OpenStreetMap Data Wrangling with MongoDB

This is project 3 of Udacity's Data Analyst Nanodegree: Wrangle OpenStreetMap Data.

The aim is to learn to use data munging techniques, such as assessing the quality of the data for validity, accuracy, completeness, consistency and uniformity, to clean the OpenStreetMap data for a select part of the world.

This GihHub repository contains the project report `data_wrangling_report.pdf` and the code used as `.py` files.

Openstreetmap Data for Central London

I started with london.osm, a 67.7 MB XML file containing Openstreetmap data for a custom area in central London, obtained with Overpass API. I chose to wrangle street name, postal code and phone number data using Python and MongoBD.

Problems Encountered in the Map Data

In "Audit of map data" I summarize the auditing approach and the problems detected. In "Data tranformation" I present the methods used to clean and tidy the data in order to resolve a selection of the problems encoutered.

At this stage I worked only on the XML data downloaded from Openstreetmap. At the end of data transformation process, I produced a JSON file ready to be uploaded to MongoDB. All work with data in MongoDB is presented in "Data Overview".

Audit of map data

The objective of auditing was to spot inconsistencies and mistakes of value or format in the data. I audited street names, postal codes and phone numbers in london.osm by analysing the XML data with Python scripts.

To audit street names, I used `audit_street.py`, a custom Python script based off of Udacity's code. The script prints all values associated with the "street" tag that don't match those in a list of expected values.

To audit postal code and phone number data, I used code like `audit_phone.py` to print out all values. Post codes and phone numbers are short strings, and some problems can be spotted simply by eyeballing the data.

1. Problems with street names

To find mistakes more easily, I filtered out correct street names using the list of expected values below. I ran `audit_street.py` several times, each time finding new names to add to the list.

```
expected = ["Acre", "Approach", "Arch", "Avenue", "Bridge", "Circle", "Circus", "Close", "Corner", "Court",
            "Crescent", "Drive", "East", "Embankment", "Estate", "Garden", "Gardens", "Gate", "Grove", "Hill",
            "Lane", "Market", "Mews", "North", "Place", "Road", "Row", "South", "Square", "Station", "Street",
            "Terrace", "Walk", "Way", "West", "Wharf", "Yard"]
```

Among the values that didn't match the list, I detected different types of mistakes and inconsistencies:

1. Abbreviation of street names
2. Small caps
3. Typos
4. Mistakes

1. Abbreviation of street names

An inconsistency problem in data format. A small number of values included abbreviations for `Road`, `Street` and `Square` instead of their full form. These values were: `Rd`, `St`, `St.` and `Sq`.

2. Small caps

Another inconsistency problem in data format. I found instances of `lane`, `place`, `street` and `market` instead of their standard capitalized versions.

3. Typos

Certain street names had a spelling mistake in how they were written. A case I found was `Steet` where the user probably meant to input `Street`, and `Picadilly` for `Piccadily`.

4. Mistakes

Several values were found that were not street names. For example `Chelsea`, `Lambeth` and `Mayfair`, which are names of neighborhoods, or `5A` which appears to be a house number.

2. Problems with post codes

Printing out post codes highlighted a frequent problem related to partial post code information. A complete post code in London has two parts and looks like this: `NW1 2BU`. Many values only included the first half of the postcode, e.g. `NW1`.

The first half of a post code corresponds to a larger area of the city than the full post code. Therefore, availability of halves of post codes leads to a problem of lack of complete detail more than wrong location.

3. Problems with phone numbers

Simply by printing out phone number values, I detected a number types of mistakes and inconsistencies:

1. Missing phone numbers
2. Inconsistent formatting
3. Not a phone number

1. Missing phone numbers

The number of phone numbers entries in the data set is relatively small. My data analysis in MongoDB - displayed below - found 1.373 phone number entries compared to 12.797 entries with street information.

2. Inconsistent formatting

Looking at the phone numbers that were printed out shows a number of inconsistent use of conventions. For example:

```
+44 20 74051992
+44-20-79352361
+44 20 7620 328
+442073704988
0207 850 0500
+44 (0)20 74025077
442074018080
```

The formatting problems in this small sample include:

- Mixed use of national and international formats with country code `+44`
- Use of `0` before the London city prefix (20) when the country code is missing
- Uneven use of brackets `( )` and dashes `-`
- Uneven spacing between digits

3. Not a phone number

I found one instance of a string like `+44 +44 20 73001000.` where the country code was entered twice and which ended with a full stop. My guess is that the user meant to input `+44 20 73001000` instead.

Data transformation

I used `transform.py` to clean and tidy the data and address a selection of the issues discussed above, namely:

- Abbreviation of street names
- Small caps in street names
- Typos in street names
- Formatting of phone numbers

The method used to solve these problems is substitution of problematic values in the XML file with the correct values.

To solve all the problems related to street names, I used a list of tuples containing values to be replaced and correct values:

```
STREET_CORRECT = [
    ("lane", "Lane"),
    ("market", "Market"),
    ("Picadilly", "Piccadilly"),
    ("place", "Place"),
    ("Rd", "Road"),
    ("St ", "Street"),
    ("St.", "Street"),
    ("Steet", "Street"),
    ("street", "Street"),
    ("Street.", "Street"),
    ("Sq", "Square"),
    ("turnstile", "Turnstile")
]
```

To reformat phone numbers to the national UK format, I used the `update_phone_number()` function in `transform.py`, also using Python library phonenumbers.

Taking `london.osm` as input, `transform.py` corrects these problems during the conversion of XML values into JSON. The resulting "clean" output file `london.osm.json`, of size 66.9 MB, is ready to be uploaded to MongoDB.

Data Overview

1. Load data into MongoDB

In order to load the JSON file into MongoDB, I had to first install MongoDB and create a new database and collection for the map data.

1. Install MongoDB

As a first thing, I installed MongoDB on my Mac using Homebrew and the official MongoDB installation tutorial.

2. Create MongoDB database

Before importing the clean data file, I had created a local MongoDB database using the following command:

```
mongod --dbpath /users/robertozanchi/Desktop/Udacity/DAND/P3/data/db
```

3. Import data file into MongoDB

I imported the JSON file into a `london` collection within a newly created `maps` database, using the command:

```
mongoimport --file /users/robertozanchi/Desktop/Udacity/DAND/P3/london.osm.json --db maps --
collection london
```

```
2016-04-03T21:25:03.266+0200    connected to: localhost
2016-04-03T21:25:06.253+0200    [#######.................] maps.london    19.6 MB/63.8 MB (30.7%)
2016-04-03T21:25:09.255+0200    [##############..........] maps.london    39.3 MB/63.8 MB (61.6%)
2016-04-03T21:25:12.251+0200    [#######################.] maps.london    62.1 MB/63.8 MB (97.4%)
2016-04-03T21:25:12.647+0200    [########################] maps.london    63.8 MB/63.8 MB (100.0%)
2016-04-03T21:25:12.648+0200    imported 287928 documents
```

A quick database query confirms the existence of 287,928 items in the collection.

```
> db.london.count()
287928
```

2. Data analysis

Within MongoDB I performed analysis of data using the following commands.

Interesting statistics

Number of nodes

```
> db.london.find({type: "node"}).count()
242133
```

Number of ways

```
> db.london.find({type: "way"}).count()
45795
```

Number of street entries

```
> db.london.aggregate([{'$match': {'address.street': {'$exists': 1}}}, {'$group': {'_id': 'Street
entries','count': {'$sum': 1},}}])
{ "_id" : "Street entries", "count" : 12797 }
```

Most common street entry

```
> db.london.aggregate([{'$match': {'address.street': {'$exists': 1}}}, {'$group': {'_id':
'$address.street','count': {'$sum': 1}}}, {'$sort': {'count': -1}}, {'$limit': 1}])
{ "_id" : "Edgware Road", "count" : 185 }
`
```

Number of phone numbers

```
> db.london.aggregate([{'$match': {'phone': {'$exists': 1}}}, {'$group': {'_id': 'Phone number
entries','count': {'$sum': 1},}}])
{ "_id" : "Phone number entries", "count" : 1373 }
```

Number of unique users

```
> db.london.distinct("created.user").length
1174
```

Top contributing user

```
> db.london.aggregate([{$group: {"_id": "$created.user", "count": {$sum: 1}}}, {$sort: {"count":
-1}}, {$limit: 1}])
{ "_id" : "Paul The Archivist", "count" : 40223 }
```

Most common amenities

```
> db.london.aggregate([{$match: {"amenity": {$exists: true}}},{$group: {"_id": "$amenity",
"count": {$sum: 1}}},{$sort: {"count": -1}}])
{ "_id" : "restaurant", "count" : 1512 }
{ "_id" : "cafe", "count" : 923 }
{ "_id" : "bicycle_parking", "count" : 768 }
{ "_id" : "telephone", "count" : 724 }
{ "_id" : "pub", "count" : 609 }
{ "_id" : "bench", "count" : 589 }
{ "_id" : "post_box", "count" : 520 }
{ "_id" : "parking", "count" : 394 }
{ "_id" : "fast_food", "count" : 345 }
{ "_id" : "motorcycle_parking", "count" : 320 }
{ "_id" : "bicycle_rental", "count" : 286 }
{ "_id" : "waste_basket", "count" : 236 }
{ "_id" : "bank", "count" : 225 }
{ "_id" : "place_of_worship", "count" : 190 }
{ "_id" : "bar", "count" : 178 }
{ "_id" : "atm", "count" : 161 }
{ "_id" : "school", "count" : 150 }
{ "_id" : "embassy", "count" : 114 }
{ "_id" : "pharmacy", "count" : 111 }
{ "_id" : "recycling", "count" : 109 }
```

Additional Ideas: complete post code values

An additional idea for improving the selected Openstreetmap dataset is related to resolving the issue of incomplete post code information, which I was not able to address in this project.

Potential solution to complete post codes

A possible way to insert complete post code values in the database would be to search for the complete post code using available street name and number data. This would require access to a database that has this information.

Benefits of potential solution

I believe that completing post code data would bring at least three kinds of benefits:

1. Higher quality data: having complete and tidy post code entries in the dataset would mean having a higher quality dataset overall.
2. Potential for richer analysis: a tangible benefit is that it would become possible to analize the data using post

code constistently as the independent variable.

3. Unlock data uses that require post code information: once reliable post code data is obtained, new real-world applications become possible. An example is using the data for commercial activities, such as shipping and delivery of orders.

Challenges

Implementing the suggested method for completing post code data would also come with its own challenges, including:

1. Find a reliable third-party data source: completing post code information would require access to a database that has a combination of correct address and correct post code information. This requires trusting a third-party source to vaidate user input.
2. Cost of data access: access and usage of an external database woud come with additonal costs. If such database were provided on commercial terms, payment of a fee may be required. Other costs are related to the use of computational power, data storage and in time required to implement the solution.
3. Technical implementaton: if I were to try and implement the proposed solution, I would likely need to expand my skillset. Using an external database, or an API, would require writing new code, and new using applications, libraries or frameworks.

Files

`audit_street.py` audits street name entries by returning all values that don't match a list of expected names;

`audit_phone.py` simply prints all phone numbers;

`data_wrangling_report.pdf` is the final project report;

`transform.py` includes code for cleaning the data and converting XML to JSON.

Resources:

To install `phonenumbers` python library, use Mac Terminal command `sudo pip install git+git://github.com/daviddrysdale/python-phonenumbers.git`