Wrangling OpenStreetMap Data with MongoDB
Auditing, Cleaning, and Querying an OSM Dataset

Ronald Rihoo

Map Area: Dallas, TX, United States

1. The Challenges Encountered with the Dallas County OSM Dataset.

1a. User- and/or Bot-Induced Errors.

Problem: extra address values for the “street” name.

Ex1: <tag k="addr:street" v="5223 alpha road dallas tx 75240"/>

Ex2: <tag k="addr:street" v="Avenue K, suite 700-285"/>

Where it should have been (momentarily disregarding capitalization):

Ex1: <tag k="addr:street" v=”alpha road”/>

Ex2: <tag k="addr:street" v="Avenue K"/>

Fix: use regex to find the 'street name' part of the string, then use any and all means to salvage the rest of it.

Problem: Incorrect placement of street name suffix (street type).

Ex1: <tag k="addr:street" v="Ave K."/>

Ex2: <tag k="addr:street" v="Avenue K, suite 700-285"/>

Suffix in the prefix position. This is clearly a mistake made by the entity that made this entry. This is meant to be "K Avenue" or "K Ave". I've lived in the Dallas County for decades and, to my knowledge, there is no "Ave K." anywhere around here; however, there is a “K Ave”.

Fix: split at comma; scan for known street types in each fragment; replace suffix-word at the end of fragment

An error like this one "Avenue K, suite 700-285" could be fixed by looking for commas and splitting the string at those points, then auditing all parts again, separately. The part that makes sense will then replace the whole thing. Hopefully, the other parts can be audited for other things, such as cities, states, and countries, so that good information can still be retained.

Problem: no space in between different parts of the name value

```
<tag k="addr:street" v="Hwy78"/>
```

Where it should have been: “Hwy 78”

Fix: search with regex; split between alphabets and numerals; rejoin with a space.

I decided to ignore highway values for this project, due to the size of the project as is with just focusing on street names. However, this is a fairly simple fix, as one could split the alphabetical (or numeric) part of the string and rejoin the fragments with a space in between.

Where:

```
string = “Hwy78”
```

Detect:

```
re.compile(r'^[a-z]{1,4}[0-9]{1,3}$', re.IGNORECASE).search(string).group()
```

Reformat:

```
new_value = ' '.join(re.split('(\d+)', string))          # result: “Hwy 78”
```

The main problem that renders this fix useless is when incorrect and extra values coexist.

Problem: Uncapitalized first letters of names, types, and designations.

```
<tag k="addr:street" v="Parkside dr"/>
```

Where it should have been: “Parkside Dr”

Fix: I added 'dr' as a key in the “unexpected_suffix” dictionary, as needed.

I was able to do this freely, because of the convenience that was provided by the error logging feature in my code.

A better fix method is to ignore the case when searching for street types. Upon finding the street type, determine if first letter is uppercase. If not, then define a replacement string that will use the built-in string upper() function to take the uppercased first letter, then concatenate the rest of the letters in the original string.

In clean.py, I have written a function to conduct this procedure, which is used by deep_scan():

```
def up(word):
   cased_word = word[0].upper() + word[1:]
   return cased_word
```

1b. Natural Street Map Data Challenges.

Challenge: there are numerous street types.

Fix: scan; compare; flag mismatches; manually inspect errors and update type list.

I had my code scan all street values by separating the lateral words, prefix and suffix (or first and last word, of the string, then flagging any unexpected prefix and suffix (comparison failures against expected values provided in lists).

I took baby-steps in this process, as one of the earliest error messages my code gave was:

> Error: street value "Saint Andrew's Court" seems to not be a valid street name.

My unexpected suffix (street type) list did not include “Court” (among others), as it does now:

> { ... 'Court' : 'Ct', ... }

Challenge: some street names include the same words as street types.

Nearly all words are street types:

Canyon Creek Drive
Trail Lake Drive
Terrace Drive

Fix: if the last word is a street type, then just accept it as so.

This problem inspired me to have the script learn the medial words (all words in between the first and last) in order to have an extra set of words to compare to; for confidence in the medial words actually being street name parts. If I had more time, then I would implement statistical models.

Problem: unexpected characters such as apostrophes in street names.

> <tag k="addr:street" v="Saint Andrew's Court"/>

Fix: remove apostrophe character from the regular expressions for error scanning.

Originally, the first layer of scanning included a regular expression search on the street name value to identify problematic characters, which included the apostrophe character.

1c. Programmer-Induced Errors.

Error: some street names include cardinal direction words (North, West, E...).

Where original is:

“Westmoreland Dr”	# 'West' is an unconventional prefix in Dallas

It would become:

“Wmoreland Dr”	# 'W' is the expected prefix

In fact:

“Courtside Lane”	# 'Court' is an unconventional suffix in Dallas

Would become:

“Ctside Ln”	# 'Ct' is the expected suffix

This caused a glitch in the combined cleaning process, which induced unnecessary changes.

Fix: separate the scan and clean phases of the prefix and suffix (and any other part).

This was caused when an independent problem was found with the suffix, which invoked the operation of the script to jump to a combined (prefix/suffix) cleaning process.

In the cleaning state, when handling street names that did not have a prefix (cardinal direction), the function would assume that there was a problematic prefix (if, and only if, it included direction or street type words), so the prefix cleaning would occur on the first word of the actual street name. I separated the expected prefix list and unexpected prefix dictionary from those of the suffix, and also separated the auditing (searching and cleaning) states from each other.

2. Overview of the Data

2a. Size of the File.

```
  81351905 bytes (81.34 MB)   dallas_map.osm
 112074318 bytes (112.1 MB)   dallas_map.osm.json
```

2b. Number of Unique Users.

```
> db.dallas.distinct('created.user').length
485
```

2c. Number of Nodes and Ways.

```
> db.dallas.find( { 'type' : 'node' } ).count()
348485

> db.dallas.find( { 'type' : 'way' } ).count()
38537
```

2d. Number of Unique Amenities.

```
> db.dallas.distinct( 'amenity' ).length
52
```

2e. Amount of Restaurants.

```
> db.dallas.find( { 'amenity' : 'restaurant' } ).count()
271
```

2f. Number of Unique Restaurants.

```
> db.dallas.aggregate([ { '$match' : { 'amenity' :
'restaurant' }}, { '$group' : { '_id' : '$name', 'unique' :
{ '$addToSet' : '$name' }}}, { '$unwind' : '$unique' },
{ '$group' : { '_id' : '$name',   'count' : { '$sum' : 1 }}}, {
'$project' : { '_id' : 0, 'count' : 1 }} ])
{ “count” : 233 }
```

3. Opportunities and Challenges of Using OSM Data.

3a. Opportunity of an Open Toolkit for Smarter Investment Strategies.

Investors could use OSM data to analyze the popularity of amenities, or other types of businesses, in a given region. This can lead to faster and better investment analysis, because an endless amount of statistical tests and algorithms can be applied to the data for gaining insight about any geographic location.

However, at this state, even the OSM dataset that I used in this project has very little information in it. It barely includes any of the restaurants or businesses. OSM has so little data for this popular region, that I was driven to contribute to the area by adding some of the restaurants and businesses. So this is a problem in OSM's current state.

Another issue is that automated analysis becomes more difficult with the fact that some business names are stored differently from other entries. As an example, from this project's OSM dataset:

```
> db.dallas.find( { 'amenity' : 'cafe' } )[25]['name']
Starbucks Coffee
> db.dallas.find( { 'amenity' : 'cafe' } )[29]['name']
Starbucks
```

Perhaps every item, in every entry needs to be cleaned out. This will require a very large, and continuously updated, dataset of universally accepted business information (ex: names, URL format, etc.), which we may compare to when cleaning out different parts of entries for business locations in OSM datasets.

To emphasize the need for cleaning out every bit of OSM data, I will present a part of the Dallas County OSM dataset that I used in this project, which speaks for itself:

```
"address": {
  "street": "Alpha Rd",
  "housenumber": "972-788-2591"
},
```

-the “housenumber” is not meant to be a house phone number, but instead a part of the address.

3b. Overcoming Challenges in the Use of OSM Data for Business Analysis.

In order to build a dataset of universally accepted business information, the problem may be best solved by establishing an online open-source business registry that has some similar functionality to OSM. I propose the name: OpenBusinessRegistry. Individuals should be able to create OBR (OpenBusinessRegistry) accounts to use for adding and/or changing business information in the database to match what they consider as universally acceptable.

Where one user makes a mistake, others can fix that mistake, as only one business name entry (along with other information that falls under the name) should exist for each business. And to aid the users in their effort in ensuring that the business name is entered only once, the application should show them automatic search results of existing entries similar to what they are trying to add to the database.

The issue with this is that some users and bots will still make entries with mistakes, regardless of any alerts and alarms popping up about similar entries. To help reduce such errors, volunteers, passionate administrators, and symbiotic analysts should have tools within the web app that shows them arrays of similar and questionable business name entries, so that they can be verified, merged with others (under one business name), or completely removed. However, there is one problem that tools can't fix.

Building a community around a cause or benefit requires strategic public campaigns. The root problem in this idea is the same as OpenStreetMap's. Will enough people participate? If not enough people are aware or have the incentive to participate, then Jane's Taco Deli and other local or small businesses might never make it to the database. In such case, the same problem will reoccur. There won't be enough data for geographic business analysis (or any business analysis); although, there is one more major strategy to consider for the general challenge.

At this stage, tools and strategies can be mixed to enhance the web app. Business data can be continuously scraped from the Internet, especially from Wikipedia and other popular sources that are already built and maintained by communities. Business names can also be scraped from business registries and even Fortunate 500 listings to ensure coverage of a finite quantity.

Automated processes can prepare large sets of business data to be reviewed by human minds. Volunteers, passionate administrators, and symbiotic analysts can clean this data up using the web app's GUI, which will be designed and tailored for UX to provide ease of operation.

Conclusion

It is easy for us to learn conventions and standards, but having to learn the ways that others can misunderstand conventions, and/or create other types of error in data entry, is an endless endeavor. The need for manual inspection, with the human mind and the use of tools such as Python and MongoDB, may always be a necessity during data clean-up (and optimization), due to the fact that the amount of errors that anyone could make, at its simplest, is as vast as: the amount of different types of characters that can be inserted in OSM datasets for each character index in strings (for keys and values) to the power of the amount of characters that could possibly be input in OSM datasets as keys and values. So concerning only ASCII characters (decimal) 32 through 126, then with just 10 characters in a string, we're left with 53 quintillion (94^10) possible errors that could be made just by typos or choice of formatting; however, let us make no mistake here, as the possibility of error should be deemed endless until someone mathematically proves that it is not.