OpenStreetMap Data Project - Wrangling Map Data for Sydney, Australia

February 6, 2018

1 OpenStreetMap Data Project: Sydney, Australia

1.0.1 Geographic Area

Sydney, New South Wales, Australia
https://mapzen.com/data/metro-extracts/metro/sydney_australia/
https://www.openstreetmap.org/relation/5750005

The map dataset chosen for this project is the Sydney, Australia metro area dataset from MapZen. I decided on this location because of a study abroad program in Australia I participated in during my undergrad. I lived in and traveled the country for six months, but was unable to trek the Harbour City. I have friends and family currently living in Sydney and would like to visit the area soon. In this project I'm looking to explore the dataset in an effort to contribute to the improvement of map data on the OpenStreetMap database website, gain some additional geographic familiarity and uncover some interesting facts about the city.

1.0.2 Creating a Sample Dataset

Before auditing and cleaning the original dataset, I found it useful to create a smaller sample file in order to validate and work through any issues discovered. This allowed for a more streamlined process when writing, running, and correcting a number of python scripts and SQL queries. The sample file created was 6.9 MB in size and this process proved to be a time saver, especially in the case of the CSV conversion script. Once the code was suitable and successfully run on the sample file, it was applied to the original uncompressed OSM XML file and resulting tables created from the CSV conversion files.

The method was utilized in assessing the quality of the data for validity, accuracy, completeness, consistency and uniformity and ultimately the effort to clean and standardize the dataset. I identified several language errors in the location data including spelling, abbreviations, and typos.

```
In [ ]: #sampleconversion.py
        #Taking a systematic sample of elements from the original OSM region (Sydney, Australi

        import xml.etree.ElementTree as ET  #Use cElementTree or lxml if too slow

        OSM_FILE = "sydney_australia.osm"
```

```
SAMPLE_FILE = "sydney_sample.osm"  #Create new file name for created sample.

k = 50 # Parameter: take every k-th top level element

def get_element(osm_file, tags=('node', 'way', 'relation')):
    """Yield element if it is the right type of tag

    Reference:
    http://stackoverflow.com/questions/3095434/inserting-newlines-in-xml-file-generate
    """
    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>')
    print
```

1.1 Data Audits

Street Name Errors Python is used to audit and print out street names in the sample dataset, containing some unusual values. The function below is used to identify and correct unexpected spelling errors, street name abbreviations and typo inconsistencies. The function takes a string with street_name as an argument and returns the corrected name, iterating over each word in an address.

In some cases mapping out more specific changes was required, especially considering the inimitable street names found nowhere else. A bit more verification was required in order to limit hypercorrections, using a popular map application. From the provided list the script audits and make changes to the 'mapping' variable in the OSM file. Therefore "ApplegumCrescent" is corrected to "Applegum Crescent".

```
In [ ]: #sydneyProcessing.py
        import xml.etree.cElementTree as ET
        from collections import defaultdict
```

```
import re
import pprint

OSMFILE = "sydney_australia.osm"
street_type_re = re.compile(r'\b\S+\.?$\xf3', re.IGNORECASE)


expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane
            "Trail", "Parkway", "Commons", "West", "East", "South", "North", "Way"]

#Correct street name abbreviations and typo (i.e., Mainland St. => Mainland Street,
#Broadway W => Broadway West, Boundary Rd. => Boundary Road)
mapping = {"St": "Street",
           "St.": "Street",
           "Rd.": "Road",
           "Ave": "Avenue",
           "west": "West",
           "W": "West",
           "street": "Street",
           "Blvd": "Boulevard",
           "Denmanstreet": "Denman Street"
           }

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)

def is_street_name(elem):
    return (elem.attrib['k'] == "addr:street")


def audit(osmfile):
    osm_file = open(osmfile, "r")
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):

        if 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'])
    osm_file.close()
    return street_types


def update_name(name, mapping):
```

```
        #makes revisions, updating street names based on mapping and what is expected.
            m = street_type_re.search(name)
            if m.group() not in expected:
                if m.group() in mapping.keys():
                    name = re.sub(m.group(), mapping[m.group()], name)
            return name


        def test():
            st_types = audit(OSMFILE)

            pprint.pprint(dict(st_types))

            for st_type, ways in st_types.iteritems():
                for name in ways:
                    better_name = update_name(name, mapping)
                    print name, "=>", better_name
                    if name == "West Lexington St.":
                        assert better_name == "West Lexington Street"
                    if name == "Baldwin Rd.":
                        assert better_name == "Baldwin Road"


        if __name__ == '__main__':
            test()
```

State Code Audit Additional auditing was performed to verify correct use and consistency of the state code. The state code for New South Wales is NSW. The results of this audit show there were no inconsistencies.

```
In [ ]: #scode.py

        import xml.etree.cElementTree as ET
        from collections import defaultdict
        import re
        import pprint

        OSMFILE = "sydney_australia.osm"
        state_type_re = re.compile(r'\b\S+\.?$\xf3', re.IGNORECASE)

        expected = ["NSW"]


        def audit_state_type(state_types, state_code):
            m = state_type_re.search(state_code)
            if m:
                state_type = m.group()
                if state_type not in expected:
```

```
                state_types[state_type].add(state_code)

    def is_state_code(elem):
        return (elem.attrib['k'] == "is_in:state_code")


    def audit(osmfile):
        osm_file = open(osmfile, "r")
        state_types = defaultdict(set)
        for event, elem in ET.iterparse(osm_file, events=("start",)):

            if elem.tag == "node" or elem.tag == "way":
                for tag in elem.iter("tag"):
                    if is_state_code(tag):
                        audit_state_type(state_types, tag.attrib['v'])
        osm_file.close()
        return state_types
        print state_types
```

A SQL query performed once the CSVs have been loaded into SQL tables confirms all 15 instances including state code are correct.

```
SELECT value, COUNT(*) as num
   FROM nodes_tags
   WHERE key='state_code'
   GROUP BY key;

NSW  15
```

XML Data Primitives The code below explores the data further by giving a count of the data primitive included in the sydney_australia.osm file. The code checks the "k" value for each "" to identify potential problems. Four regular expressions check for certain patterns in the tags, and whether we have any tags with problem characters.

The function 'key_type', provides a count of each of four tag categories in a dictionary. The four categories are "lower" for tags that contain only lowercase letters and are valid, "lower_colon" for otherwise valid tags with a colon in their names, "problemchars" for tags with problematic characters, and "other" for other tags that do not fall into the other three categories.

```
In [ ]: #sydneyProcessing.py

        import xml.etree.cElementTree as ET
        from collections import defaultdict
        import re
        import pprint

        #Tag types, a count of each of four tag categories placed in a dictionary

        lower = re.compile(r'^([a-z]|_)*$')
```

```
lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')


def key_type(element, keys):
    if element.tag == "tag":
        print(element.get('k'))
        if lower.search(element.get('k')):
            keys["lower"] += 1
        elif lower_colon.search(element.get('k')):
            keys["lower_colon"] += 1
        elif problemchars.search(element.get('k')):
            keys["problemchars"] += 1
        else:
            keys["other"] += 1
            pprint.pprint(keys)
    return keys


def process_map(filename):
    keys = {"lower": 0, "lower_colon": 0, "problemchars": 0, "other": 0}
    for _, element in ET.iterparse(filename):
        keys = key_type(element, keys)

    return keys


def test():
    keys = process_map('sydney_australia.osm')
    pprint.pprint(keys)


#if __name__ == "__main__":
#    test()
```

{'problemchars': 8, 'lower': 745578, 'other': 8614, 'lower_colon': 104685}

CSV Conversion After the auditing and cleaning steps were performed, the OSM file was converted to CSV files for inclusion in a SQL database as tables. The script below details the conversion to CSV format, which can then be converted in SQL tables.

```
In [ ]:  %load 'schema.py'
```

```
In [ ]: #saConversion.py

        import csv
        import codecs
        import pprint
```

```
import re
import xml.etree.cElementTree as ET

import cerberus

import schema

OSM_PATH = "sydney_australia.osm"

NODES_PATH = "nodes.csv"
NODE_TAGS_PATH = "nodes_tags.csv"
WAYS_PATH = "ways.csv"
WAY_NODES_PATH = "ways_nodes.csv"
WAY_TAGS_PATH = "ways_tags.csv"

LOWER_COLON = re.compile(r'^([a-z]|_)+:([a-z]|_)+')
PROBLEMCHARS = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')

SCHEMA = schema.schema

# Make sure the fields order in the csvs matches the column order in the sql table sch
NODE_FIELDS = ['id', 'lat', 'lon', 'user', 'uid', 'version', 'changeset', 'timestamp']
NODE_TAGS_FIELDS = ['id', 'key', 'value', 'type']
WAY_FIELDS = ['id', 'user', 'uid', 'version', 'changeset', 'timestamp']
WAY_TAGS_FIELDS = ['id', 'key', 'value', 'type']
WAY_NODES_FIELDS = ['id', 'node_id', 'position']


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
    p = 0

    if element.tag == 'node':
        for i in NODE_FIELDS:
           node_attribs[i] = element.attrib[i]
        for tag in element.iter("tag"):
            node_tags_attribs = {}
            temp = LOWER_COLON.search(tag.attrib['k'])
            is_p = PROBLEMCHARS.search(tag.attrib['k'])
            if is_p:
               continue
            elif temp:
```

```
                split_char = temp.group(1)
                split_index = tag.attrib['k'].index(split_char)
                type1 = temp.group(1)
                node_tags_attribs['id'] = element.attrib['id']
                node_tags_attribs['key'] = tag.attrib['k'][split_index+2:]
                node_tags_attribs['value'] = tag.attrib['v']
                node_tags_attribs['type'] = tag.attrib['k'][:split_index+1]
            else:
                node_tags_attribs['id'] = element.attrib['id']
                node_tags_attribs['key'] = tag.attrib['k']
                node_tags_attribs['value'] = tag.attrib['v']
                node_tags_attribs['type'] = 'regular'
            tags.append(node_tags_attribs)
        return {'node': node_attribs, 'node_tags': tags}
    elif element.tag == 'way':
        id = element.attrib['id']
        for i in WAY_FIELDS:
            way_attribs[i] = element.attrib[i]
        for i in element.iter('nd'):
            d = {}
            d['id'] = id
            d['node_id'] = i.attrib['ref']
            d['position'] = p
            p+=1
            way_nodes.append(d)
        for c in element.iter('tag'):
            temp = LOWER_COLON.search(c.attrib['k'])
            is_p = PROBLEMCHARS.search(c.attrib['k'])
            e = {}
            if is_p:
                continue
            elif temp:
                split_char = temp.group(1)
                split_index = c.attrib['k'].index(split_char)
                e['id'] = id
                e['key'] = c.attrib['k'][split_index+2:]
                e['type'] = c.attrib['k'][:split_index+1]
                e['value'] = c.attrib['v']
            else:
                e['id'] = id
                e['key'] = c.attrib['k']
                e['type'] = 'regular'
                e['value'] =  c.attrib['v']
            tags.append(e)


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

```
# ================================================== #
#               Helper Functions                     #
# ================================================== #
def get_element(s_file, tags=('node', 'way', 'relation')):
    """Yield element if it is the right type of tag"""

    context = ET.iterparse(s_file, events=('start', 'end'))
    _, root = next(context)
    for event, elem in context:
        if event == 'end' and elem.tag in tags:
            yield elem
            root.clear()


def validate_element(element, validator, schema=SCHEMA):
    """Raise ValidationError if element does not match schema"""
    if validator.validate(element, schema) is not True:
        field, errors = next(validator.errors.iteritems())
        message_string = "\nElement of type '{0}' has the following errors:\n{1}"
        error_string = pprint.pformat(errors)

        raise Exception(message_string.format(field, error_string))


class UnicodeDictWriter(csv.DictWriter, object):
    """Extend csv.DictWriter to handle Unicode input"""

    def writerow(self, row):
        super(UnicodeDictWriter, self).writerow({
            k: (v.encode('utf-8') if isinstance(v, unicode) else v) for k, v in row.ite
        })

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)


# ================================================== #
#               Main Function                        #
# ================================================== #
def process_map(file_in, validate):
    """Iteratively process each XML element and write to csv(s)"""

    with codecs.open(NODES_PATH, 'w') as nodes_file, \
         codecs.open(NODE_TAGS_PATH, 'w') as nodes_tags_file, \
         codecs.open(WAYS_PATH, 'w') as ways_file, \
         codecs.open(WAY_NODES_PATH, 'w') as way_nodes_file, \
```

```
        codecs.open(WAY_TAGS_PATH, 'w') as way_tags_file:

    nodes_writer = UnicodeDictWriter(nodes_file, NODE_FIELDS)
    node_tags_writer = UnicodeDictWriter(nodes_tags_file, NODE_TAGS_FIELDS)
    ways_writer = UnicodeDictWriter(ways_file, WAY_FIELDS)
    way_nodes_writer = UnicodeDictWriter(way_nodes_file, WAY_NODES_FIELDS)
    way_tags_writer = UnicodeDictWriter(way_tags_file, WAY_TAGS_FIELDS)

    nodes_writer.writeheader()
    node_tags_writer.writeheader()
    ways_writer.writeheader()
    way_nodes_writer.writeheader()
    way_tags_writer.writeheader()

    validator = cerberus.Validator()

    for element in get_element(file_in, tags=('node', 'way')):
        el = shape_element(element)
        if el:
            if validate is True:
                validate_element(el, validator)

            if element.tag == 'node':
                nodes_writer.writerow(el['node'])
                node_tags_writer.writerows(el['node_tags'])
            elif element.tag == 'way':
                ways_writer.writerow(el['way'])
                way_nodes_writer.writerows(el['way_nodes'])
                way_tags_writer.writerows(el['way_tags'])


if __name__ == '__main__':
    # Note: Validation is ~ 10X slower. For the project consider using a small
    # sample of the map when validating.
    process_map(OSM_PATH, validate=True)
```

1.2 Dataset Overview Statistics

Size of Files

```
sydney_australia.osm .... 344.2 MB
sydneyAu.db ............. 188.8 MB
nodes.csv ............... 127.3 MB
nodes_tags.csv .......... 6.2 MB
ways.csv ................ 12.4 MB
ways_tags.csv ........... 23.4 MB
ways_nodes.cv ........... 44 MB
```

Number of Unique Users

```
sqlite> SELECT COUNT(DISTINCT(e.uid))
FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways) e;

count = 2382
```

Number of Unique Nodes

```
sqlite> SELECT COUNT(*) FROM nodes;

count = 1529440
```

Number of Unique Ways

```
sqlite> SELECT COUNT(*) FROM ways;

count = 208933
```

1.2.1 Exploration of Additional Nodes

I chose to explore a number of additional node types to further the geographical exploration of the map data for consideration in the planning of a potential holiday trip. I started with a query into the top ten amenities Sydney has to offer. It is interesting to note the top amenity in Sydney is the communal bench.

Amenities:

```
sqlite> SELECT value, COUNT(*) as num
   ...> FROM nodes_tags
   ...> WHERE key='amenity'
   ...> GROUP BY value
   ...> ORDER BY num DESC
   ...> LIMIT 10;

bench             1465
restaurant        1015
cafe              884
drinking_water    836
parking           764
toilets           660
fast_food         604
bicycle_parking   588
post_box          423
place_of_worship  397
```

Everybody gets hungry. The next query provides a breakdown of the types of restaurants that call Sydney home.

Cuisine:

```
sqlite> SELECT nodes_tags.value, COUNT(*) as num
   ...> FROM nodes_tags
   ...>     JOIN (SELECT DISTINCT(id) FROM nodes_tags WHERE value='restaurant') i
   ...>     ON nodes_tags.id=i.id
   ...> WHERE nodes_tags.key='cuisine'
   ...> GROUP BY nodes_tags.value
   ...> ORDER BY num DESC
   ...> LIMIT 20;

thai                 74
chinese              64
italian              55
pizza                45
japanese             41
indian               34
vietnamese           16
korean               15
greek                12
sushi                10
regional             9
asian                8
international        8
malaysian            8
burger               7
asian;sushi;japanese 6
lebanese             6
seafood              6
fish_and_chips       5
french               5
```

This query explores data identifying religious structures, which could be used to identify potential architectural wonders worth visiting.

Liturgy, Top 3:

```
sqlite> SELECT nodes_tags.value, COUNT(*) as num
   ...> FROM nodes_tags
   ...>     JOIN (SELECT DISTINCT(id) FROM nodes_tags WHERE value='place_of_worship') i
   ...>     ON nodes_tags.id=i.id
   ...> WHERE nodes_tags.key='religion'
   ...> GROUP BY nodes_tags.value
   ...> ORDER BY num DESC
   ...> LIMIT 3;

christian  338
buddhist   9
muslim     9
```

The following query shows many different tourist attractions worth visiting. I'm impressed by the 327 viewpoints located throughout the city. Many of which provide glorious beach vistas.

Tourism, Top 10:

```
sqlite> SELECT nodes_tags.value, COUNT(*) as num
   ...> FROM nodes_tags
   ...>     JOIN (SELECT DISTINCT(id) FROM nodes_tags WHERE key='tourism') i
   ...>     ON nodes_tags.id=i.id
   ...> WHERE nodes_tags.key='tourism'
   ...> GROUP BY nodes_tags.value
   ...> ORDER BY num DESC
   ...> LIMIT 10;

information  341
viewpoint    327
picnic_site  232
hotel        130
attraction   123
artwork      121
museum       51
hostel       43
camp_site    21
motel        17
```

City art is explored in this query to provide some potential mid-point attractions while walking the city streets from one attraction to the next. City art also draws a nice contrast with artwork contained in museums.

City Art

```
sqlite> SELECT nodes_tags.value, COUNT(*) as num
   ...> FROM nodes_tags
   ...>     JOIN (SELECT DISTINCT(id) FROM nodes_tags WHERE key='artwork_type') i
   ...>     ON nodes_tags.id=i.id
   ...> WHERE nodes_tags.key='artwork_type'
   ...> GROUP BY nodes_tags.value
   ...> ORDER BY num DESC;

sculpture     39
statue        20
mural         6
interactive   3
gates         1
graffiti      1
installation  1
sculptor      1
streetart     1
```

In the interest of exploring how the data is added to the map dataset I thought we could investigate the top contributing users. From the results below, we can see the top ten contributing users out of the 2382 total unique users. Based on these findings, user "balcoath" made 18.7% of the total contributions for the top ten users.

1.2.2 Top Contributing Users

```
sqlite> 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;

balcoath    117344
inas        89241
TheSwavu    74266
aharvey     66423
ChopStiR    60487
ozhiker2    51861
"Leon K"    46508
cleary      41601
Rhubarb     40735
AntBurnett  37524
```

1.2.3 Potential Areas for Improvement

Data integrity and the standardization of data is paramount in being able to perform reliable data analysis, and some of the more common errors in this dataset appear to be the result of human error. As is likely the case with the street name errors identified. Perhaps creating a data-entry template, Readme file, or a data type verification process via a java application could be utilize when users are manually entering data into the OpenStreetMap databases.

This would likely prevent any mistakes before they are entered into the database, making the need to routinely audit and clean the data less cumbersome. However, some anticipated problems with this suggestion include the likelihood that some users wouldn't read the supporting materials or even ignore the instructions altogether. This might be avoided with solid and complete input from a GPS device.

1.2.4 Conclusions

In the effort to reveal insights into the diversity of religious structures in the city, the "places of worship" value tag in the nodes_tag table was explored. For the purposes here, we could use these results to gain additional insight in the makeup of religious structures, and potentially identify some of the more popular religious structures worth visiting.

While reviewing the results I couldn't help consider the lack of representation of those who don't affiliate with any religion. This is likely the result of this population not having a central place of worship. Gathering additional data in the proportional religious affiliation of the citizens of Sydney for comparison would provide a better glimpse into the religious makeup of the city.

All the data explored in this project has provided some interesting results, which are sure to prove helpful in the planning of a future holiday to Sydney, Australia.