OpenStreetMap Data Wrangling with SQL

OpenStreetMap is an open source alternative to Google Maps.

http://www.openstreetmap.org

Users can map things such as nodes, streets, spots of interest, polylines of buildings, etc. The data is stored in XML, for ease of analysis although it is a propriety version of XML called OSM XML. http://wiki.openstreetmap.org/wiki/OSM_XML

Some highlights of the OSM XML format relevent to this project are:

- OSM XML is list of instances of data primatives (nodes, ways, and relations) within a given location.
- Nodes are abstract representations of physical locations.
- Ways are representative of pathways, for human, and non-human pathing.
- Nodes and ways both have children tag elements of key value pairs of descriptive information about the parent node or way.
- Due to it being user generated and open source, there is likely going to be dirty data. This project's goals are auditing, cleaning, and data summarization using Python 2.7 and SQLite.

Location: St.Charles, MO, USA

Data Sources:

https://overpass-api.de/api/map?bbox=-90.7189,38.7013,-90.3117,38.8884

The unzipped data was 176 MB

Files included:

- OpenStreetMap Data Wrangling with SQL.ipynb - the jupyter notebook you're currently reading.
- Audit.py - includes the update functions, as well as the intial audit used to create the update functions.
- OSM_to_CSV.py - iterates through the OSM file, calls the update functions from the audit.py file and then seperates the values into their appropriate csv file. The csv file is then checked against the schema.py for proper database schema.
- schema.py - this is a file that is the python equivelant of the database_wrangling_schema.sql that is used to verify the data is formatted properly for database upload.
- data_wrangling_schema.sql - the file that schema.py is based off, is not used but is included for reference.
- creating_db.py - this file creates the database and the tables inside. As well as checking for duplicate tables and then inserts the data from the converted .csv files into their proper table.
- queries.py - this file contains the queries used for our data exploration phase.
- sample1percent.osm - a sample of the dataset that is 1% of the size or every 100 top level lines.
- nodes.csv, nodes_tags.csv, ways.csv, ways_nodes.csv, ways_tags.csv - the csv files created from the OSM_to_CSV.py file

Problems Encountered:

After taking a small sample of the dataset using Sample_Streetmap.py, I used Audit.py to audit the sample data. I noticed following problems:

- Abbreviated street types (‘St’, ‘ST’, ‘Pkwy’, ‘Ctr’, etc).
- Abbreviated street directional indicators, such as N for North and S for South, like N. Broadway, instead of North Broadway.
- Inconsistent street types and directional indicators, if all types were abbreviated, but since they weren't I decided to correct them to their non abbreivated forms.
- Fixing the abbreivated street directional indicators caused an issue when other streets such as Route N would be changed to Route North. This was incorrect and so we had to find a way to fix the directional indicators while leaving the Routes and Highways with N, S, E, or W alone.
- Surprisingly, there was only a single postal code issue. Unfortunately this meant we still needed to correct all of them.
- City names were often plain wrong, in the case of one being called "drive-through" and we also had the issue of many different versions of the Saint prefix which is very common in this dataset. So we had to fix the incorrect ones and standardize any inconsistencies.
- Time, auditing 176MB OSM file takes a VERY long time when done iteratively.

In [3]:

```
expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane", "Road",
            "Trail", "Parkway", "Commons"]
```

In [7]:

```
#Makes a dictionary of all of the street types to allow us to create a list to update

def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)

#If the element key for 'k' is 'addr:street', return the associated value pair
def is_street_name(elem):
    return (elem.attrib['k'] == "addr:street")


""" Here we create a dictionary of type set called street_types,
and turn the open function into a variable for ease of use in the future
Next is my pride and joy, instead of using "for et.iterparse" to iterate directly line by line
through the file instead we use the osm_file var to open the file in memory, and
then turn it into an iterable. This saves a TON of time, as we can iterate on the file
in memory instead of iterating the file line by line. Once we do this, we then iterate through and
for each tag that matches "node" or "way", we check if it is a street name, and if so we run the audit_
street_types function.
we then clear the root tree, saving memory and time, close the file, and return the updated street_type
s dict.
"""

def audit_s(osmfile):
    street_types = defaultdict(set)
    osm_file = open(osmfile, "r")

    # get an iterable
    iterable = ET.iterparse(osm_file, events=("start", "end"))

    # turn it into an iterator
    iterable = iter(iterable)

    # get the root element
    event, root = iterable.next()

    for event, elem in iterable:
        if event == "end" and (elem.tag == "node" or elem.tag == "way"):
            for tag in elem.iter("tag"):
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])
        root.clear()

    osm_file.close()
    return street_types
""" The update_street function takes the information we learned from the audit_s function
and utilizes that to check a manually created mapping dictionary and DONT_UPDATE tuple.
These two objects are created by reading the report from audit_s and choosing how we want to standardiz
e the types.
to go above and beyond, we also standardized prefixes such as N for North. Unfortunely this caused an i
ssue where Highway or Route,
which often had the suffix N would be incorrectly corrected to North, such as Route North.
Therefore we created the DON_UPDATE tuple, and check each value against the tuple, and if there is a ma
tch
the value is not updated. To fix the street types, we broke the value into parts
seperated by whitespace using .split(), then change the value if it matches the key found in mapping, t
o the paired value.
Finally, the seperated parts are then rejoined with a space inbetween using the .join() function.
"""
def update_street(name):
    mapping = {"St": "Street",
            "Rd.": "Road",
            "Rd": "Road",
            "N.": "North",
            "N": "North",
            "S.": "South",
            "Blvd": "Boulevard",
            "Blvd.": "Boulevard",
            "Expy": "Expressway",
            "Ln": "Lane",
            "Ctr": "Center",
            "Ctr.": "Center",
            "5th": "Fifth",
            "4th": "Fourth",
            "3rd": "Third",
            "2nd": "Second",
            "1st": "First",
            #There was a street named just dade...that's it..so I went on google to find the real addres
s, so this corrects that occurance.
            "Dade": "South Dade Avenue",
            "MO-94": "Highway 94"
           }

    DONT_UPDATE = ('route','suite')

    if name.lower().startswith(DONT_UPDATE):
        return name
    else:
        return ' '.join(mapping.get(part, part).title() for part in name.split())
```

Incorrect Postal Codes

There was a single postal code which was obviously a user error, which the user accidentally added an extra digit to the end this was fixed in the audit.py

In [ ]:

```
def dicti(data, item):
    """This function creates a dictionary where postcodes can be held.
    The dictionary key will be the postcode itself and the dictionary value
    is a count of postcodes that were repeated throughout the dataset."""
    data[item] += 1

#This function returns the elem if 'k' matches "addr:postcode"
def is_postcode(elem):
    return (elem.attrib['k'] == "addr:postcode")

#This codes is identical in function the the street function of similar name
def audit_p(osmfile):
    osm_file = open(OSMFILE, "r")
    data = defaultdict(int)

    # get an iterable
    iterable = ET.iterparse(osm_file, events=("start", "end"))

    # turn it into an iterator
    iterable = iter(iterable)

    # get the root element
    event, root = iterable.next()

    for event, elem in iterable:
        if event == "end" and (elem.tag == "node" or elem.tag == "way"):
            for tag in elem.iter("tag"):
                if is_postcode(tag):
                    dicti(data, tag.attrib['v'])
        root.clear()
    osm_file.close()
    return data

# This is the function that actually changes the post code to the proper values
# It is called in the OSM_to_XML file, when writing the changes to the .csv

def update_postcode(postcodes):
    output = list()

    if re.search(postcodes_re, postcodes):
        new_zip = re.search(postcodes_re, postcodes).group(1)
        output.append(new_zip)

    return ', '.join(str(x) for x in output)
```

Incorrect and Inconsistent City Names

In [ ]:

```python
#Once again, this is similar in function to audit_street
def audit_city(city_dict, city_ex):
    m = cities_re.search(city_ex)
    if m:
        city_group = m.group()
        city_dict[city_group].add(city_ex)

#Same function as is_postcode, but for addr:city
def is_city(elem):
    return (elem.attrib['k'] == "addr:city")

#Same function as audit_s, but for city values.
def audit_C(osmfile):
    city_dict = defaultdict(set)
    osm_file = open(osmfile, "r")

    # get an iterable
    iterable = ET.iterparse(osm_file, events=("start", "end"))

    # turn it into an iterator
    iterable = iter(iterable)

    # get the root element
    event, root = iterable.next()

    for event, elem in iterable:
        if event == "end" and (elem.tag == "node" or elem.tag == "way"):
            for tag in elem.iter("tag"):
                if is_city(tag):
                    audit_city(city_dict, tag.attrib['v'])
        root.clear()
    osm_file.close()
    return city_dict

""" Same function as the update_street, except instead of it skipping the
the matched tuple, instead it instead uses the ofallon_mapping dict to correct the
inconsistency of some cities being listed as O'fallon and some as O fallon.
"""
def update_city(name):
    OFALLON = ('o')
    ofallon_mapping = {"O": "O'"}
    city_mapping = {"St": "Saint",
                "St.": "Saint",
                "bridgeton" : "Bridgeton",
                "drive-through": "O'Fallon",
                "Bass": "Saint",
                "Pro": "Charles",
                "Drive": "",
                "UNINCORPORATED": "Saint Peters",
                }

    if name.lower().startswith(OFALLON):
        return ''.join((ofallon_mapping.get(part, part)).title() for part in name.split())
    return ' '.join((city_mapping.get(part, part)).title() for part in name.split())
```

Prep for database - SQLite

The next step is to prepare the data to be inserted into a SQL database using OSM_to_CSV.py. To do so I parsed the elements in the OSM XML file, checking for any problem characters, before breaking each tag into seperate parts, then putting each of these parts into a seperate dictionary. These dictionaries were then used to create individual csv files with each csv file correlating to a seperate table in the soon to be established database. These csv files can then easily be imported to a SQL database as tables. The “shape_element()” function is used to transform each element in the correct format, the process_map() function then is used to pull the "shaped elements" from the dictionaries created earlier and write them to their own csv files.

In [ ]:

```python
def shape_element(element, node_attr_fields=NODE_FIELDS, way_attr_fields=WAY_FIELDS,
                  problem_chars=PROBLEMCHARS, default_tag_type='regular'):
    """Clean and shape node or way XML element to Python dict"""
    node_attribs = {}
    way_attribs = {}
    way_nodes = []
    tags = []  # Handle secondary tags the same way for both node and way elements

    if element.tag == "node":
        for item in NODE_FIELDS:
            try:
                node_attribs[item] = element.attrib[item]
            except:
                node_attribs[item] = "9999999"
    #Iterating Through 'tag' elements
        for tagz in element.iter('tag'):
            tk = tagz.attrib['k']
            tv = tagz.attrib['v']
            if not problem_chars.search(tk):
                tag_dict_node = {}
                tag_dict_node['id'] = element.attrib['id']

                # Calling the street_update function to clean up problematic
                # street names based on Audit.py file.
                if is_street_name(tagz):
                    better_name_node = update_street(tv)
                    tag_dict_node['value'] = better_name_node

                # Calling the update_postcode function to clean up problematic
                # postcodes based on Audit.py file.
                elif is_postcode(tagz):
                    better_postcode_node = update_postcode(tv)
                    tag_dict_node['value'] = better_postcode_node

                # Calling the update_postcode function to clean up problematic
                # postcodes based on Audit.py file.
                elif is_city(tagz):
                    better_city_node = update_city(tv)
                    tag_dict_node['value'] = better_city_node

                else:
                    tag_dict_node['value'] = tv

                if ':' not in tk:
                    tag_dict_node['key'] = tk
                    tag_dict_node['type'] = 'regular'
                # Dividing words before and after a colon ':'
                else:
                    tk_split = tk.split(":")
                    if len(tk_split) == 2: #If the key was an empty field
                        tag_dict_node['key'] = tk_split[1]

                    elif len(tk_split) == 3:
                        tag_dict_node['key'] = tk_split[1] + ":" + tk_split[2]
                    else:
                        tag_dict_node['key'] = tk

                    if len(tk_split) >= 2: #If the key was an empty field
                        tag_dict_node['type'] = tk_split[0]
                    else:
                        tag_dict_node['type'] = 'regular'
                tags.append(tag_dict_node)
        print({'node': node_attribs})
        return {'node': node_attribs, 'node_tags': tags}

    elif element.tag == 'way':
        for item in WAY_FIELDS:
            try:
                way_attribs[item] = element.attrib[item]
            except:
                way_attribs[item] = "9999999"

        # Iterating through 'tag' tags in way element.
        for tagz in element.iter('tag'):
            wk = tagz.attrib['k']
            wv = tagz.attrib['v']
            if not problem_chars.search(wk):
                tag_dict_way = {}
                tag_dict_way['id'] = element.attrib['id']
                tag_dict_way['id'] = element.attrib['id']

                # Calling the street_update function to clean up problematic
                # street names based on audit.py file.
                if is_street_name(tagz):
                    better_name_way = update_street(wv)
                    tag_dict_way['value'] = better_name_way

                # Calling the update_postcode function to clean up problematic
                # postcodes based on audit.py file.
                elif is_postcode(tagz):
                    better_postcode_way = update_postcode(wv)
                    tag_dict_way['value'] = better_postcode_way

                # Calling the update_postcode function to clean up problematic
                # postcodes based on audit.py file.
                elif is_city(tagz):
                    better_city_way = update_city(wv)
                    tag_dict_way['value'] = better_city_way

                # For other values that are not street names or postcodes.
                else:
                    tag_dict_way['value'] = wv

                if ':' not in wk:
                    tag_dict_way['key'] = wk
                    tag_dict_way['type'] = 'regular'
                #Dividing words before and after a colon ':'
                else:
                    wk_split = wk.split(":")

                    if len(wk_split) == 2: #If the key was an empty field
                        tag_dict_way['key'] = wk_split[1]

                    elif len(wk_split) == 3:
                        tag_dict_way['key'] = wk_split[1] + ":" + wk_split[2]
                    else:
                        tag_dict_way['key'] = tk

                    if len(wk_split) >= 2: #If the key was an empty field
                        tag_dict_way['type'] = wk_split[0]
                    else:
                        tag_dict_way['type'] = 'regular'

                tags.append(tag_dict_way)

    # Iterating through 'nd' tags in way element.
        for index, tagz in enumerate(element.iter('nd')):
            tag_dict_nd = {}
            tag_dict_nd['id'] = element.attrib['id']
            tag_dict_nd['node_id'] = tagz.attrib['ref']
            tag_dict_nd['position'] = index

            way_nodes.append(tag_dict_nd)

        return {'way': way_attribs, 'way_nodes': way_nodes, 'way_tags': tags}
```

Creating SQLite Database and Tables

I made use of python sqlite3 library to create a flatfile SQLite database. I created nodes, nodes_tags, ways, ways_tags, ways_nodes tables and parsed the csv files to fill the respective tables. Also of note I included DROP TABLE IF EXISTS before each table was created. This was to make updating the database much simpler whenever testing code, as I often included bogus fields or rows to test or edited the file itself and time or processing constraints werent a worry so it was much easier then performing an UPDATE statement or other methods that would be more commonly used in real world scenarios. This removed a major concern, which was corruption, duplication, or otherwise major integrity issues with the database itself. I also included a CREATE TABLE IF NOT EXISTS statement, which would double of the safety of the previous measure by causing the database to not create a table if the previous DROP TABLE IF EXISTS didn't run properly.

Below is a sample of the code found in the creating_db.py file:

In [8]:

```
import sqlite3
import csv

db = 'osm_stchas.sqlite'

# Connecting to the database
con = sqlite3.connect(db)
con.text_factory = str
cursor = con.cursor()
```

In [9]:

```
# Here we drop the nodes_tags table if it exists to save us from data integrity issues when rerunning t
his file.
cursor.execute('''
    DROP TABLE IF EXISTS nodes_tags
''')
con.commit()

# Here we create nodes_tags table.
cursor.execute('''
    CREATE TABLE IF NOT EXISTS nodes_tags(id INTEGER, key TEXT, value TEXT, type TEXT)
''')
con.commit()

with open('nodes_tags.csv', 'rb') as f:
    dr = csv.DictReader(f)
    iterate_db = [(i[b'id'], i[b'key'], i[b'value'], i[b'type']) for i in dr]

# Lets go ahead and insert the data into the nodes_tags tables from the 'nodes_tags.csv' file.
cursor.executemany('INSERT INTO nodes_tags(id, key, value, type) VALUES(?, ?, ?, ?);', iterate_db)
con.commit()
```

Finally, we can actually explore the data now!

File size:

osm_stchas.sqlite --- 125.6MB

In [ ]:

```
# Number of Nodes
def number_of_nodes():
    output = cursor.execute('SELECT COUNT(*) FROM nodes')
    return output.fetchone()[0]
print('Number of nodes: %d' % (number_of_nodes()))
```

In [ ]:

```
Number of nodes: 723413
```

In [ ]:

```
# Number of Ways
def number_of_ways():
    output = cursor.execute('SELECT COUNT(*) FROM ways')
    return output.fetchone()[0]
print('Number of ways: %d' %(number_of_ways()))
```

In [ ]:

```
Number of ways: 70274
```

In [ ]:

```
# Number of Unique Users
def number_of_unique_users():
    output = cursor.execute('SELECT COUNT(DISTINCT e.uid) FROM \
                            (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways) e')
    return output.fetchone()[0]
print('Number of unique users: %d' %(number_of_unique_users()))
```

In [ ]:

```
Number of unique users: 846
```

In [ ]:

```
# Query for Top 10 Amenities in St Charles
query = "SELECT value, COUNT(*) as num FROM nodes_tags \
            WHERE key='amenity' \
            GROUP BY value \
            ORDER BY num DESC \
            LIMIT 10"

# Top 10 Amenities in St Charles
def top_ten_amenities_in_st_charles():
    output = cursor.execute(query)
    pprint(output.fetchall())
    return None

print('Top 10 Amenities:\n')
top_ten_amenities_in_st_charles()
```

In [ ]:

```
Top 10 Amenities:

[(u'fast_food', 68),
 (u'place_of_worship', 66),
 (u'restaurant', 62),
 (u'school', 61),
 (u'bench', 44),
 (u'parking', 36),
 (u'toilets', 26),
 (u'fountain', 26),
 (u'bank', 21),
 (u'drinking_water', 19)]
```

In [ ]:

```
# Type of religions that each place_of_worship value returned
query = "SELECT value, COUNT(*) as num FROM nodes_tags \
            WHERE key='religion' \
            GROUP BY value"

def types_of_religion():
    output = cursor.execute(query)
    pprint(output.fetchall())
    return None

print('Different types of shops:\n')
types_of_religion()
```

In [ ]:

```
Different types of shops:

[(u'christian', 59), (u'multifaith', 1), (u'muslim', 1)]
```

Additional Ideas for Improvement and Anticipated Problems :

After the above review and analysis, there are a few stand out features as a resident of this area myself. First, why do they simply say christian? A significant amount of these are Catholic and another large chunk are non-demonimational. So greater detail should be applied on standardizing labels applied. Also, muslim isn't a religion. It's a description of a Follower of the religion of Islam.

Another thing that can be improved, is creating a list or dictionary like we did for the mapping of common brand name or often misspelled bussinesses to ensure that there is a standardization of naming.

However, lastly, the most important and certainly most difficult aspect would be keeping this dataset up to date. Since the dataset is not a cached or saved dataset and businesses and roads continue to change and update it would be an enormous responsibility to keep the data clean AND real time. This is a very eye opening project to size and scale of the jobs that we generalize as "big data".

References

For most of the python help, I took advantage of four sources:

- https://stackoverflow.com
- https://docs.python.org/
- Multiple users and moderators of the official Python Discord channel. With specific debuging and optimization help (cutting off an insane 700 seconds of runtime) from users salt-die and Dan6erbond.
- Alumni from this program and WGU, including Michael Kuehn and Pranav Suri and Sriram Jaju, via their personal githubs, and linked in messages
  - https://sriramjaju.github.io/2017-06-16-openstreetmap-data-wrangling-with-sql/
  - https://github.com/pranavsuri/Data-Analyst-Nanodegree
  - https://github.com/mkuehn10/P3-Wrangle-OpenStreetMap-Data

For sql I didn't need much help, but often used https://www.w3schools.com/sql/default.asp for reminders.