Data Wrangling

May 12, 2017

1 Wrangling OSM Map Data - by Christopher Ivanovich

Written in partial fulfillment of the Udacity Data Analysis Nanodegree requirements

1.1 Area of exploration: Tokyo, using OnStreetMap opensource data obtained as an XML file

1.1.1 Getting Our Feet Wet

- Full decompressed file size of the Tokyo XML file is in excess of 4GB, used for extracting user contribution data.
- Initial checks on the full data set confirmed that all Node elements contain latitude and longtiude values immediately coercible to the float data type. It makes sense that these values would be among the most error-free, given their centrality in meeting the basic needs of a map.

1.1.2 Interesting finds from the full dataset using Python scripts

- The top contributor (out of a total of 5437 unique contributors) to the Tokyo OSM dataset has made a staggering 2,000,012 entries as of the time the dataset was downloaded (April 20, 2017). This is over 20,000 more entries than the 2nd-ranked contributor, and highly indicative of the use of a bot pulling data from other source.
- There was a total of 21,269,077 entries created by all 5437 unique users (i.e. high-level entries with XML tags of Node, Way, or Relation).

1.2 Problems with Standardization

After extracting a set of unique Node Tag keys from the map extraction OSM file, it became quite clear that there has not been much in the way of standarization applied to these data. We see that common problems involve: - **Rendering data as keys**, when that data would be better represented as values (e.g. "microbrewery" as a key, rather than as a value mapped to the universal "amenity" key; "unisex", rather than as a value of a generalized additional information key like the common "note" key). - Other "weird" keys seem to be the attempt of the contributor to show that something is present or available at a given establishment, as I have found that the value mapped to these keys is often just "yes". This explains why someone would think to write "microwave" or "telephone" as a key, and pair it with the value "yes". These keys would also make more sense as values paired with the "note" key.

- **A lack of standardization around providing names for locations**
- "name:" versus simply "name" as a key (lack of standardization). When there is a colon it is usually followed by a two-letter language abbreviation, though sometimes with a longer abbreviation (e.g. szl).
- representative examples as k,v tuples: ('name:jp', u'9̆ebb5̆e033̆0a83̆0f33̆0d13̆0a43̆0a23̆0de3̆0f33̆0b73̆0e73̆0f3'), ('name', u'5̆3434̆'), ('name', 'Muse'), ('name:en', 'MARY JUNE'). These show us that both English and Japanese-script names have been filed under a general "name" key, a "name:en" key, or a "name:jp" key.
- competing attempts also to define the different ways of rendering a Japanese name for a place, e.g. name:ja_rm, lang:ja_kana, lang:ja_rm, lang:ja_full, alt_name:ja (I would argue that the Japanese name should not be an alternative name in Japan).
- and oddly, 'name:etymology:wikidata', because for some reason someone thought to provide scientific details for a particular tree species, and rather than give the source of those details as separate entries with a key like 'reference', they cobbled together that particular abomination.
- **A lack of standardization around "cuisine" tag entries:** ('cuisine', u'9̆9035̆b50'), ('cuisine', 'vegan'),('cuisine:ja', u'4̆e327̆13c'). As before, the presence of unicode characters have been confirmed to be primarily Japanese language entries.

1.3 Cleaning some of these keys in preparation for creating an SQL database

Based on further inspection of the data using several different custom scripts, I believe sensible cleaning tasks include: - replacing all instances of "cuisine:xx" with "cuisine", as extracting a full set of cuisine-bearing key, value pairs has shown that the form "k=cuisine", "v=descriptor" is the prevailing norm, and assigning "xx" to be the value descriptor (coerced to "japanese" where "xx" is "ja"). - replacing all key instances of "name:xx" with "xx", creating a new tag-type called "name" for all "name" entries, and confirming that a given name value matches the language it is filed under.

1.4 What's in a "name"?

Upon first building the database, 6096 entries in the ways_tags table had "name" as a key, while 9206 had it as a tag type. The code provided in the data.py file in the course is aimed at setting anything before the first appearance of a colon ":" as a tag type. In these 6096 instances, the conflict seemst to have arisen from the XML where `<tag k='name' />`, i.e. the key has no colon. Further, the same problem was confirmed to arise in the nodes_tags table. In these (numerous) cases, the shape_element method provided writes "name" as a key, and lists the tag type as "regular", the default tag type.

Clearly, it is strange to have almost half of the entries list "name" as a key when the other half lists it as a tag type. Furhter, as noted previously, some tag values for names (like 'szl and 'Latn') fail to meet the ISO 639 standard for abbreviating languages. To remedy two birds with one stone, I make use of the **langdetect** library to identify the languages used in the value fields of the tag elements, with the help of an additional method and a slight rewrite of the data.py shape_element method (see next section for the rewrite code):

```
from langdetect import detect
def langtype(v):
    try:
        lang = detect(unicode(v)[0])
        return lang
    except:
    #as there are frequent errors involving "empty values", which were often numerical
    #strings
        return v
```

It needs to be noted that this library fails on multilingual entries like **" Hello"**. I somewhat get around this by limiting the detect method to only the first character of the unicode string, as most multilingual entries seem to be of this ordered form.

1.5 "cuisine" travails

Also, as noted previously, we encounter a similar problem in auditing cuisine entries. If "cuisine" as a key in the XML has no colon followed by some further information, it gets assigned in the SQL table as a key rather than as a tag type, and "regular" becomes the defining tag type. Where a 2-letter abbreviation is provided for the cuisine type, cuisine becomes the tag type, creating a conflict between these two formats (e.g. 'ja', 'some_description', 'cuisine' versus 'cuisine', 'description', 'regular' as field entries in the table). We elect to standardize this by forcing the key type to "cuisine" if there is a colon present, and overwriting the "v" value to be what comes after the colon (most often it is "ja" for Japanese). Below is a sample of changes made to the shape_element function in data.py that accomplish this, and also the the changes that accomplished the task above for names and language values.

```
def shape_element("a ton of arguments"):
    ...some code...

 if element.tag == 'node':
    node = element.attrib
    node_attribs = {k:v for k, v in node.items() if k in node_attr_fields}

    node_tags = element.findall("tag")
    for elem in node_tags:
     #more code omitted, see full data.py file
        if 'name' in k: #to handle the problems with the "name" entries
            ttype = "name"
            k = langtype(v) #a call to our new helper method
        elif 'cuisine:' in elem['k']: #for our cuisine troubles
            v = 'japanese' if elem['k'][colpos+1:] == 'ja' else elem['k'][colpos+1:]
            #if we see "cuisine:ja", assign value as "japanese", otherwise, assign what follow
            k = 'cuisine'
            ttype = "regular"
        else:
            ttype = default_tag_type if ":" not in elem['k'] else elem['k'][:colpos]
        tags.append({'id':id, 'key':k, 'value':v, 'type':ttype})
    return {'node': node_attribs, 'node_tags': tags}

 #changes are repeated for the "way tag" elements
```

And this seems to have done the trick. At least as far as names and cuisine:ja entries are concerned...

1.6 Basic dataset info

A quick note: Initially I attempted this project with a smaller subset of the data, as instruced, using an extract of about 140 MB. But, figuring "Hey, why not?", I took on the added challenge of using the full dataset, which, apart from writing the csv files, hasn't been as processor-consuming as I'd feared. I suspect this speaks to the power of relational databases rather than my coding prowess.

1.6.1 File sizes

```
tokyo_japan.osm ......... 4.03 GB
tokyofull.db .......... 2.28 GB
nodes.csv ............. 1.48 GB
nodes_tags.csv ........ 65.64 MB
ways.csv .............. 164.50 MB
ways_tags.csv ......... 290.38 MB
ways_nodes.csv ......... 519.62 MB
```

1.6.2 Number of nodes

```
sqlite> SELECT COUNT(*) FROM nodes;
18405479
```

1.6.3 Number of ways

```
sqlite> SELECT COUNT(*) FROM ways;
2853362
```

1.7 Querying our *cleaner* database

1.7.1 Unique Users

First, I query the database to obtain the number of unique users. For the added functionality of Python 2.7's string encoding/decoding functions I do this using the sqlite library and run scripts from within the Spyder IDE.

```
query1 = """SELECT COUNT(distinct e.user)
FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways) e;"""
```

This yields a total of **5437 unique users** across the nodes and ways tables.

1.7.2 Top 10 contributors

To obtain the top 10 contributors, I make use of code taken from the sample SQL project provided in the instructions:

```
query2 = """SELECT e.user, COUNT(*) as num
FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways) e
GROUP BY e.user
ORDER BY num DESC
LIMIT 10;"""
```

the results of which are:

```
              0        1
0   futurumspes  2000012
1       Tom_G3X  1838115
2         Ryo-a  1014523
3      nyampire   888944
4         watao   824279
5      u_kubota   745968
6   Gravel Crew   667555
7      kurauchi   667081
8       yoshitk   651999
9  u_kubota_imp   410133
```

1.7.3 Cuisine finds

```
query3 = '''
SELECT count(*)
FROM nodes_tags #and again, ways_tags
WHERE key = "cuisine";'''
```

yields 13804 and 653 cuisine entries from the nodes_tags and ways_tags tables. Restricting the search to just entries `sql WHERE key = "cuisine" AND value LIKE "ja%"` yields a mere 3360 and 196 restaurants explicitly coded as being Japanese, respectively.

The top ten restaurant entries:

```
              0     1
0      japanese  3285
1   coffee_shop  1450
2       chinese  1223
3         sushi   811
4        noodle   793
5         ramen   723
6        burger   718
7       italian   616
8     beef_bowl   298
9  noodle;ramen   270
```

We see that noodle appears essentially three times in this table--much as I love a big bowl of ramen, I wouldn't insist it deserves separate key-level classification from, say, udon or soba. Furthermore, should the "Japanese" key coverage be expanded to include noodle, ramen, and noodle;ramen? Does it make sense to label restaurants by ethnicity first, and then to provide further details in another tag or in a third attribute?

Also, I believe these values leave out a fair number of restaurants, as I've seen things like "beaf-bawl" and "barger" which have a large impact on our ability to classify cuisine values. Further, "noodle", "sushi", and "ramen" should all be classed under "japanese", if this dataset is to have greater standardization--so too should sushi and beef_bowl, which thus suggests that six of the top ten entries could be folded into the japanese entry.

1.7.4 Submission activity by year

```
entries_by_year = {}

for year in range(2007,2018):
    query4 = '''
    SELECT COUNT(timestamp)
    FROM nodes
    WHERE timestamp LIKE "{0}%"
    ;'''.format(str(year))

    c.execute(query4)
    rows = c.fetchall()
    entries_by_year[year] = rows[0][0]
```

yields the following pandas DataFrame results (values begin at 2007): `python 2007 4045 2008 41986 2009 85112 2010 2026753 2011 1766275 2012 2694144 2013 2419404 2014 2277371 2015 2677708 2016 3176941 2017 1235740` Repeating the process for the Ways table reveals: `python 2007 16 2008 3858 2009 9126 2010 56325 2011 265816 2012 357937 2013 385704 2014 402609 2015 428042 2016 662876 2017 281053` Submission activity for both data types seem to have had high points in 2016, which surprised me given the growing strength of competing projects, like Maps.me.

1.8 Additional Exploration

1.8.1 Top 10 Amenities

Using an only slightly modified query with key = "amenity", we get some more interesting top 10 results:

```
                  0      1
0           parking  38832
1            school   6333
2   place_of_worship   2360
3         grave_yard   1527
4        restaurant   1242
5   bicycle_parking   1188
6   public_building    799
7      kindergarten    757
8           toilets    532
9          hospital    528
```

It is tempting to draw inferences about Japanese culture based on the 3rd and 4th place entries (I'll spare you unnecessary masquerading as an anthropologist; although in my memory, graveyards are frequently located on shrine and temple grounds), but it is also important to note that restaurants come in at 5th place, meaning that not all food entries were filed under some variant of "cuisine"--yet another problem for another day.

Also, clearly the 10th entry is nonsense (only 532 toilets in all locations represented in the Nodes data in the world's (formerly now?) largest city?).

Finally, kindergarten could arguably be lumped in with schools.

1.8.2 The probable role of "Power Users" and Messy Entries

Two major areas for improvement of the auditing process with this dataset lie in the proliferation of keys describing the same general category of something under many guises (e.g. ramen, beef bowl, noodle for Japanese cuisine), and in the frequent misspellings of user-generated input of all stripes (e.g. tranportationr, beaf_bawl).

A quick look at the input from three of our top contributors, **who I define as "power users" with entries greater than 1 million**, suggests that the fault lies not with programmatically generated input (Nodes and Ways data), but with human-entered Tags data. Why? Behold!

```
SELECT n.user, count(*)
FROM nodes AS n JOIN nodes_tags AS nt ON n.id = nt.id
WHERE n.user = "futurumspes" OR n.user = "Tom_G3X" OR n.user = "Ryo-a"
GROUP BY n.user;
```

```
             0      1
0        Ryo-a  21164
1      Tom_G3X     32
2  futurumspes  97410

#repeated for ways_tags
             0    1
0        Ryo-a  215
1  futurumspes  127
```

In the tags dataests these three power users made far fewer tags contribtions as opposed to Nodes and Ways contributions, indicating that they were responsible primarily for mapping out the basic structure of the data, while leaving the nitty-gritty of restaurant names and such to less technically-inclined users. It is almost inconveiably to me that they made as many contributiosn as they have without scanning map data from elsewhere (Tom_G3X in particular did little tagging) which in turn suggests that the observed errors and poor standards are likely at least partially the fault of hand-entered data.

1.9 Conclusion

To conclude this lengthy report and exploration of the full Tokyo.osm dataset, it seems not at all surprising that there is a lot of messy data here. The amazing thing to me seems to be that a few power users were able to do a solid job generating the basic infrastructure of the map as part of an opensource project contribution.

Though it may be a belabored point, this dataset could really benefit from the tightening of standards for tags' key-value pairs, with perhaps an additional field for more detailed description. The current method seems to consist of adding multiple tags for a single location, which invites a great deal of redundancy and possible confusion, as we've seen.