OSM Data Audit

February 28, 2018

1 OpenStreetMap Analysis

Providence, RI and surronding area

1.1 Initial download and review

The target for this analysis is the OpenStreetMap data for the city of Providence, Rhode Island, and the contiguous urban area. This includes a large portion of Rhode Island, and parts of Massachusetts; covering cities from Warwick and Bristol in RI, up to Attleboro and Rehoboth in MA. The file was downloaded on November 28, 2017.

The downloaded XML file is approximately 170MB in size. The large size of the file prohibits any sort of in-memory analysis of the XML data. My immediate goal was a naive parsing of the file into JSON format so the data could be loaded into MongoDB and reviewed there. Following the initial review in MongoDB, I would then update the parsing process to better clean and capture the XML data. So, for the first pass, my central task was to parse all the data without the script choking.

Looking at the OSM Wiki (https://wiki.openstreetmap.org/wiki/OSM_XML), there's an immediate problem. The OSM XML format doesn't have an XSD schema associated with it; so, there's no way to validate the contents of the XML file. We're relying on the OSM software to spit out the data in a consistent format. It mostly likely will, but you can't make any assumptions when using other people's data. If there are irregularities in the XML data -- inconsistent element attributes, irregular nesting of elements, irregular element contents, etc. -- this could halt the script, or lead to anomalies in the JSON output.

The wiki does offer some good news, however. We get a full list of the "data primitives" in the OSM XML: https://wiki.openstreetmap.org/wiki/Elements. Specifically, there are `nodes`, `ways`, and `relations`. We also learn that these nodes can have child elements. All 3 can have `tag` elements as children; additionally, `relations` can have `member` children, and `ways` can have `nd` children. We also get a list of element attributes, but there's no promises about the consistency of their presence.

The main function in the audit file, `parse_osm_xml`, includes a number of assertions to confirm that all the data adds up. As the function parses each indidvidual XML element, it updates several running counts of simple information. These are: * the total number of each element (including both parent and child elements) * the total number of each attribute, grouped by element * the total number child elements, grouped by child element, grouped by parent element

Prior to writing the JSON output, the assertions confirm that 1) the total number of each element attribute equals the total number of elements (so all attributes are always present) and 2) the total number of possible child elements (ie, `nd`, `tag`, `member`) equals the number of those elements that are in fact children of parent elements. There is a third assertion, confirming that none of the elements include text content. While buliding the counts for validation, the script also grabs sample values of various data. These provide an easy way to get a sense of what the data looks like, without having to open up the XML file. And anyhow, we're already going through the file, so we may as well grab whatever we can while we're there. The samples are: * each element attribute and one its values, grouped by element * each `key` and `value` attribute value on every tag element

After running the script a few times and outputting these counts and samples to the console, I realized it made more sense to write their contents to a file for future reference. The output is stored in the included `data_key.json` file, which proved helpful in developing the project.

1.2 XML parsing: first pass

With the assertions in place, the XML parsing script could make certain assumptions about the incoming data without a lot of exception handling. The workhorse function is `model_elem`, which grabs `node`, `way`, and `relation` XML elements, and converts them, their attributes, and their child elements into a nested dictionary structure. The returned dicitionaries are then converted to JSON for upload to MongoDB.

Modeling the main OSM datatypes was fairly straightforward, since I knew what attributes to expect, and that they'd always be present on all the elements. These all relate to administrative metadata, with `nodes` also having attributes for `lat` and `lon`. I added an additional attribute, `datatype`, which captures what kind of an element each dictionary was built from, making it easier to filter to future MongoDB queries.

Interestingly, the `data_key.json` file revealed that there were hidden top-level elements not mentioned in the OSM wiki. In addition to the root `osm` element, and a `bounds` element shown in the wiki example data, there are also `note` and `meta` elements. I sidestepped these by adding conditional logic to the script, only processing `nodes`, `ways`, and `relations`.

The greatest complexity was in handling the child elements, specifically `tags`. `ways` and `relations` also have `nd` and `member` children, respectively. Fortunately, both those elements' most important attributes are the same: a `ref` attribute, pointing at a `node` id. These are handled using the same logic, leaving only the `tags` to contend with.

The `tag` handling gets its own function, `model_tag`. All information for a `tag` is contained in its `k` and `v` attributes, which is straightforward enough. The values of these attributes are to be stored in the dictionary for the `tag`'s parent element. The naive solution would be to store every `k` value as a key in the parent dictionary, and every `v` as the key's value. There would be data lost there, however, since some of the `ks` and `vs` are related to each other. These are hierarchical relationships, indicated by a colon in the `k` value. Optimally, we would capture these as nested documents within the parent document, allowing for smarter, richer MongoDB queries.

My initial solution was to split the `k` value on the `:` character if it was present, and use the result to build the nested data. This caused the script to break, however, and a look at `data_key.json` quickly showed why. Some `k` values include multiple colons, indicating a deeply nested structure (see the `seamark` and `service` attribute values). Reviewing the data, these `tags` were uncommon enough that I decided to only nest the attribute values 1 level deep. Some structure would be lost, but nothing that seemed worth the added complexity.

This process covered the bulk of XML data capture. The next step was to run the process, get the JSON, load it into MongoDB, and see what was there.

1.3 Initial stats and observations

With the data loaded into MongoDB, running some initial numbers using the Mongo client was straightforward. Better, I discovered that these queries could be saved as Javascript files, run against MongoDB from the command line, and the output piped to text files for storage. The first set of queries is stored in `queries/general_stats.js`, and the output in `data/general_stats.json`. (This naming scheme holds for all sets of queries and outputs.) The queries give us a basic overview of the database.

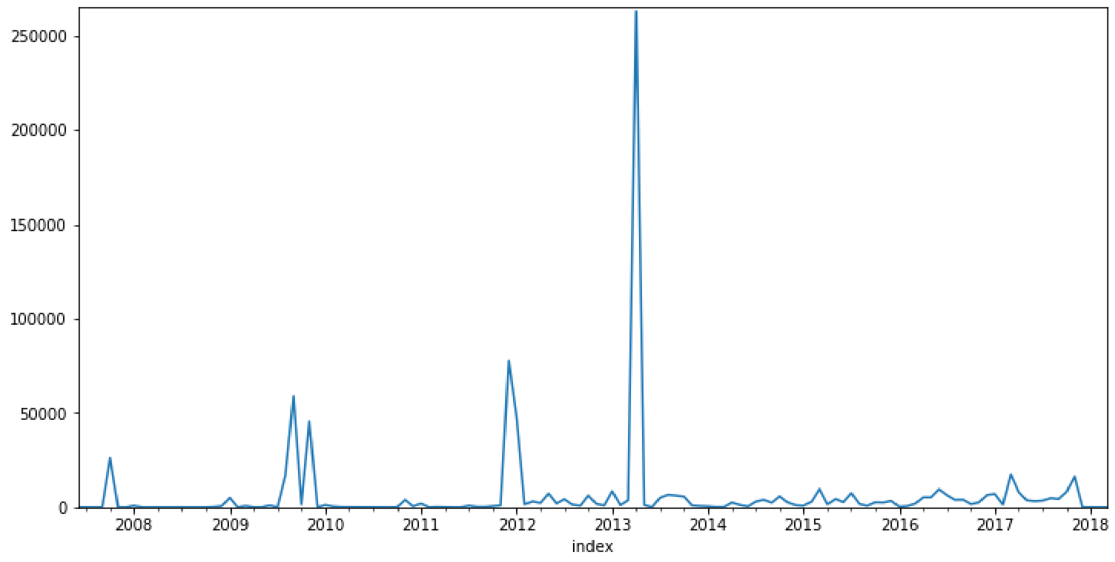
Cross-checking with the `data_key.json` file, we see everything adds up as expected: 821910 documents, covering 720267 nodes, 101406 ways, and 337 relations. The size of the collection is given by `dataSize`: 214MB. This is significantly less than the uploaded file size of ~295MB.

We also get our first glimpse of the user data. Each piece of OSM data (represented by a MongoDB document) is associated with one of 605 distinct users. Each user uploaded on average 1358 OSM data points, with a standard deviation of 14177. This indicates a high degree of variability and significant skew, presumably from the most few most active uploaders. The maximum amount of uploads for a single user is 262718, with 112 users contributing only a single piece of data. The earliest data is from September 2007, and the most recent data is from November 2017.

This variability in upload behavior hints at the origins of the dataset. To get a better understanding, I thought it would be helpful to plot data points by time of upload. To do this, I would query MongoDB using the `created.timestamp` attribute, ouput the results as JSON, then feed the JSON into Python scripts in this notebook.

Timestamp processing in MongoDB posed some challenges. Because the data was uploaded as JSON, all timestamp values are initially stored as strings. This made it difficult to handle some of the date manipulations I was attempting. After some trial and error, I decided it made more sense to parse month, day, and year values in the initial Python auditing script, and store them as additional data on each document. This made the MongoDB queries much simpler to write. The queries are stored in `queries/count_by_date.js`. Below, we see the total number of OSM data points uploaded per month since June 2006.

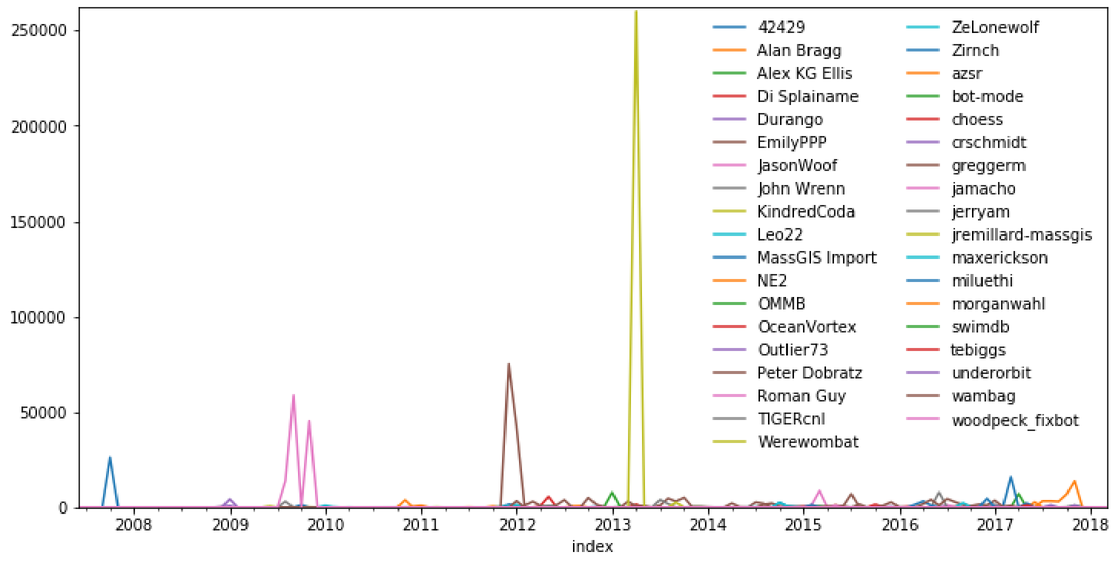
```
In [2]: import pandas as pd
        import json
        import matplotlib
        import matplotlib.pyplot as plt

        matplotlib.rcParams['figure.figsize'] = (12,6)

        with open('data/count_by_date.json','r') as f:
            jdata = json.load(f)
        data = [ { 'date' : '{0}-{1}'.format(d['_id']['year'], d['_id']['month']),
                   'count' : d['count'] } for d in jdata ]
        df = pd.DataFrame(data)

        rng = pd.date_range('6/1/2007', periods=130, freq='M')
        ts = pd.DataFrame(
            [ {'date' : "{0}-{1}".format(x.year, x.month) } for x in rng ],
            index=rng)

        merged = ts.reset_index().merge(df, how='left', on='date'). \
            set_index('index').fillna(0)
final = merged.drop('date',axis=1)
maxval = final.max().max()
final.plot(
    use_index=True, xlim=(final.index[0],final.index[-1]),
    ylim=(0,maxval + maxval*.01), legend=None)
plt.show()
```



We see a few major spikes in 2009 and 2012; and a gigantic spike in 2013, when nearly a third of the data is uploaded. There is also some general activity between 2012 and 2017, although the scale of the graph is skewed. With a sense of the overall numbers, it made sense to graph the same activity grouped by user. This would hopefully provide some insight into user behavior: who were 1-time contributors, who is behind the major spikes, etc. Since there are 605 distinct users, I decided to focus on the top contributors. The query, stored in `filtered_user_date.js`, returns data for those users who contributed at least 1000 OSM data points. The output is graphed below:

```
In [3]: import pandas as pd
        import json
        import matplotlib
        import matplotlib.pyplot as plt

        matplotlib.rcParams['figure.figsize'] = (12,6)

        with open('data/filtered_user_date.json','r') as f:
            jdata = json.load(f)
        data = [ { 'jnr' : '{0}-{1}'.format(d['_id']['year'], d['_id']['month']),
                   'user': d['_id']['user'], "count" : d['count'] }
                       for d in jdata ]
        df = pd.DataFrame(data)
```

```
rng = pd.date_range('6/1/2007', periods=130, freq='M')
ts = pd.DataFrame(
    [ {'jnr' : "{0}-{1}".format(x.year, x.month) } for x in rng ],
    index=rng)

for user, group in df.groupby('user'):
    user_data = group.drop('user',axis=1)
    ts = ts.reset_index().merge(
        user_data, how='left', on='jnr').set_index('index'). \
        fillna(0).rename(columns={'count': user})

final = ts.drop('jnr',axis=1)
maxval = final.max().max()
final.plot(
    use_index=True, xlim=(final.index[0],final.index[-1]),
    ylim=(0,maxval + maxval*.01) )
plt.legend(loc='upper right', ncol=2,frameon=False)
plt.show()
```



What's perhaps most interesting to see here is that the major spikes are all attributed to different users. We also notice that 3 of these users -- MassGIS Import, woodpeck_fixbot, and jremillard-massgis -- sound like they are operating in some sort of official capacity.

The scale of the graph is still making most of the data hard to see, so I added a widget to control it.

```
In [5]: import pandas as pd
        import json
        import ipywidgets as widgets
import matplotlib
import matplotlib.pyplot as plt

matplotlib.rcParams['figure.figsize'] = (12,6)

with open('data/filtered_user_date.json','r') as f:
    jdata = json.load(f)
data = [ { 'jnr' : '{0}-{1}'.format(d['_id']['year'], d['_id']['month']),
            'user': d['_id']['user'], "count" : d['count'] }
            for d in jdata ]
df = pd.DataFrame(data)

rng = pd.date_range('6/1/2007', periods=130, freq='M')
ts = pd.DataFrame(
    [ {'jnr' : "{0}-{1}".format(x.year, x.month) } for x in rng ],
    index=rng)

for user, group in df.groupby('user'):
    user_data = group.drop('user',axis=1)
    ts = ts.reset_index().merge(
        user_data, how='left', on='jnr').set_index('index'). \
        fillna(0).rename(columns={'count': user})

final = ts.drop('jnr',axis=1)
maxval = final.max().max()

def f(y):
    final.plot(
        use_index=True, xlim=(final.index[0],final.index[-1]),
        ylim=(0,y), legend=None)
    plt.show()

interactive_plot = widgets.interactive(
    f, y=widgets.IntSlider(min=1000,max=50000,step=1000, value=25000))
interactive_plot
```

```
interactive(children=(IntSlider(value=25000, description=u'y', max=50000, min=1000, step=1000),
```

The graph is still too busy, but we get a sense of the general hum of activity running below the major uplaod events. (omitted in PDF; please see notebook)

1.4 More stats

Having considered the general conditions of the data -- how much of it there is, and when and by whom it was produced -- it was time to dive a lttle deeper. In particular, I was interested to know the "shape" of the data. I knew, for instance, that the data consists of documents representing nodes, ways, and relations. And, I knew that all nodes have `pos` attributes (for latitude and longitude); and that all ways have `nd` attributes and all relations have `member` attributes (both relating to node ids). But while I knew pos should be single-valued for each node, I didn't how many values to expect for nd or member. And what about the tag data, which contains all of our descriptive, human-readable information? That can be found on any of the nodes, relations or ways. What was the distribution of that description?

In other words, what I wanted to know is: what do the uploaded documents look like? I know each document will have a created field containing metadata, along with the datatype field I added identifying nodes, relations, and ways. But what other fields can I expect to find, and how many of them? I used the data_key.json file to estimate some of this, but I wanted some more definite answers from MongoDB.

Writing the queries to answer these questions proved a little complicated. Because of its schemaless design, MongoDB makes you jump through a few hoops to answer collection-wide questions; especially when those questions regard the document shape, and not the data values contained in the documents. After some trial and error, I produced the queries contained in queries/deeper_stats.js. The results surprised me, challenging my expectations about the data.

There are 3 groups of queries: * The first group looks at ways and relations documents, and asks questions about the status of their nd and members fields, respectively. * The next group, nested under all_docs, looks at all the documents in the collection and generates stats regarding the number of fields on each document. The queries ignore the created and datatype fields. So, each document should have at least 1 field (1 of pos, nd, or member, for each kind of datatype), and any number greater than that should reflect associated tag data. * Building on those queries, the next group, nested under fltrd_docs, generates stats for 2 clasess of documents: those with only 1 field (ie, no tag data), and those with more than 1 field.

Looking at data/deeper_stats.json, we see some striking results. Mainly, we see that ways and relations are the more information-rich data, rather than nodes. From the OSM wiki and my preliminary evaluation, I assumed the opposite: that nodes carried most of the human description, and that ways and relations existed mostly to connect nodes. Instead we see that 683566 node documents -- ~94% of nodes and ~83% of the entire collection -- carry only created, datatype, and pos fields. There are 36701 node documents with tag data, which is not insignificant. But, nearly all ways and relations are carrying some additional description. A necessary future step will be to take that description and project it onto the nodes associated with those ways and relations.

(We also notice some interesting outliers: 1 relation associated with 7590 different nodes, and 1 way joining 987 nodes. It turns out these are the Amtrak Northeast Regional trainline, and the 100 Acre Cove in Barrington, RI, respectively.)

1.5 Revisions

By and large, the uploaded data proved to be fairly regular. The addr.postcode and addr.street values were easily cleaned up with simple spot-checking and string normalization. It made sense to validate the created.timestamp values, since they're being relied upon in the analysis and are easy enough to check programatically. (All timestamp values were clean.)

The same thinking led me to write a check for the pos data on node documents. The bounds element from the XML file has attributes listing the maximum and minimum latitude and longitude for the downloaded data. Presumably, this is generated from OSM's online map selection tool. I thought this would be a purely academic exercise, but in fact 10689 nodes have pos values outside the specified range. These all appear to be right around the bounds, however.

In exploring the data, most of the fields outside of created and addr appeared to be too inconsistent to be of much value. Often some particular field is highly descriptive, but it only appears on one or two documents, suggesting it was created by hand. But a few fields did look promising, in that they were numerous, consistent, and interpretable by me: `building`, `amenity`, `leisure`, and `landuse`. These appeared on 71896, 2872, 1461, and 969 (non-distinct) documents, respectively; and each carried largely similar data, putting a human label on what function a particular place serves.

I took some existing values from the `building` field -- `residential`, `commercial`, `industrial`, and `civic` -- and used them as top-level values for what I found in the `building`, `amenity`, `leisure` and `landuse` fields. I then mapped these top-level values into a new document field, `zone`. This is the contents of the `model_zone` function in the `audit.py` file. I didn't map all the values for those fields; I chose to map only those values I felt I could confidently categorize as residential, commercial, industrial, or civic. These are still guesses, but if I need to alter the mapping, it's easy enough to do. The result: 31643 documents with a `zone` field; 25927 `residential`, 1375 `commercial`, 1353 `industrial`, and 2988 `civic`.

1.6 Future steps

The initial processing of the dataset points to 2 immediate next steps. The first, and more daunting, is to join the ways and relations to their associated nodes; or, vice versa. As noted earlier, the greater portion of human-readable metadata is attached to ways and relations, while all the geolocation data is attached to the nodes. To begin doing more interesting analysis of the data, we'd need to merge these data points using the `nd` and `member` attributes. I believe the better move will be to push the data from ways and members onto individual nodes, rather than the other way around. This will result in repeated data and a larger database, but also I hope a more useful database.

With the geolocation data denormalized, we can move on to the next step: asking questions about Providence and its surroundings. I'm hoping a good first place to start will be the `zone` data: where are the industrial, residential, and commercial zones? How well are civic zones distributed; evenly throughout the city, or concentrated in certain areas? Where is the highest denisty of commercial activity? Where are the residential zones, relative to other zones? These upper-level classifications should be a good place to begin asking broad questions about the layout of the Providence urban area.

Stepping back form the data, some questions still remain about how the data was generated. From the user activity, I began resarching into MassGIS, which I discovered is the Masschusetts Bureau of Geographic Information (https://www.mass.gov/orgs/massgis-bureau-of-geographic-information). The bulk of the data in this dataset is provided by MassGIS, including some metadata that seems specific to their systems. The data fields `massgis`, `source`, and `tiger` all carry what looks like valuable information, but much of it is opaque to me. It might be worth contacting MassGIS to learn more about the data: how it was created, and how to use it.