Markdown_for_OSM_Project

September 12, 2016

1 Data Wrangling on OpenStreetMaps Data

1.1 City of Oxford - United Kingdom

11 September 2016

by Dale Millar - Udacity Data Analyst Nanodegree - Project 3
dale@dalemillar.com

1.2 Overview of the Data

I chose the English university city of Oxford for my study - I know it well and was really interested to study how OSM had captured the data. It's a quintesentially English city and I thought it would also be fun to see how it contrasted with some of the US cities discussed in the course such as Chicago and Charlotte. It also became clear during the study that there are some pretty gigantic holes in the data and this is something that I'd like to contribute to fixing moving forward.

Map Area

- https://mapzen.com/data/metro-extracts/your-extracts/b4c276e5c44c

File Sizes

- oxford_england.osm 64.6 Mb
- street8.db 39.4 Mb
- nodes.csv 22.4 Mb
- nodes_tags.csv 1.6 Mb
- ways.csv 2.7 Mb
- ways_tags.csv 5.3 Mb
- ways_nodes.csv 9.4 Mb

1.3 Problems with Data

1.3.1 Incorrectly Formatted Post Codes

As is the norm, UK post codes have a unique, set format, which if violated, make them useless for both the Royal Mail (UK postal service) and for satellite navigation systems which typically offer an option to set destination using post codes.

Post code format should be:

```
(2 Letters) + (1 or 2 Numbers) + (1 Number) + (2 Letters)
```

An initial investigation of the sample.osm file indicated multiple instances of incorrectly entered post-codes as can be seen from this partial sample generated with the python script postcode_script.py on the oxford.osm file.

```
In [ ]: OX3 -- Incorrect Post Code
        OX2 -- Incorrect Post Code
        OX29 -- Incorrect Post Code
        OX29 -- Incorrect Post Code
        OX5 -- Incorrect Post Code
        OX5 -- Incorrect Post Code
        OX4 1AS
        OX2 -- Incorrect Post Code
        OX1 3EX
        OX1 3EX
```

I created a helper function called post_coder which used a regex expression to identify incorrect post codes. post_coder() is called from the secondary_tags() function which in turn is called by shape_element.

One of the areas for future work that I am interested in is finding a way to use node tag lat and lon attributes to automatically fix corrupt post codes - more of this later, but for now, I simply replaced the corrupt post codes with empty strings. My rationale for this is that an incorrect post code is worse than useless, potentially involving incorrect mail delivery or mis-direction from SatNav, whilst no post code will still allow the Royal Mail to deliver correctly and other location mechanism (lat/lon, address) to be used with SatNavs.

As a result of this corrective action, I was able to clean up all of the offending codes in the data and successfully import them into the sqlite data base. A query looking at post codes in both nodes_tags and ways_tags reveals that 340 invalid post codes were corrected and that other codes look normal (LIMIT 10 used here, but by inspection of the complete query, no incorrect post codes were found).

```
In [ ]: OX29 -- Incorrect Post Code
        OX5 -- Incorrect Post Code
        OX5 -- Incorrect Post Code
        OX4 1AS
        OX2 -- Incorrect Post Code
        OX1 3EX
        OX1 3EX
        OX1 5AS
        OX1 5AS
        OX2 0BU
        OX1 -- Incorrect Post Code
        OX1 3UJ
        OX1 2HB
```

```
In [ ]: cur = conn.cursor()
        cur.execute('SELECT value,COUNT(*) as Count \
        FROM (SELECT id,value FROM ways_tags WHERE key = "postcode" or key = "posta
```

```
        SELECT id,value FROM nodes_tags WHERE key = "postcode" or key = "postal_cod
        GROUP BY value \
        ORDER BY Count DESC LIMIT 10')
        all_rows = cur.fetchall()
        print('Value |     Count ')
        pprint(all_rows)
        conn.close()
```

```
In [ ]: Value |     Count
        [(u'', 340),
         (u'OX4 3JS', 48),
         (u'OX4 3LJ', 48),
         (u'OX4 3EH', 46),
         (u'OX4 2BQ', 45),
         (u'OX4 3PF', 45),
         (u'OX4 4QS', 45),
         (u'OX4 3QS', 44),
         (u'OX4 3SR', 44),
         (u'OX4 1JT', 43)]
```

As a matter of interest, there were no post codes thrown up from outside the Oxford area - this is partly, I imagine, down to the strength of the OX notation which provides immediate recoginition that it is an Oxfordshire post code and prevents data entry errors.

1.3.2 Abbreviated Street Names

It is common for street names to be abbreviated such as:

- Road - Rd
- Street - St
- Avenue - Ave
- Drive - Drv

I used the Python script abbrev_script.py to examine the oxford.osm data.

I wanted to eliminate these abbreviation and at the same time avoid losing any address data by dropping unknown abbreviations.

I built an expected list of last words of address attributes as shown below. I specifically looked for the problem abbreviations which I listed in the way_dict dictionary (again see below) inside the last_word_address_fixer() helper function which was called from the secondary_tags() function. If the last word of the address was not in expected, then last_word_address_fixer() would either insert a correct word from way_dict or return the unadulterated address.

```
In [ ]: expected=['Road','Lane','Street','Close','Way','Green',\
                  'Avenue', 'Turn', 'Mead', 'Drive']
        way_dict={'rd':'Road','st':'Street','ave':'Avenue','drv':'Drive'}
```

The following query shows that the sqlite data is clean (again I've used LIMIT 10 to save space, but inspection provides clear evidence that all common abbreviations have been cleaned up.

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT value,COUNT(*) as Count \
        FROM (SELECT id,value FROM ways_tags WHERE key = "street" \
        or key = "naptan:Street" UNION ALL \
        SELECT id,value FROM nodes_tags WHERE key = "street" \
        or key = "naptan:Street") post1 \
        GROUP BY value \
        ORDER BY Count DESC LIMIT 10')
        all_rows = cur.fetchall()
        print('Street Name |     Count ')
        pprint(all_rows)
        conn.close()

In [ ]: Street Name |     Count
        [(u'', 9250),
         (u'Herschel Crescent', 207),
         (u'Cumnor Hill', 136),
         (u'Southfield Park', 124),
         (u'Westbury Crescent', 75),
         (u'Rose Hill', 70),
         (u'The Glebe', 65),
         (u'The Grates', 59),
         (u'Canning Crescent', 44),
         (u'Demesne Furze', 43)]
```

This is all very satisfactory until one realizes that there are 9,250 empty strings where there should be street addresses. If that's not bad enough, it really is odd that Herschel Crescent (which is a small street) has so many mentions, and other major roads apparently so few. There are probably major ommisions here and this is another area which would be interesting to address moving forward.

1.4 Data Overview and Additional Ideas

Number of Nodes

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(*) FROM nodes \
        ')
        all_rows = cur.fetchall()
        print('Total Number of nodes:')
        pprint(all_rows)

        conn.close()

        Total Number of nodes:
        [(274598,)]
```

Number of Ways

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(*) FROM ways \
        ')
        all_rows = cur.fetchall()
        print('Total Number of ways:')
        pprint(all_rows)

        conn.close()

        Total Number of ways:
        [(46615,)]
```

Number of Unique Users

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(DISTINCT(f.uid)) \
        FROM (SELECT uid FROM ways UNION ALL SELECT uid FROM nodes) f\
        ')
        all_rows = cur.fetchall()
        print('Total Number of Unique Contributors:')
        pprint(all_rows)
        conn.close()

        Total Number of Unique Contributors:
        [(563,)]
```

Top 10 Contributing Users

```
In [ ]: cur.execute('SELECT user,COUNT(*) as count \
        FROM (SELECT user FROM ways UNION ALL SELECT user FROM nodes) f \
        GROUP BY user \
        ORDER BY count DESC LIMIT 10\
        ')
        all_rows = cur.fetchall()
        print('Top 10 Contributors:')
        pprint(all_rows)
        print

        Top 10 Contributors:
        [(u'Andrew Chadwick', 39009),
         (u'craigloftus', 37095),
         (u'GordonFS', 28930),
         (u'Max--', 19108),
         (u'Richard Mann', 17052),
```

```
 (u'Robert Whittaker', 16890),
 (u'Govanus', 16104),
 (u'EdLoach', 15243),
 (u'extua', 11551),
 (u'mprhode', 7922)]
```

Number of Contributors with 1 Post Only

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(*) FROM (SELECT f.user,COUNT(*) as count \
        FROM (SELECT user FROM ways UNION ALL SELECT user FROM nodes) f \
        GROUP BY f.user HAVING count = 1)')
        all_rows = cur.fetchall()
        print('Once Only Contributors:')
        pprint(all_rows)
        print

        conn.close()

        Once Only Contributors:
        [(120,)]
```

1.5 Additional Ideas

Ommissions in the Data The most glaring problem found in terms of completeness was the lack of street name information in ways_tags. The value field was left blank 9,250 times out of a total of 11,234 street names. This is obviously something that needs to be addressed - it may be possible to use another db to do this, the most obvious one being the Royal Mail web site at (http://www.royalmail.com/find-a-postcode). Single post codes normally refer to single streets - if one can associate a ways_tags post code with a street name via it's parent way tag, then it should be possible to automatically insert street names using a join with the Royal Mail database.

Just a couple of issues to note:

- I said that one street is normally referred to by one post code - unfortunately there are exception when small cul-de-sac's or side roads are too small to have a separate code and are accommodated under the 'mother street' post code. This will result in occasional errors in inserting false street names - indeed these small streets may never be named in OSM, or worse still, the name of the cul-de-sac ends up being used for the main street. There is a reall value judgement to be made here.
- The second problem is one of licensing - the post code database is a proprietary piece of intellectual property owned by the Royal Mail. In a now privatized delivery environemnt, there is every chance that this hard-won data set will not be made available to OSM without licensing payments and this is of course an antithesis of what OSM is all about.

The other option of course is to work back from the ways_tags street name to the node associated with that way. At that point, one has lat and lon data and investigation of web sites such as https://www.doogal.co.uk allow one to convert post codes into lat and lon. Now while this

suggests to me that there must be a table in the db with combinations of house numbers and post codes paired to lat/lon co-ordiantes, it looks to me that every single API for similar sites only has look-up for post codes to give lat and lon and not in the opposite direction. I can't see how this could be anything other than coincidence and the result of more common use cases, but it is worth flagging as a potential problem for OSM data insertion of post codes.

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(*) as Count \
        FROM (SELECT id,value FROM ways_tags WHERE key = "street" \
        or key = "naptan:Street" UNION ALL \
        SELECT id,value FROM nodes_tags WHERE key = "street" \
        or key = "naptan:Street") post1 \
        ')
        all_rows = cur.fetchall()
        print('Count ')
        pprint(all_rows)
        conn.close()

        Count
        [(11234,)]
```

I'm a big believer in the use of buses in urban areas to cut down on pollution and congestion. A key to getting people on buses and out of their cars is to provide an Uber-like level of convenience. Open an app on your smart phone, plug in where you want to get to and be guided to the nearest appropriate bus stop. As it stands, the National Public Transport Access Nodes (naptan) data set is uploaded into the UK OSM data set, but right now does not have lat lon co-ordinates. Future work could utilize node lat lon data to ensure every bus stop had proper co-ordinates and that they formed a way that could superimpose bus routes onto OSM. This could be used to develop an app that could guide users to bus stops and map their journeys, providing convenience and greater utilisation of the bus.

There are some problems with linking naptan bus stop locations with nodes; naptan location codes could easily be mis-matched with lat lon node information which has not been validated. This would be disastrous for the app which would of course rely on accurate geographic data.

A way around this would be to use GPS at each of the bus stops (there are no more than one hundred so one guy on a bike could do it in a day) and build a new table that linked naptan location codes to actual GPS data. That way, all doubt is eliminated. The bus stop layout in the city is shown at https://www.oxfordshire.gov.uk/cms/sites/default/files/folders/documents/roadsandtransport/publictrans

The other major issue with the app would be the locating of the user when she wants to ride a bus. Obviously there are plenty of API's that one can use to provide geographic data on cell phone location, but this then has to be interfaced to OSM data to provide a calculation of how to get from the user's location to the nearest relevant bus stop. This mash-up of functionality is obviously possible (eg Uber), but there are obvious licensing issues that need to be overcome.

```
In [ ]: cur = conn.cursor()
        cur.execute('SELECT key, COUNT(*) as count \
        FROM (SELECT key FROM ways_tags UNION ALL SELECT key FROM nodes_tags) f \
```

```
WHERE key LIKE "%bus%" \
GROUP BY key \
ORDER BY count DESC LIMIT 8 \
')
all_rows = cur.fetchall()
print('Total Number of bus-related mentions mentions:')
pprint(all_rows)

conn.close()

Total Number of bus-related mentions mentions:
[(u'naptan:PlusbusZoneRef', 676),
 (u'naptan:BusStopType', 92),
 (u'bus_routes', 81),
 (u'bus:forward', 49),
 (u'bus:backward', 21),
 (u'tourist_bus', 18),
 (u'bus', 9),
 (u'tour_bus_only', 9)]
```

Finally, there are really very few people contributing to OSM in Oxford (563 total with 120 once-only contributors) - a city with over 22,000 top-quality students. Indeed, it appears that 65% of nodes and ways data has been input by the top 10 contributors, indicating very low user participation. We have discussed major holes in the data around street names and bad post codes, quite apart from the potential to drive new start-up activity around transport applications, restaurant finders etc etc. I cannot help but think that there is an opportunity to vastly improve the data by publicizing the opportunity in the under/post graduate community and encouraging individuals to really get involved in upping the amount of contributors to the data set.

Having said that, it's worth pointing out that there are almost certainly some drawbacks in this wider approach:

- How does one maintain data quality with a large increase in contributors? With only a dozen or so major contributors, it is probably fair to assume that they are a dilligent bunch who take OSM seriously along with the quality of the data they input. If a much wider group of people start to input data, one obviously exposes OSM to the danger of much poorer quality of information. There is a balance to be drawn; new user input probably would have to be vetted for quality initially and this governance would obviously restrict the number of new contributors OSM could handle.

- With a wider net cast, the chance of bringing in malevolent contributors becomes much higher. This type of person is much harder to control - she will be highly compliant initially and then, when left to her own devices will start to enter false data, malware or APT; highly disruptive to a volunteer organization. Personal vetting and analysis of social media is a good way to identify this kind of person, but the need to do it brings a considerable overhead to OSM

On balance, the need to obtain more contributors has to be off-set against the overhead on governing and controlling these new users - it is very much a double-edged sword.

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(DISTINCT f.user) as count \
        FROM (SELECT user FROM ways UNION ALL SELECT user FROM nodes) f \
        ')
        all_rows = cur.fetchall()
        print('Total Number of Contributors:')
        pprint(all_rows)
        print

        cur.execute('SELECT user,COUNT(*) as count \
        FROM (SELECT user FROM ways UNION ALL SELECT user FROM nodes) f \
        GROUP BY user \
        ORDER BY count DESC LIMIT 10\
        ')
        all_rows = cur.fetchall()
        print('Top 10 Contributors:')
        pprint(all_rows)
        print

        cur.execute('SELECT COUNT(*) as count \
        FROM (SELECT DISTINCT(id) FROM ways UNION ALL SELECT DISTINCT(id) FROM node
        ')
        total_number = cur.fetchall()
        print('Total Number of Nodes and Ways:')
        pprint(total_number)
        print
        conn.close()

        print ("Percentage of Total Contributions from Top 10 Contributors")
        total_number_s = [x[0] for x in total_number]
        total = total_number_s[0]

        for name in all_rows:
            percent = float(name[1])/float(total)
            print name[0] + ' ' + str(round((percent)*100,2)) + '%'

        #65% of all data put in by top 10 contributors.

        conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT COUNT(*) FROM (SELECT f.user,COUNT(*) as count \
        FROM (SELECT user FROM ways UNION ALL SELECT user FROM nodes) f \
        GROUP BY f.user HAVING count = 1)')
        all_rows = cur.fetchall()
        print('Once Only Contributors:')
        pprint(all_rows)
        print
```

```
conn.close()

Total Number of Contributors:
[(563,)]

Top 10 Contributors:
[(u'Andrew Chadwick', 39009),
 (u'craigloftus', 37095),
 (u'GordonFS', 28930),
 (u'Max--', 19108),
 (u'Richard Mann', 17052),
 (u'Robert Whittaker', 16890),
 (u'Govanus', 16104),
 (u'EdLoach', 15243),
 (u'extua', 11551),
 (u'mprhode', 7922)]

Total Number of Nodes and Ways:
[(321213,)]

Percentage of Total Contributions from Top 10 Contributors
Andrew Chadwick 12.14%
craigloftus 11.55%
GordonFS 9.01%
Max-- 5.95%
Richard Mann 5.31%
Robert Whittaker 5.26%
Govanus 5.01%
EdLoach 4.75%
extua 3.6%
mprhode 2.47%

Once Only Contributors:
[(120,)]
```

1.6 Additional Data Exploration

Top 10 Bank ATM Quantities

```
In [ ]: cur = conn.cursor()
        cur.execute('SELECT nodes_tags.value,COUNT(*) as count \
        FROM nodes_tags \
        JOIN (SELECT id FROM nodes_tags WHERE value = "atm") t \
        ON nodes_tags.id = t.id \
        WHERE nodes_tags.key = "operator" \
        GROUP BY nodes_tags.value \
        ORDER BY count DESC \
```

```
LIMIT 10 \
')
all_rows = cur.fetchall()
print('Top 10 Banks ATMs:')
pprint(all_rows)
conn.close()
```

```
Top 10 Banks ATMs:
[(u'Barclays Bank PLC', 5),
 (u"Sainsbury's", 5),
 (u'NatWest', 4),
 (u'HSBC', 3),
 (u'Lloyds Bank', 3),
 (u'Barclays', 2),
 (u'Co-operative Bank', 2),
 (u'RBS', 2),
 (u'Royal Bank of Scotland', 2),
 (u'Santander', 2)]
```

Top 10 Fast Food and Restaurant Cuisines (NOT Fish and Chips!!)

```
In [ ]: conn = sqlite3.connect(sqlite_file)
cur = conn.cursor()
cur.execute('SELECT nodes_tags.value,COUNT(*) as count \
FROM nodes_tags \
JOIN (SELECT id FROM nodes_tags WHERE value = "fast_food" or value = "resta
ON nodes_tags.id = t.id \
WHERE nodes_tags.key = "cuisine" \
GROUP BY nodes_tags.value \
ORDER BY count DESC \
LIMIT 10 \
')
all_rows = cur.fetchall()
print('Top 10 Fast Food and Restaurant "Cuisines":')
pprint(all_rows)
conn.close()
```

```
Top 10 Fast Food "Cuisines":
[(u'chinese', 19),
 (u'indian', 14),
 (u'sandwich', 12),
 (u'italian', 9),
 (u'fish_and_chips', 8),
 (u'burger', 6),
 (u'pizza', 6),
 (u'thai', 6),
 (u'asian', 3),
 (u'kebab', 3)]
```

Top 10 Public House Names (UK Bars have Old Fashioned Names)

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT nodes_tags.value,COUNT(*) as count \
        FROM nodes_tags \
        JOIN (SELECT id FROM nodes_tags WHERE value = "pub") t \
        ON nodes_tags.id = t.id \
        WHERE nodes_tags.key = "name" and nodes_tags.value != "photograph" \
        GROUP BY nodes_tags.value \
        ORDER BY count DESC \
        LIMIT 10 \
        ')
        all_rows = cur.fetchall()
        print('Top 10 Pub Names:')
        pprint(all_rows)
        conn.close()

        Top 10 Pub Names:
        [(u'Red Lion', 3),
         (u'The Chequers', 2),
         (u'The Fox', 2),
         (u'The Kings Arms', 2),
         (u'The Plough', 2),
         (u'The White Hart', 2),
         (u'Three Horseshoes', 2),
         (u'(former) Woodstock Arms', 1),
         (u'All Bar One', 1),
         (u'Brewin Dolphin', 1)]
```

Top 10 Amenity Mentions (Oxford is renowned for it's cycling culture)

```
In [ ]: conn = sqlite3.connect(sqlite_file)
        cur = conn.cursor()
        cur.execute('SELECT value, COUNT(*) as Count \
        FROM nodes_tags \
        WHERE key = "amenity" \
        GROUP BY value \
        ORDER BY Count \
        DESC \
        LIMIT 10\
        ')
        all_rows = cur.fetchall()
        print('Top 10 Amenity Mentions:')
        pprint(all_rows)
        conn.close()

        Top 10 Amenity Mentions:
```

```
[(u'bicycle_parking', 482),
 (u'post_box', 332),
 (u'bench', 184),
 (u'telephone', 142),
 (u'pub', 98),
 (u'restaurant', 95),
 (u'cafe', 83),
 (u'fast_food', 72),
 (u'parking', 66),
 (u'atm', 54)]
```

1.7 Conclusions

The initial inspection of the OSM data for Oxford indicates that it is imcomplete and in need of considerable enhancemnt.

It is not uniform insofar as there are clusters of rich data (in Herschel Crescent for example) and areas where there appears to be very sparse data. This indicates very serious areas of incompleteness.

The data that is avaiable does appear to be accurate, but this is using my own local knowledge and should not be used as an engineering assessment of data quality.

The data has been well cleaned as it appertains to this project.

```
In [ ]:
```