Humboldt OSM Project

November 14, 2016

Welcome to my project - an exploration and conversion of OpenStreetMap XML data to CSVs to be further analyzed in SQLite3.

This project uses Python with a few critical data analysis modules, primarily ElementTree. A SQLite database is specified from CSVs and queries run to answer a variety of questions.

1 Project Overview

In brief, our goals are the following: 1. Audit OSM XML to better understand our data 2. Improve the data programmatically 3. Convert from OSM XML to CSVs 4. Construct schemata for SQLite3 and import CSVs 5. Run queries

2 Data Munge

2.1 Sampling a Large Dataset

The unzipped OSM XML file for Humboldt County is over five hundred megabytes in size. It is unwieldy and unable to be kept in memory buffer for easy manipulation.

In samplerfrag.py, we expose a flexible function for sampling the total dataset. By parsing elements with ElementTree's iterparse method, then looping through and writing elements to file when a division remainder is equal to zero, we may tune the divisor to sample a precise fraction of our total dataset.

```
In [9]: def get_element(osm_file, tags=('node', 'way', 'relation')):
            """Yield element if it is the right type of tag"""
            context = iter(ET.iterparse(osm_file, events=('start', 'end')))
            _, root = next(context)
            for event, elem in context:
                if event == 'end' and elem.tag in tags:
                    yield elem
                    root.clear()

        with open(SAMPLE_FILE, 'wb') as output:
            output.write('<?xml version="1.0" encoding="UTF-8"?>\n')
            output.write('<osm>\n  ')

            # Write every kth top level element
```

```
            for i, element in enumerate(get_element(OSM_FILE)):
                if i % k == 0:
                    output.write(ET.tostring(element, encoding='utf-8'))

            output.write('</osm>')
```

With k=100, our sample.osm file is 5,555KB. This is much more manageable than our full OSM file.

2.2 Auditing Tags

Most data in OSM XML is contained in tags within elements. When we run our audit fragment, tagexplore.py, we find the following:

```
({'lower': 3663, 'lower_colon': 7127, 'other': 15,
'problemchars': 0},

{None: 68473, 'is_in:iso_3166_2': 1, 'name_1': 2,
'source:hgv:national_network': 4, 'tiger:name_base_1': 5,
'tiger:zip_left_1': 3})
```

By comparing tags against a few regular expressions, the audit filters tags into dictionary keys, seperating tags into categories based on their format. A further level of exploration is conducted into the "others" that fail to trigger the regular expressions filter. We find no problem characters, but quite a collection of "others".

Our problem children in this data seem to belong to four groups: 1. Tags with underscores 2. Tags with hyphens 3. Tags with more than one colon 4. The big one - Nones. Nearly seven million of them.

2.3 Street Name Consistency Audit and Programmatic Improvement

Street names are contained within many of our tags. However, because the contributor base of OpenStreetMaps is varied, street type naming conventions are often inconsistent. For orderly data, we must identify misnamed street types and update them. These tasks are handled by audit_xml.py and tagmaul.py respectively.

In brief, these programs use a regular expression to identify potential street names, then compare that street's type against a list of expected values ("Street", "Road", etc.).

```
In [4]: def audit_street_type(street_types, street_name): #used in audit()
            m = street_type_re.search(street_name) #regex
            if m:
                street_type = m.group()
                if street_type not in expected:
                    street_types[street_type].add(street_name)
```

Street types that are not in the list of expected values ("St", "Rd", etc) are aggregated into a dictionary for handling by mapping them to their correct equivalent in tagmaul.py { "St": "Street" }. Most unexpected types were able to be handled and updated, leaving our current output dictionary of oddball street types looking sparse.

```
{'13N11': set(['Forest Route 13N11']),
'1N18': set(['Forest Rt 1N18']),
'2S05': set(['Forest Route 2S05']),
'5N27': set(['Forest Route 5N27']),
'6N12D': set(['Forest Route 6N12D']),
'A': set(['Road A'])}
```

With some more handling with regular expressions, even these unconventional street types may be handled.

The update function inspects every identified street type name and compares them against the dictionary mapping, returning the new type name if the input name matches a key in the dictionary.

```
In [7]: def update_name(name, mapping):      # used in updater()
            newname = None
            namesplit = name.split()
            for key, value in mapping.iteritems():
                if namesplit[-1] == key:
                        newname = name.replace(key,value)
            if newname == None:
                return name
            return newname
```

The updater also prints to console when a street type name change is made, like so:

```
Before: 7th St
After: 7th Street
```

This lets us observe that our updater is working as intended.

This update program is where most future improvements may be made. With some more regex work, we may handle dirtier and more complex names, like those that have directional contractions (Se Cnr => Southeast Corner) and names like 'Bank Rd North' where the last member of the list is not the actual street type.

2.4 Tiger GIS Handling

Many street names in Humboldt Country's OSM are contained in Tiger GIS tags, which require special handling. The primary difficulty lies in distinguishing tags where k="name" and are Tiger GIS street names from tags where k="name" and might be a hairdresser or cemetary. Adding to the difficulty, the tag attribute ['k']="tiger:name_base" always comes after ['k']="name".

I chose to handle it by implementing a sort of logic switch that will audit values where k="name" if it is followed later by a k="tiger:name_base".

```
In [ ]: maybename = ""
        istiger = False
        for tag in elem.iter("tag"):
            if tag.attrib['k'] == "name":
                maybename = tag.attrib['v']
            if tag.attrib['k'] == "tiger:name_base":
```

```
            istiger = True
        if is_street_name(tag):
            audit_street_type(street_types, tag.attrib['v'])
        if istiger:
            audit_street_type(street_types,maybename)
```

3 Querying the Database

After exporting our cleaned OSM XML to CSVs by element and tag and specifying SQL schemata for the database, we may run queries! What follows is a selection of queries and their results.

```
In [ ]: # Number of NODES/WAYS/NODES_TAGS/WAYS_TAGS/WAYS_NODES
        sqlite> SELECT COUNT(*) FROM nodes;
        2809878
        sqlite> SELECT COUNT(*) FROM ways;
        133188
        sqlite> SELECT COUNT(*) FROM nodes_tags;
        15598
        sqlite> SELECT COUNT(*) FROM ways_tags;
        1059831
        sqlite> SELECT COUNT(*) FROM ways_nodes;
        2965493

        # Number of unique user contributors
        sqlite> SELECT COUNT(DISTINCT(users.uid))
           ...> FROM (SELECT uid FROM nodes
           ...> UNION ALL SELECT uid FROM ways) users;
        332

        # How many default values were created due to missing values?
        sqlite> SELECT COUNT(*) FROM nodes WHERE
           ...> id = 0000 OR lat = 0.00 OR lon = 0.00 OR
           ...> user = "Missing Value" OR uid = 0000 OR
           ...> version = "Missing Value" OR changeset = 0000 OR
           ...> timestamp = "Missing Value";
        1
        sqlite> SELECT COUNT(*) FROM ways WHERE
           ...> id = 0000 OR lat = 0.00 OR lon = 0.00 OR
           ...> user = "Missing Value" OR uid = 0000 OR
           ...> version = "Missing Value" OR changeset = 0000 OR
           ...> timestamp = "Missing Value";
        0
        sqlite> SELECT * FROM nodes WHERE
           ...> id = 0000 OR lat = 0.00 OR lon = 0.00 OR
           ...> user = "Missing Value" OR uid = 0000 OR
           ...> version = "Missing Value" OR changeset = 0000 OR
           ...> timestamp = "Missing Value";
```

```
32820705,40.6303271,-123.4687854,"Missing Value",0,1,167590,
"2007-07-25T09:56:21"

# Do the same top users contribute to both ways and nodes?
sqlite> SELECT user, COUNT(user)
   ...> FROM nodes
   ...> GROUP BY user
   ...> ORDER BY COUNT(user) DESC LIMIT 5;
nmixter|1791267
woodpeck_fixbot|854394
Apo42|38957
Chris Lawrence|12500
Dilys|11620
sqlite> SELECT user, COUNT(user)
   ...> FROM ways
   ...> GROUP BY user
   ...> ORDER BY COUNT(user) DESC LIMIT 5;
pnorman_mechanical|50404
jumbanho|37794
Bilbo|14410
DaveHansenTiger|9436
nmixter|4706

# What types of tag keys are most common?
sqlite> SELECT key, COUNT(key)
   ...> FROM nodes_tags
   ...> GROUP BY key
   ...> ORDER BY COUNT(key) DESC LIMIT 5;
name|1880
ele|1296
feature_id|1071
created|1025
county_id|773
sqlite> SELECT key, COUNT(key)
   ...> FROM ways_tags
   ...> GROUP BY key
   ...> ORDER BY COUNT(key) DESC LIMIT 5;
source|123917
fcode|107833
fdate|107824
com_id|107804
ftype|107745

# What amenities are available in Humboldt?
sqlite> SELECT value, COUNT(value)
   ...> FROM nodes_tags
   ...> WHERE key = "amenity"
   ...> GROUP BY value
```

```
   ...> ORDER BY COUNT(value) DESC LIMIT 5;
school|136
place_of_worship|89
restaurant|67
bench|51
fast_food|41

# What are some libraries in Humboldt?
sqlite> SELECT *
   ...> FROM nodes_tags
   ...> WHERE value = "library" LIMIT 3;
368168979|amenity|library|"regular"
368168982|amenity|library|"regular"
368169062|amenity|library|"regular"

# What are the coordinates each library's node?
sqlite> SELECT nodes_tags.id,nodes.lat,nodes.lon
   ...> FROM nodes JOIN nodes_tags
   ...> ON nodes_tags.id = nodes.id
   ...> WHERE nodes_tags.value = "library" LIMIT 3;
368168979|41.4559732|-122.8947551
368168982|40.5795738|-124.2611664
368169062|40.5990202|-124.1522755

# Which nodes are referenced by node_id in ways_nodes the most?
sqlite> SELECT nodes.id AS Node,
   ...> COUNT(ways_nodes.node_id) AS NodeUse
   ...> FROM nodes JOIN ways_nodes
   ...> ON nodes.id = ways_nodes.node_id
   ...> GROUP BY Node
   ...> ORDER BY NodeUse DESC LIMIT 3;
86122425|7
1025120607|7
1056491759|7
```

4 Future Improvements to the Dataset

My first recommendation is a fairly obvious one – Tiger GIS keys and values need to be integrated into the OSM in a way that doesn't cause name overlap with other types of keys and values. It may be a special challenge for an open map system, as everybody contributing to the map may have different ideas about structure, but the Tiger tags in particular took quite a bit of time and effort to untangle.

Tiger is not the only offender, it seems. There are very many tag.attrib['k'] = 'name' values that are not useful, including a few different coordinate notations.

Open projects like these make me think of Wikipedia. Naturally, Wikipedia's user base is very large and fiercely competitive when it comes to the rigor of the information contained, a standard perhaps unlikely for OSM. However, the process of providing existing structure and alteration

auditing by a group of trusted volunteers tends to give Wikipedia the advantage of flexibility and integrity.

In short, if a similar system for auditing data contributions to OSM could be managed, there may be less potential for overlap of tag keys and values from different user notation systems.

In [ ]: