id": "$" + by_field }},
        { "$group": { "_id": by_field, "count" : {"$sum" : 1 } } } ]

Distinct  created.user  - count 3852
Distinct  address.postcode  - count 765
Distinct  address.city  - count 430
Distinct  source  - count 436
```

Top Values

```
pipeline = [
     { "$match": { by_field : { "$exists" : True } } },
     { "$group" : { "_id": "$" + by_field , "count" : {"$sum" : 1 }}},
     { "$sort": { "count" :-1 }},
     { "$limit" : 1 } ]

Top created.user Rub21_nycbuildings  - count 4923
Top address.postcode 11234  - count 24
Top address.city New York  - count 4
Top amenity parking  - count 10
Top source Bing  - count 6
```

Auditing the Zip codes and the States

I queried the distinct zip codes, sorted, so that I could spot check them for invalid values. The entries that don't start with 10-11 and entries that don't have length of 5 are the ones that could require attention..

```
      pipeline = [
       { "$match": { "address.postcode" : { "$exists" : True } } },
      { "$group" : { "_id": "$address.postcode", "count" : {"$sum" : 1
}}},
      { "$sort": {"_id":1 }} ]

    RESULT:
      ... {u'count': 849, u'_id': u'08816'}, ...
          {u'count': 97, u'_id': u'08854'}, ...
```

These two zip codes from the above output start with 0, so I searched for them on usps and found 08816 is in EAST BRUNSWICK NJ and 08854 is in PISCATAWAY NJ. There were other zip codes on the map that had other issues - such as NY or NJ prefixes, a phone number, two digit values, and some with the 4 digit zip code suffix after a dash. I also checked the address states and saw that many of the entries were from NJ. There were many state values that can be improved, such as lower case and spelled out ones.

Improvement on OSM XML conversion to JSON

While exploring an initial sample set osm records in MongoDB, I noticed that many records didn't have pos, in other words, lacked the latitude and longitude coordinates. So I revisited my shaping function and omitted the pos data when it is not populated.

```
        if None not in pos: attributes['pos'] = pos
```

Then when working with the full set of data, I saw that there was already a 'type' field in some of the records, but the translation required 'type' as the name of the value specifying the root element from the xml document. So I needed to rename the one in the original data set..

```
        replace_tags = {'addr':'address','type': 'type_as_specified'}
```

After both of these fixes, the next steps were to regenerate the file, wipe out the database, and import the new file. So it was an iterative process.

3. Exploring the data in MongoDB

This section contains further explorations to the dataset.

```
get_distincts = ["source", "cuisine", "leisure", "office", "service", "shop", "sport", 
“amenity”]
for field_name in get_distincts:
    print "Distinct ", field_name
    pprint.pprint (distinct_with_count(field_name))

Distinct source
[{u'_id': u'NJ2002LULC', u'count': 7618},
 {u'_id': u'Bing', u'count': 5581},
 {u'_id': u'USGS Geonames', u'count': 2021},
 {u'_id': u'Yahoo', u'count': 1733}, ...

Distinct cuisine
[{u'_id': u'coffee_shop', u'count': 299},
 {u'_id': u'burger', u'count': 293},
 {u'_id': u'pizza', u'count': 259},
 {u'_id': u'italian', u'count': 244}, ...

Distinct leisure
[{u'_id': u'pitch', u'count': 6230},
 {u'_id': u'park', u'count': 3509},
 {u'_id': u'playground', u'count': 1219},
 {u'_id': u'swimming_pool', u'count': 680}, ...

Distinct office
[{u'_id': u'company', u'count': 83},
 {u'_id': u'yes', u'count': 42},
 {u'_id': u'government', u'count': 30},
 {u'_id': u'lawyer', u'count': 30}, ...

Distinct service
```

```
[{u'_id': u'parking_aisle', u'count': 14238},
 {u'_id': u'driveway', u'count': 4200},
 {u'_id': u'yard', u'count': 2671},
 {u'_id': u'spur', u'count': 1296}, ...

Distinct shop
[{u'_id': u'supermarket', u'count': 637},
 {u'_id': u'convenience', u'count': 461},
 {u'_id': u'clothes', u'count': 459},
 {u'_id': u'yes', u'count': 184}, ...

Distinct sport
[{u'_id': u'baseball', u'count': 1986},
 {u'_id': u'tennis', u'count': 1533},
 {u'_id': u'basketball', u'count': 950},
 {u'_id': u'soccer', u'count': 592}, ...

Distinct amenity
[{u'_id': u'parking', u'count': 7061},
 {u'_id': u'bicycle_parking', u'count': 4874},
 {u'_id': u'school', u'count': 4703},
 {u'_id': u'place_of_worship', u'count': 4694},
 {u'_id': u'restaurant', u'count': 3313}, ...
```

4. Additional Ideas

Merging the data from other sources

The data imported from tiger and gnis store the address or municipality specific information in different fields. It could be helpful to further develop the 'shape' function to provide a richer transformation from OSM XML tags to the JSON document, to use the address structure for all of them.

I did try to run an analysis to see if there were any duplicate entries between the tiger imports and the osm address data, but this type of comparison would be dependent on more complex transformation because the formats of the street names are different.

Trying to reconcile the streets and other data that are in different formats into one shared data type can diminish the quality of the data if some of the street names are re-formatted improperly during transformation. Possibly, cross referencing the street-names with a good set of data, or running multiple compose-decompose functions could help prevent or find these issues.

Improving State Names and Zip codes

Many of the values for state names and zip codes can be corrected like the street names were. A solution for the state names could be similar to the solution for street names, where a value like "Ny" can map to "NY". The solution for zip codes could be dynamic, to parse out the 5 digit value and use it, removing any characters such as NY or the 4 digit suffix.