Data Wrangling (with MongoDB)

This is a report summarising my work carried out in wrangling Open Street Map data with Python and MongoDB.

Lesson 6 Answers

The answers for the exercises in Lesson 6 of the course are provided in Appendinx A. Comment blocks have been removed for brevity.

Project

For my dataset, I used the Overpass Api to get OpenStreetMap data for the area around Greenwich, London in the United Kingdom. I wanted to find the most popular cuisines served by food businesses in the area. The raw xml file was just over 120MB in size.

Initial Exploration

As part of exploration, I used code from Lesson 6, but pointing to the target data file instead. Using code from Exploring Users in Appendix A, I saw that there are contributions from 1430 unique users in my data set. I used the following code to get an idea of the overall data set:

```
import xml.etree.cElementTree as ET
import pprint
import collections

def count_tags(filename):
    tags = collections.Counter()
    for event, elem in ET.iterparse(filename, ("start",)):
        tags[elem.tag] += 1
    return tags

def test():
    tags = count_tags('greenwich.osm')
    pprint.pprint(tags)

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

I got the following numbers:

Tag	Count
Tag	608882
Nd	565645
Node	453930
Member	86244
Way	80145
Relation	2703
Note	1
Meta	1
Osm	1

This is a moderate size of data, and simply counting tags took a notable amount of time (though not considerably long). I could have explored the data further with Python, and started cleaning it up, but decided to import the data into MongoDB and use it for exploration, cleaning, processing, and querying. This allowed me to try out different things quickly using MongoDB's query language, without having to write repetitive xml processing in Python.

Into Mongo

I used the code from Lesson 6 (Preparing for Database) to first convert the xml osm file to a json one. The resultant file was 96.9MB. I used the following command to import it into MongoDB into a collection called "raw1":

mongoimport --db osm --collection raw1 --file greenwich.json

For further processing, I use a script written in JavaScript in a file. I launched the mongo shell, and loaded this file to run queries. This allowed me to progressively improve the script quickly and easily.

Data Overview

I carried out some basic exploration of the data set. The details are given here:

- Size of the greenwich.osm: 120MB.
- Size of greenwich.osm.json: 96.9MB.

I carried out the following MongoDB queries to get some statistics:

Total Number of Documents:

```
> db.raw1.find().count()
456226
```

Number of Unique Users:

```
> db.raw1.distinct("created.user").length
1190
```

Number of Nodes:

```
> db.raw1.find({'type':'node'}).count()
453908
```

Number of Ways:

```
> db.raw1.find({'type':'way'}).count()
2296
```

Number of Restaurants:

```
> db.raw1.find({'type':'node', 'amenity':'restaurant'}).count()
349
```

Exploring Cuisines

I wanted to find the most popular cuisine served by businesses in the area. With some querying, I established that there was no single business type ("amenity") covering all food. For example, "restaurant", and "fast_food" were different types of amenities, but they both were related to businesses serving food. Upon further exploration, I established that "cuisine" was a definitive

property that was on all businesses serving food. I counted the number of food serving businesses using the following MongoDB query:

```
var cuisine_entries = db.raw1.find({'cuisine': {$exists:1}}).length();
print(cuisine_entries);
```

This gave me 465 businesses. I did a simple count of each cuisine using the following aggregation:

```
var cuisines = db.raw1.aggregate([
        {$match: {'cuisine': {$exists:1}}},
        {$group: { _id: "$cuisine", 'count': {$sum:1} } }
    ]);

cuisines.forEach(printjson);
```

This showed initial promise, however there were a few issues:

- Cuisines were mixed case. For example, British and british were considered different.
- Some entries had multiple cuisines separated by "&", ",", or ";".
- Some entries had leading or trailing underscores. For example, "_steak".
- For a single entry, there was redundant text. This cuisine was "british_food", which is obviously in the category of the much more frequent "british" cuisine.

Data Cleansing

I decided to clean the data set, and store all entries related to cuisines in a separate collection. I noticed that further aggregation would be done on the "cuisine" field, and as such, added an index to it. I also converted the "cuisine" field to be an array, instead of a delimiter separated string. I then proceeded to put cleaned up cuisine data in the new collection:

```
var cleanup_cuisine_name = function(c){
    var lowered = c.toLowerCase();
    var split = lowered.split(/[,;&]+/)
    for(var s in split){
        split[s] = split[s].trim()
        if(split[s] && split[s][0] == '_') split[s] = split[s].substring(1);
        if(split[s] && split[s][split[s].length-1] == '_')
            split[s] = split[s].substring(0, split[s].length-1);
        if(split[s] == "british_food") split[s] = "british";
    }

    return split;
}

db.createCollection("cuisines");
db.cuisines.createIndex({cuisine:1});

db.raw1.find({'cuisine': {$exists:1}}).snapshot().forEach(function(el){
    el.cuisine = cleanup_cuisine_name(el.cuisine);
    db.cuisines.save(el);
});
```

Data Cleansing 2

As a second activity, I wanted to create another raw dataset with entries that are limited to the Royal Borough of Greenwich by their post code. This required filtering out the entries outside of Greenwich. I put the cleaned data in another collection called “greenwich”. I used the following query to achieve this:

```
db.raw1.aggregate([{
                $match: {
                "address.postcode" : {
        $regex:/(SE2|SE3|SE7|SE8|SE9|SE10|SE12|SE13|SE18|SE28|DA15|DA16|BR7)\s.*/
                        }
                }
        },
        {$out:"greenwich"}
]);
```

The post code prefixes were found from Wikipedia.

Data Aggregation

With the data prepared, I used MongoDB’s aggregation features to create a sorted list of cuisines in decreasing count order. I put the results in a new collection for efficient querying. I used the following code to do this:

```
db.cuisines.aggregate([
                {$match: {'cuisine': {$exists:1}}},
                {$group: { _id: "$cuisine", 'count': {$sum:1} } },              //one entry per array
                {$unwind: "$_id"},      //one entry per cuisine per array from last step
                {$group: {_id:"$_id", count: {$sum: "$count"}}},          //counts
                {$sort: {count: -1}},                                     //descending order
                {$out: "popular_cuisines"}                                //output to new collecion
        ]);
```

Querying Data

With the data ready, I used the following to see the top 10 cuisines:

```
> db.popular_cuisines.find().limit(10)
{ "_id" : "sandwich", "count" : 51 }
{ "_id" : "chinese", "count" : 47 }
{ "_id" : "indian", "count" : 34 }
{ "_id" : "italian", "count" : 31 }
{ "_id" : "coffee_shop", "count" : 27 }
{ "_id" : "pizza", "count" : 22 }
{ "_id" : "burger", "count" : 21 }
{ "_id" : "chicken", "count" : 21 }
{ "_id" : "japanese", "count" : 21 }
{ "_id" : "fish_and_chips", "count" : 20 }
```

Other Ideas

Exploring the data set further, I noticed that many entries have a “wheelchair” field, that is either “yes”, “no”, or “limited”:

> db.raw1.aggregate([{$match: {"wheelchair":{$exists:1}}}, { $group: {_id:"$wheelchair", count: {$sum:1}} }])

{ "_id" : "yes", "count" : 221 }
{ "_id" : "no", "count" : 58 }
{ "_id" : "limited", "count" : 22 }

This information can be used to find institutions (like train stations, restaurants), etc. that are wheelchair accessible. We could create a geospatial index on “pos” (which is a 2d array) to enable querying for wheelchair accessible institutions near any particular point. This could then be put behind an API, and served via a mobile application. Such a system could benefit wheelchair users in finding restaurants that they can easily access via accessible train stations. As the quality of information in the chosen area is high, this information could be gathered easily. And since there are a finite number of businesses and train stations, scalability should not be a problem. However, there would still be challenges. Quite often, there are disruptions in service, and public transport is diverted via other routes. There are also cases where station lifts are down for maintenance, and though the station remains in service, it is no longer accessible until the lift service resumes. Such information is available via Transport for London’s APIs, and that would need to be leveraged to augment the osm information in a near real time manner. Otherwise, customers could be disappointed after having planned a meal at a restaurant.

Appendix A: Lesson Answers

Iterative Parsing

```
import xml.etree.cElementTree as ET
import pprint
import collections

def count_tags(filename):
    tags = collections.Counter()
    for event, elem in ET.iterparse(filename, ("start",)):
        tags[elem.tag] += 1
    return tags


def test():
    tags = count_tags('example.osm')
    pprint.pprint(tags)
    assert tags == {'bounds': 1,
                    'member': 3,
                    'nd': 4,
                    'node': 20,
                    'osm': 1,
                    'relation': 1,
                    'tag': 7,
                    'way': 1}



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

Data Model

The second option (with the nested document structure) would be my preferred option.

Tag Types

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

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":
        k = element.attrib["k"]
        if lower.match(k):
            keys["lower"] += 1
        elif lower_colon.match(k):
```

```
            keys["lower_colon"] += 1
        elif problemchars.match(k):
            keys["problemchars"] += 1
        else:
            keys["other"] += 1

    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('example.osm')
    pprint.pprint(keys)
    assert keys == {'lower': 5, 'lower_colon': 0, 'other': 2, 'problemchars': 0}


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

Exploring Users

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

def get_user(element):
    return

def process_map(filename):
    users = set()
    for _, element in ET.iterparse(filename):
        if "uid" in element.attrib:
            users.add(element.attrib["uid"])
    return users
def test():
    users = process_map('example.osm')
    pprint.pprint(users)
    assert len(users) == 6

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

Improving Street Names

```
import xml.etree.cElementTree as ET
from collections import defaultdict
```

```
import re
import pprint

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


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

# UPDATE THIS VARIABLE
mapping = { "St": "Street",
            "St.": "Street",
            "Rd.": "Road",
            "Ave": "Avenue"
          }

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'])

    return street_types


def update_name(name, mapping):
    for k in mapping.keys():
        if name.endswith(k):
            return name.replace(k, mapping[k])

    return name

def test():
    st_types = audit(OSMFILE)
```

```
    assert len(st_types) == 3
    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()
```

Preparing for Database

```
import xml.etree.cElementTree as ET
import pprint
import re
import codecs
import json
lower = re.compile(r'^([a-z]|_)*$')
lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')
CREATED = [ "version", "changeset", "timestamp", "user", "uid"]
def shape_element(element):

    node = {}

    if element.tag == "node" or element.tag == "way" :

        for key in element.attrib.keys():

            val = element.attrib[key]

            node["type"] = element.tag

            if key in CREATED:

                if not "created" in node.keys():

                    node["created"] = {}

                node["created"][key] = val

            elif key == "lat" or key == "lon":

                if not "pos" in node.keys():

                    node["pos"] = [0.0, 0.0]

                old_pos = node["pos"]

                if key == "lat":
```

```
            new_pos = [float(val), old_pos[1]]
        else:
            new_pos = [old_pos[0], float(val)]
        node["pos"] = new_pos
    else:
        node[key] = val
    for tag in element.iter("tag"):
        tag_key = tag.attrib['k']
        tag_val = tag.attrib['v']
        if problemchars.match(tag_key):
            continue
        elif tag_key.startswith("addr:"):
            if not "address" in node.keys():
                node["address"] = {}
            addr_key = tag.attrib['k'][len("addr:") : ]
            if lower_colon.match(addr_key):
                continue
            else:
                node["address"][addr_key] = tag_val
        elif lower_colon.match(tag_key):
            node[tag_key] = tag_val
        else:
            node[tag_key] = tag_val
for tag in element.iter("nd"):
    if not "node_refs" in node.keys():
        node["node_refs"] = []
    node_refs = node["node_refs"]
    node_refs.append(tag.attrib["ref"])
    node["node_refs"] = node_refs

return node
```

```
    else:
        return None


def process_map(file_in, pretty = False):
    # You do not need to change this file
    file_out = "{0}.json".format(file_in)
    data = []
    with codecs.open(file_out, "w") as fo:
        for _, element in ET.iterparse(file_in):
            el = shape_element(element)
            if el:
                data.append(el)
                if pretty:
                    fo.write(json.dumps(el, indent=2)+"\n")
                else:
                    fo.write(json.dumps(el) + "\n")
    return data

def test():
    data = process_map('example.osm', True)
    pprint.pprint(data)

    correct_first_elem = {
        "id": "261114295",
        "visible": "true",
        "type": "node",
        "pos": [41.9730791, -87.6866303],
        "created": {
            "changeset": "11129782",
            "user": "bbmiller",
```

```
        "version": "7",
        "uid": "451048",
        "timestamp": "2012-03-28T18:31:23Z"
      }
    }
    assert data[0] == correct_first_elem
    assert data[-1]["address"] == {
                        "street": "West Lexington St.",
                        "housenumber": "1412"
                         }
    assert data[-1]["node_refs"] == [ "2199822281", "2199822390",  "2199822392", "2199822369",
                        "2199822370", "2199822284", "2199822281"]


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