OpenStreetMap Data Case Study
“Data Wrangling with MongoDB” (Udacity.com)

Name
Matthew T. Banbury

Udacity Email
matthewbanbury@gmail.com

Map Area
Charlotte, NC, United States
https://www.openstreetmap.org/relation/177415
http://metro.teczno.com/#charlotte

This map is of my hometown, so I’m more interested to see what database querying reveals, and I’d like an opportunity to contribute to its improvement on OpenStreetMap.org.

Problems Encountered in the Map

After initially downloading a small sample size of the Charlotte area and running it against a provisional `data.py` file, I noticed five main problems with the data, which I will discuss in the following order:

- Over-abbreviated street names (`“S Tryon St Ste 105”`)
- Inconsistent postal codes (`“NC28226”, “28226-0783”, “28226”`)
- “Incorrect” postal codes (*Charlotte area zip codes all begin with “282” however a large portion of all documented zip codes were outside this region.*)
- Second-level “k” tags with the value `“type”` (*which overwrites the element’s previously processed* `node[“type”]` *field*).
- Street names in second-level “k” tags pulled from Tiger GPS data and divided into segments, in the following format:

```
<tag k="tiger:name_base" v="Stonewall"/>
<tag k="tiger:name_direction_prefix" v="W"/>
<tag k="tiger:name_type" v="St"/>
```

Over-abbreviated Street Names

Once the data was imported to MongoDB, some basic querying revealed street name abbreviations and postal code inconsistencies. To deal with correcting street names, I opted not use regular expressions, and instead iterated over each word in an address, correcting them to their respective mappings in `audit.py` using the following function:

```
def update(name, mapping):
    words = name.split()
    for w in range(len(words)):
        if words[w] in mapping:
            if words[w-1].lower() not in ['suite', 'ste.', 'ste']:
            # For example, don't update 'Suite E' to 'Suite East'
                words[w] = mapping[words[w]]
    name = " ".join(words)
    return name
```

This updated all substrings in problematic address strings, such that:

"S Tryon St Ste 105"

becomes:

"South Tryon Street Suite 105"

Postal Codes

Postal code strings posed a different sort of problem, forcing a decision to strip all leading and trailing characters before and after the main 5-digit zip code. This effectually dropped all leading state characters (as in "NC28226") and 4-digit zip code extensions following a hyphen ("28226-0783"). This 5-digit constriction benefits MongoDB aggregation calls on postal codes.

Regardless, after standardizing inconsistent postal codes, some altogether "incorrect" (or perhaps misplaced?) postal codes surfaced when grouped together with this aggregator:

```
# Sort postcodes by count, descending
db.char.aggregate([{"$match":{"address.postcode":{"$exists":1}}},
                   {"$group":{"_id":"$address.postcode",
                              "count":{"$sum":1}}},
                   {"$sort":{"count":-1}}])
```

Here are the top ten results, beginning with the highest count:

```
[   {"_id" : "29732", "count" : 103},     #
    {"_id" : "28134", "count" : 27},      #
    {"_id" : "28226", "count" : 17},
    {"_id" : "29745", "count" : 10},      #
    {"_id" : "28216", "count" : 10},
    {"_id" : "28105", "count" : 10},      #
    {"_id" : "28104", "count" : 8},       #
    {"_id" : "28027", "count" : 7},       #
    {"_id" : "28273", "count" : 7},
    {"_id" : "29730", "count" : 7},...]   #
```

These results were taken before accounting for Tiger GPS zip codes residing in second-level “k” tags. Considering the relatively few documents that included postal codes, of those, it appears that out of the top ten, seven aren’t even in Charlotte, as marked by a “#”. That struck me as surprisingly high to be a blatant error, and found that the number one postal code and all others starting with `"297"` lie in **Rock Hill, SC**. So, I performed another aggregation to verify a certain suspicion…

```
# Sort cities by count, descending
> db.char.aggregate([{"$match":{"address.city":{"$exists":1}}},
                     {"$group":{"_id":"$address.city",
                                "count":{"$sum":1}}},
                     {"$sort":{"count":-1}}])
```

And, the results, edited for readability:

```
[   {"_id" : "Rock Hill",     "count" : 111},
    {"_id" : "Pineville",     "count" : 27},
    {"_id" : "Charlotte",     "count" : 26},
    {"_id" : "York",          "count" : 24},
    {"_id" : "Matthews",      "count" : 10},
    {"_id" : "Concord",       "count" : 4},
    {"_id" : "Lake Wylie",    "count" : 2},
    {"_id" : "Locust",        "count" : 1},
    {"_id" : "Monroe",        "count" : 1},
    {"_id" : "Fort Mill, SC", "count" : 1},
    {"_id" : "Belmont, NC",   "count" : 1},
    {"_id" : "Rock Hill, SC", "count" : 1}   ]
```

These results confirmed my suspicion that this metro extract would perhaps be more aptly named **“Metrolina”** or the **“Charlotte Metropolitan Area”** for its inclusion of surrounding cities in the sprawl. More importantly, three documents need to have their trailing state abbreviations stripped. So, these postal codes aren’t “incorrect,” but simply unexpected. However, one final case proved otherwise.

A single zip code stood out as clearly erroneous. Somehow, a `"48009"` got into the dataset. Let’s display part of its document for closer inspection (for our purposes, only the `"address"` and `"pos"` fields are relevant):

```
> db.char.find({"address.postcode":"48009"}).pretty()
{
    ...
    "pos" : [
            35.2134608,
            -80.8270161
    ],
    "address" : {
```

```
            "street" : "North Old Woodward Avenue",
            "housenumber" : "280",
            "postcode" : "48009"
    },
    ...
}
```

It turns out, *"280 North Old Woodward Avenue, 48009"* is in **Birmingham, Michigan**. All data in this document, including those not shown here, are internally consistent and verifiable, *except* for the latitude and longitude. These coordinates are indeed in Charlotte, NC. I'm not sure about the source of the error, but we can guess it was most likely sitting in front of a computer before this data entered the map. The document can be removed from the database easily enough:

```
> db.char.remove( { "address.postcode" : "48009" } )
```

Second-Level "K" Tags with Value "type"

While attempting to validate some basic statistics of the OSM file, I compared the original results processed with `mapparser.py` to some queries in MongoDB.

Here are the `mapparser.py` "node" and "way" counts:

```
{   "node" : 1471350,
    "way"  : 84502  }
```

And, their relative MongoDB queries:

```
> db.char.find({"type":"node"}).count()
1471344

> db.char.find({"type":"way"}).count()
84501
```

Let's find out what's responsible for that 7-document discrepancy. First, a count:

```
# Count all non-"node" non-"way" docs
> db.char.find({"type": {"$nin":["node","way"]}}).count()
6
```

The count is only 6 because `mapparser.py` counted the Birmingham, Michigan document I removed above, but it's now removed from our database. So far, so good. Now, let's display a count of those 6 documents by type:

```
# Count the discrepant docs by type
> db.char.aggregate([{"$match":{"type":{"$nin":["node","way"]}}},
                     {"$group":{"_id":"$type",
                                "count":{"$sum":1}}},
                     {"$sort":{"count":-1}}])
```

The results:

```
[ {"_id" : "public",     "count" : 4},
  {"_id" : "outdoor",    "count" : 1},
  {"_id" : "child care", "count" : 1} ]
```

The fact that these six documents have type fields with values not equal to “node” or “way” was puzzling, at first, considering how `shape_element()` only processes either of those two tags. So the simplest explanation would be that somewhere inside each of those certain elements lie a value whose text is the string “type.” This turned out to be the case, and making the same queries above after reprocessing the database returned equivalent tag counts for “node” and “way.” A simple fix to the `node_update_k()` function in `data.py` changes the node’s field name to `“service_type”` if `k == “type”`.

Tiger GPS Data

Closer inspection of the documents after MongoDB import revealed that thousands of OSM elements contained street name information for which `shape_element()` did not account. In the OSM file, Tiger GPS street name information took the form:

```
<tag k="tiger:cfcc" v="A41"/>
<tag k="tiger:county" v="Mecklenburg, NC"/>
<tag k="tiger:name_base" v="12th"/>
<tag k="tiger:name_direction_prefix" v="E"/>
<tag k="tiger:name_type" v="St"/>
<tag k="tiger:reviewed" v="no"/>
<tag k="tiger:zip_left" v="28206"/>
<tag k="tiger:zip_right" v="28206"/>
```

Not all values here are of use, but we can certainly take advantage of any `“name_”` values and `“zip_left”` but `“zip_right”` is irrelevant to our audit, even if the value were correct. It will take a little work to join together the street name segments, however, many Tiger GPS elements also contains a `k=“name”` tag in an element always preceding all `“tiger:__”` tags. This tag combines all street name segments and fortunately even fleshes out all prefix, type, and suffix abbreviations. For example:

```
<tag k="name" v="South Dotger Avenue"/>
<tag k="tiger:name_base" v="Dotger"/>
<tag k="tiger:name_direction_prefix" v="S"/>
<tag k="tiger:name_type" v="Ave"/>
```

In these cases, `“name”` would have already been processed and could be accessed in the `node` before needing to combine street name segments. When not present, `process_tiger()` and `join_segments()` work together to save the segments as a dictionary value into the field `node[“address”][“street”]` then replace the value with the full street name after processing all segments.

Accounting for and cleaning Tiger GPS tags had a significant effect on the overall shape of the dataset. See the Data Overview and Additional Ideas for details about before and after processing.

Data Overview and Additional Ideas

This section contains basic statistics about the dataset, the MongoDB queries used to gather them, and some additional ideas about the data in context.

File sizes

```
charlotte.osm ......... 294 MB
charlotte.osm.json .... 322 MB
```

```
# Number of documents
> db.char.find().count()
1555851
```

```
# Number of nodes
> db.char.find({"type":"node"}).count()
1471349
```

```
# Number of ways
> db.char.find({"type":"way"}).count()
84502
```

```
# Number of unique users
> db.char.distinct({"created.user"}).length
336
```

```
# Top 10 contributing users
> db.char.aggregate([{"$group":{"_id":"$created.user",
                                "count":{"$sum":1}}},
                   {"$sort":{"count":-1}},
                   {"$limit":1}])
```

```
[   { "_id" : "jumbanho",         "count" : 823324 }
    { "_id" : "woodpeck_fixbot", "count" : 481549 },
    { "_id" : "TIGERcnl",         "count" : 44981 },
    { "_id" : "bot-mode",         "count" : 32033 },
    { "_id" : "rickmastfan67",    "count" : 18875 },
    { "_id" : "Lightning",        "count" : 16924 },
    { "_id" : "grossing",         "count" : 15424 },
    { "_id" : "gopanthers",       "count" : 14988 },
    { "_id" : "KristenK",         "count" : 11023 },
    { "_id" : "Lambertus",        "count" : 8066 }   ]
```

```
# Number of users appearing only once (having 1 post)
> db.char.aggregate([{"$group":{"_id":"$created.user",
                                "count":{"$sum":1}}},
                   {"$group":{"_id":"$count",
                              "num_users":{"$sum":1}}},
                   {"$sort":{"_id":1}},
                   {"$limit":1}])

[   { "_id" : 1, "num_users" : 56 }   ]  # "_id" represents post count
```

Additional Ideas about Users

The contributions of users seems incredibly skewed, possibly due to automated versus manual map editing (*the word "bot" appears in some usernames*). Here are some user percentage statistics:

- Top user contribution percentage (**"jumbanho"**)
  52.92%
- Combined top 2 users' contribution (**"jumbanho"** and **"woodpeck_fixbot"**)
  83.87%
- Combined Top 10 users contribution
  94.3%
- Combined number of users making up only **1%** of posts
  287 (about 85% of all users)

Query results after implementing additional Tiger GPS processing ideas

```
# Number of docs with an address field before
> db.char.find({"address":{"$exists":1}}).count()
488
# And after
41414
```

```
# Number of docs with a postal code field before
> db.char.find({"address.postcode":{"$exists":1}}).count()
245
# And after
399
```

```
# Number of docs with a street name field before
> db.char.find({"address.street":{"$exists":1}}).count()
259
# And after
37053
```

```
# Number of actual Charlotte postal codes before
db.char.find({"address.postcode":{$regex:/^282/}}).count()
61
# And after
86
```

Miscellaneous results

```
# Top 10 appearing amenities
> db.char.aggregate([{"$match":{"amenity":{"$exists":1}}},
                     {"$group":{"_id":"$amenity",
                                "count":{"$sum":1}}},
                     {"$sort":{"count":-1}},
                     {"$limit":10}])

[   {"_id" : "place_of_worship", "count" : 587},
    {"_id" : "school",           "count" : 416},
    {"_id" : "parking",          "count" : 345},
    {"_id" : "restaurant",       "count" : 123},
    {"_id" : "grave_yard",       "count" : 81},
    {"_id" : "fast_food",        "count" : 72},
    {"_id" : "fire_station",     "count" : 52},
    {"_id" : "fuel",             "count" : 37},
    {"_id" : "library",          "count" : 32},
    {"_id" : "bench",            "count" : 30}   ]
```

```
# Biggest religion (no surprise here)
> db.char.aggregate([{"$match":{"amenity":{"$exists":1},
                                "amenity":"place_of_worship"}},
                     {"$group":{"_id":"$religion",
                                "count":{"$sum":1}}},
                     {"$sort":{"count":-1}},
                     {"$limit":1}])

[   { "_id" : "christian", "count" : 577 }   ]
```

```
# Most popular cuisines
> db.char.aggregate([{"$match":{"amenity":{"$exists":1},
                                "amenity":"restaurant"}},
                     {"$group":{"_id":"$cuisine",
                                "count":{"$sum":1}}},
                     {"$sort":{"count":-1}},
                     {"$limit":8}])
```

```
[   { "_id" : "null",         "count" : 65 }, # undocumented
    { "_id" : "pizza",        "count" : 9 },
    { "_id" : "american",     "count" : 9 },
    { "_id" : "mexican",      "count" : 5 },
    { "_id" : "steak_house", "count" : 5 },
    { "_id" : "chinese",      "count" : 4 },
    { "_id" : "sandwich",     "count" : 4 },
    { "_id" : "ice_cream",    "count" : 3 }   ]
```

Conclusion

After this review of the data it's obvious that the Charlotte area is incomplete, though I believe it has been well cleaned for the purposes of this exercise. It interests me to notice a fair amount of GPS data makes it into OpenStreetMap.org on account of users' efforts, whether by scripting a map editing bot or otherwise. With a rough GPS data processor in place and working together with a more robust data processor similar to `data.py` I think it would be possible to input a great amount of cleaned data to OpenStreetMap.org.

Thinking about the user percentages I outlined earlier, I'm reminded of "gamification" as a motivating force for contribution. In the context of the OpenStreetMap, if user data were more prominently displayed, perhaps others would take an initiative in submitting more edits to the map. And, if everyone sees that only a handful of power users are creating more than 90% a of given map, that might spur the creation of more efficient bots, especially if certain gamification elements were present, such as rewards, badges, or a leaderboard.

Thank you for reading, and please refer to the Python files to see implementations of discussed ideas.