OpenStreetMap project submission

Project overview

The goal of this project is to work with a selected geographical area of interest from the openstreetmap.org database to audit data definition problems, clean the problematic data, import the cleaned data into an SQL database, and run suitable queries to analyze the data in order to gain insight. In short, this is a practical exercise in data wrangling, which is perhaps the most vital and the most time-consuming part of most data analysis projects.

Overview of dataset

The dataset was downloaded from https://www.openstreetmap.org in the XML OSM format. I chose the geographical area defining the area in and around where I live. This includes the Borough of Elmbridge and surrounding areas – Walton-on-Thames, Weybridge, Hersham, Esher, and parts of other adjacent towns. The data boundaries are as follows:

Min. longitude: -0.5294, Max. longitude: -0.3360

Min. latitude: 51.3374, Max. latitude: 51.4216

The uncompressed OSM file was 55,939 KB in size. Based on my PC performance, this size was sufficient for the file to be opened in the notepad++ editor, perform search operations on tags, and ensure relatively quick Python program performance.

A quick manual browse through the dataset revealed that it is (a) far from complete! (b) There are a large number of tag types used for nodes and ways (c) The same kind of geographical feature occurring more than once, is defined in varying levels of detail. (d) The quality of data for the major address-related tags is generally high.

Auditing process

The auditing process began with extracting the distinct tag types within the chosen OSM data. This revealed over 300 unique tags for nodes and ways. A few key tags were chosen for auditing, namely: country, flat number, house name, house number, postcode, street name, interpolation, phone, website, source name and type. These tags were chosen as it would be easy to recognise invalid formats and in some cases, invalid values. A separate Python function was written to audit each of the chosen tag types.

Auditing was carried out iteratively. At first, the audit function simply captured and printed out distinct values for each of the tags. Then, I carried out a manual inspection of the output to isolate correct values from incorrect ones, and added the definition of correct values to the corresponding tag's audit. Regular expressions were used to isolate correct phone, house number, flat number, website, and postcode values from incorrect ones. Then, the audit was run again, this time narrowing down the list of invalid values. Further manual inspection revealed that some invalid values could be programmatically cleaned while others could be split up into multiple valid values, or in a couple of cases, the value would be correct if it was identified by the house number tag rather than the house name tag.

Problems encountered during audit

Here is a list of problems encountered by tag type:

- Flat number – Some flat numbers were identified by the "addr:flat" tag and others by the "addr:flatnumber" tag.
- House name – Some house names were identified by the "addr:housename" tag and others by the "addr:name" tag. Some house names of the format <number> - <number> were incorrectly tagged as house name when they are actually house numbers. Some house names of the format "Flat" <number> were incorrectly tagged as house name when they are actually flat numbers. Some house names of the format <number> <name> combined the house number and house name, when two different tags should have been used, one for house number and one for house name.
- City – There are some discrepancies in city names, specifically spelling errors and case.

- Street type – Interestingly, street names in the UK cannot be identified by defining regular expressions for street name endings. For instance, “Hillcrest”, “Heathside” are all valid street names. Hence, we must rely on some manual inspection to isolate invalid street names. Street names were found to be mostly correct, with the exception of “ROAD” and “Rd” used for “Road”. This would be addressed by updating the street mapping from the invalid to valid street name ending.
- Postcode – Postcodes were all found to be in valid format, but this took auditing postcodes multiple times to be able to update the regular expression to correctly isolate valid and unexpected postcodes.
- House number – The issues here center on the use of space, comma and semi-colon to specify multiple house numbers. Two invalid values: “Council Offices” and “Padley” were manually identified as being incorrect values for house number, which should be tagged as house name”.
- Phone – Problems with phone numbers are mainly to do with extraneous characters such as spaces, parentheses and period within it which must be cleaned. In some cases, multiple phone numbers were specified, each of which had to be inspected. Further, some phone numbers contained country codes, some did not. Others included the country code as well as the trailing 0. These had to be standardized.
- Website – Only around 10 OSM entries contained website information. Some contained the “http” prefix, others did not. I also tested each link programmatically to see which links were actually valid. Invalid links were recorded.
- Type – No problems encountered. All data valid per OSM specification.
- Country – No problems encountered. All data valid per OSM specification. However, this information is redundant and need not be included in the data.
- Interpolation - No problems encountered. All data valid per OSM specification.

Programmatically cleaning the data

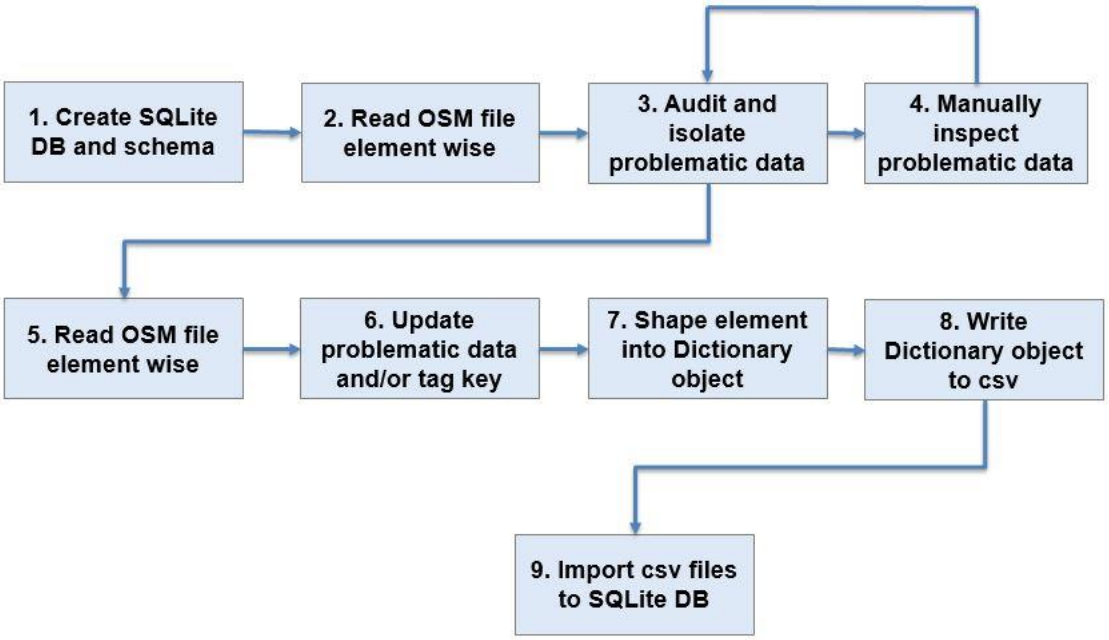
A separate Python update function was written to programmatically clean each of the audited tag types where problems were encountered. Here is the cleaning logic per tag type:

- Flat number – Tags were standardized to have flat numbers represented by “addr:flatnumber”. The “addr:flat” tags were converted to “addr:flatnumber”.
- House name – “addr:name” tags were converted to “addr:housename”. House names in the format <number> - <number> or simply numeric house names were re-tagged as “addr:housenumber”. House names in the format “Flat <number>” were re-tagged as “addr:flatnumber”. House names in the format <number> - <name> were split to separate the house number and house name. The house number portion was placed within a newly created “addr:housenumber” tag, while the house name portion was placed within the existing “addr:housename” tag. If however the <name> portion ended with “Street”, the “addr:housename” tag was ignored, as the “addr:street” tag for the node or way would always be present.
- City – City names are mostly valid. Misspellings and difference in case were addressed by using a mapping dictionary.
- Street type – As described in the above section, street names were found to be mostly valid. Street endings of “ROAD” and “Rd” were cleaned to replace them with “Road”.
- Postcode – Postcodes are recorded in 2 formats in the OSM file – full postcodes and partial postcodes. At first, the partial postcodes seem like a bad idea, but diving into the data, we can see that partial postcodes are used to designate postal code districts, and they are therefore correct in context. No cleaning was required here.
- House number – To clean house numbers, I trimmed out spaces and replaced semi-colon separators with the more widely used comma. Two specific values, “Council Offices” and “Padley” were re-tagged as “addr:housename”.
- Phone – The first step here was to strip out extraneous characters such as parentheses, spaces and periods. Multiple phone numbers within the same tag were separated and cleaned, before joining them again. Each phone number was standardized by prefixing it with the country code, and the trailing 0 was removed if it was present.

- Website - I decided to standardise these to include the “http://” prefix where it did not exist. Invalid websites earlier identified were simply recorded. NOTE – this slowed down the performance of the code, and I would not use this method on a larger dataset!

Importing the data into SQLite

Here is a quick look at the flow of activities involved:



Before importing the data into SQLite, I looked at the schema provided for the nodes, ways and related tags and position tables. I first created a new “openstreetmap” database within sqlite, and executed the schema creation scripts for the tables, given in the project specification, in this new database. The tables were now ready to receive the data! The next step was to move the data from the Python environment into persistent .csv files which would contain the same schema as created in the database. Once the csv files mirroring the database schema were available, with all of the cleaned data populated, then the csv mode and import command within the SQLite database could be used to pull in the cleaned OSM data.

In order to move the data from the Python environment into csv files, I carried out two passes on the OSM data: (1) programmatically read into memory each element of the OSM file, identified the tag and audited it. Manually examined the audit results, updated the audit code for each tag to improve the identification of problematic data (e.g. by updating regular expressions for the various fields), and repeated the audit process iteratively until all of the problematic data was correctly isolated. (2) In the second pass, I wrote Python code to programmatically read into memory each element of the OSM file, invoke a generic “updateTag” function that would, under the hood, identify the tag and invoke the specific update function for that tag. The returned updated tag would now contain the correct tag name and value. A generic “shapeElement” function would then convert the cleaned element into a dictionary object consisting of parent (i.e. node or way) element attributes and corresponding tag attributes. The dictionary would then be parsed by a UnicodeDictWriter object that would write that element into the csv file. Any re-tagging of elements in the form of adding or ignoring tags was also carried out as part of the update function.

Once the csv files were prepared, these were imported into the SQLite database after setting mode to csv. The tables were now ready for analysis.

Queries and Analysis of data

1. Total number of nodes:

```
SELECT COUNT(*) FROM Nodes;
```

246861

2. Total number of ways:

```
SELECT COUNT(*) FROM Ways;
```

38615

3. Top 10 contributing users:

```
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;
```

OJW'|62974
Ian Haylock'|23254
UniEagle'|18541
marsbar28'|17487
Mark_S'|15681
ecatmur'|14881
NormTeasdale'|13426
LivingWithDragons'|11564
jpennycook'|9455
Zorac'|7111

4. In what year were most entries created?

```
SELECT e.YearPart, SUM(e.CountByYear) AS CountByYear FROM
(
SELECT strftime('%Y',DATE(substr(timestamp, 3, 10))) AS YearPart, COUNT(*) AS CountByYear FROM Ways
GROUP BY YearPart
UNION ALL
SELECT strftime('%Y',DATE(substr(timestamp, 3, 10))) AS YearPart, COUNT(*) AS CountByYear FROM Nodes
GROUP BY YearPart
) e
GROUP BY e.YearPart;
```

2006|107
2007|285
2008|1808
2009|1617
2010|72798
2011|18393
2012|80032
2013|17937
2014|13916
2015|46059
2016|11844
2017|13536
2018|7144

5. Count of distinct keys:

```
SELECT COUNT(DISTINCT key) FROM
(SELECT key FROM nodes_tags
UNION
SELECT key FROM ways_tags
);
```

370

6. Count of interesting tags!

```
SELECT e.key, SUM(e.CountOfKey) AS CountOfKey FROM
(
SELECT key, COUNT(key) AS CountOfKey FROM nodes_tags
WHERE
```

```
key LIKE '%cycleway%' OR key LIKE '%religion%' OR key LIKE '%disused%' OR key LIKE ‘%food%' OR key LIKE
'%school%' OR key LIKE '%sport%' OR key LIKE '%recycling%'
GROUP BY key
UNION ALL
SELECT key, COUNT(key) AS CountOfKey FROM ways_tags
WHERE
key LIKE '%cycleway%' OR key LIKE '%religion%' OR key LIKE '%disused%' OR key LIKE ‘%food%' OR key LIKE
'%school%' OR key LIKE '%sport%' OR key LIKE '%recycling%'
GROUP BY key
) e
GROUP BY e.key;
```

```
cycleway'|670
disused'|20
food'|5
public_transport'|10
recycling_type'|5
religion'|59
school'|1
sport'|262
```

7. Distinct religion:

```
SELECT DISTINCT(e.value), SUM(e.CountOfValue) FROM
(
SELECT value, COUNT(value) AS CountOfValue FROM nodes_tags
WHERE key like '%religion%'
GROUP BY value
UNION
SELECT value, COUNT(value) AS CountOfValue FROM ways_tags
WHERE key like '%religion%'
GROUP BY value
) e
GROUP BY e.value;
```

```
christian'|58
muslim'|1
```

8. Popular sport:

```
SELECT DISTINCT(e.value), SUM(e.CountOfValue) FROM
(
SELECT value, COUNT(value) AS CountOfValue FROM nodes_tags
WHERE key like '%sport%'
GROUP BY value
UNION
SELECT value, COUNT(value) AS CountOfValue FROM ways_tags
WHERE key like '%sport%'
GROUP BY value
) e
GROUP BY e.value
ORDER BY 2 DESC
LIMIT 5;
```

```
soccer'|80
tennis'|75
equestrian'|14
bowls'|13
cricket'|13
```

9. Most descriptive elements in terms of number of tags:

   Node:

```
SELECT id, COUNT(key) FROM nodes_tags
GROUP BY id
ORDER BY 2 DESC
LIMIT 1;
```

```
565754349|18
```

Way:

```
SELECT id, COUNT(key) FROM ways_tags
GROUP BY id
ORDER BY 2 DESC
LIMIT 1;
```

```
418768435|25
```

10. Percentage contribution of top 10 users:

```
SELECT SUM(f.num) * 100/(SELECT SUM(g.counts) FROM (SELECT COUNT(*) as counts FROM nodes UNION
SELECT COUNT(*) as counts FROM ways) g)
FROM
(
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
) f;
```

```
68
```

11. Total number of unique users:

```
SELECT COUNT(DISTINCT uid) FROM
(SELECT DISTINCT uid FROM nodes UNION SELECT DISTINCT uid FROM ways);
```

```
501
```

Ideas for additional improvements

Having worked with this data set both using both manual and programmatic inspection, I noticed that the data set for the chosen geographical area is far from complete! Even for the areas I am thoroughly familiar with, a large number of residential homes, stores and streets are unlisted. Further, for the nodes and ways listed, there is a wide variation in definition for the same kind of tag. This, in my view, makes the data difficult to use for any reliable analysis. Another issue is that the dataset is unwieldy due to the large number of tags or keys used. There are altogether 370 different tags used here! I strongly suggest that the openstreetmap project should operate to a tighter schema. Each tag type must have a list of allowed attributes with some flexibility provided, of course, for legacy data and comments. The list of allowed tags should be reviewed and pruned. New incoming data should be validated to the revised schema. There should also be designated sources that are authorised to supply definition data for public areas. For instance, Ordnance Survey's OS Street View could be the definitive source for street data. There should be a defined process for removing "disused" entries. Redundant data that cannot be retrieved must be removed, e.g. references to image data.

In terms of analysis, given more time, I would inspect a larger subset of tags and especially, research ways of comparing the completeness of street information with the OS Street View database.

Benefits and problems with additional improvements

The obvious benefit of the above-described improvements in the data definition would be a significant improvement in data quality. A tighter schema for the dataset as a whole, coupled with tag-specific attribute rules would enable end users (primarily developers) to conduct reliable analysis using openstreetmap data. A reduction in the number of tags would result in greater standardization, and removing redundant data will not have some impact on dataset size, but more importantly, remove "loose ends" in the data.

The above benefits are not without problems though! Openstreetmap is a live system and at the same time, a work in progress. Any schema updates require meticulous thinking and planning, not only for the present, but also taking into account the future of urban areas. This requires close collaboration with urban planners. For instance, how will the project be impacted by urban sensor infrastructure? Another issue with changing definitions is the impact on live systems that currently rely on openstreetmap. Data migration, upgrades to dependent systems, upgrade of data sources that feed into the openstreetmap project will all have to be looked at. Any downtime and change management processes will need to be shared with data providers, integrators and consumers.