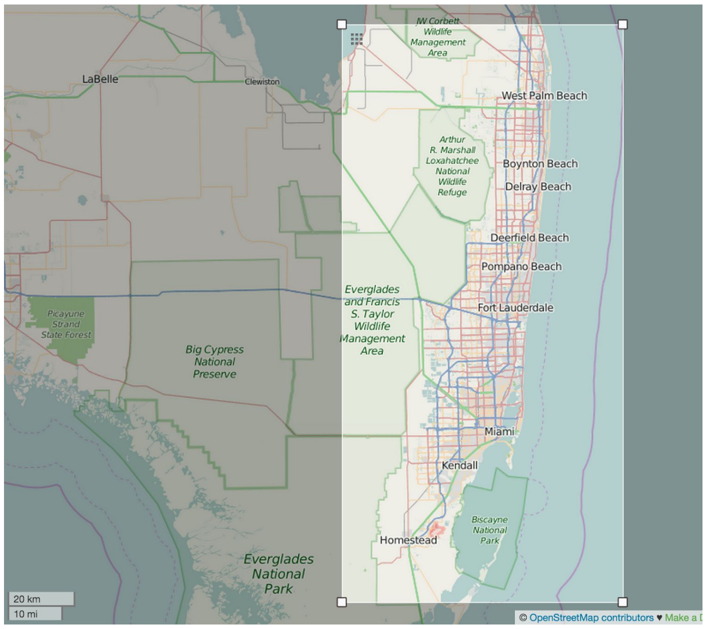
Udacity Data Analysis Nanodegree - David Smith - October 7, 2015

Project 3: Wrangle Open Street Map Data

Miami, Florida, United States


JW Corbett Wildlife Management Area
LaBelle
Clewiston
West Palm Beach
Arthur R. Marshall Loxahatchee National Wildlife Refuge
Boynton Beach
Delray Beach
Deerfield Beach
Pompano Beach
Everglades and Francis S. Taylor Wildlife Management Area
Fort Lauderdale
Picayune Strand State Forest
Big Cypress National Preserve
Miami
Kendall
Biscayne National Park
Homestead
Everglades National Park
20 km
10 mi
© OpenStreetMap contributors

Area Analyzed

I chose to analyze the metropolitan area of Miami, Florida, in the United States. Pictured above, this area stretches along the southeastern coast of the Florida peninsula from Homestead and Biscayne National Park in the South to Lake Okeechobee in the North, and from the Everglades National Park in the West to the Atlantic Ocean in the East. I chose this area because it includes the city where I live, Coral Gables.
https://www.openstreetmap.org/relation/1216769
https://s3.amazonaws.com/metro-extracts.mapzen.com/miami_florida.osm.bz2

Problems Encountered in the Map Data

Upon downloading the OSM file, I ran it through the audit_street_names function from Lesson 6. I also wrote a new python function called audit_zips that scanned all zip codes for non-5- or 9-digit zip codes. The results showed three major issues with the address data:

1. Non-standard street names (e.g., St. vs. Street vs. street)
2. Non-standard direction names (e.g., Northwest vs. NW vs. N.W.)
3. Erroneous postal codes (e.g., FL-33134 or 3447 or 361-0529)

To address these problems, I wrote code to do all of my data cleanup in Python before loading the data into MongoDB.

Non-Standard Street Names

I enhanced the “mapping” array to include many more possibilities; instead of simply looking for Street, Avenue, or Road, I looked for a much longer list of more obscure street types like Circle, Terrace, and Vista. From a Google search, I found this list of possible street names and their abbreviations and copy-pasted it into Excel and saved it as a CSV. After expanding the spreadsheet to include more possibilities, I wrote the function shown below to dynamically read in the CSV rather than maintaining that list within my code.

```
def generate_standardized_street_mapping(street_type_file):
    mappings = {}
    fieldnames = ["abbreviation", "preferred"]
    with open(street_type_file, "rU") as csvfile:
        reader = csv.DictReader(csvfile, fieldnames=fieldnames)
        for row in reader:
            mappings[row["abbreviation"]] = row["preferred"]
    csvfile.close()
    return mappings

STREET_TYPE_CSV = "standardized_street_types.csv"
st_mapping = generate_standardized_street_mapping(STREET_TYPE_CSV)
```

To the standardized street types file, I added obvious errors shown in the audit data like “Cirlce”, “street”, and “Druve”. I also added many more valid street names to the “expected” array to reduce the number of street names that needed to be processed.

Non-Standard Direction Names

The City of Miami is divided into four quadrants: Northwest, Northeast, Southeast, and Southwest, so it seemed like a good idea to standardize the directional names at the beginning of the street name string. During the audit, I found the following counts:

```
Northwest 64        S.W. 1              North 23            E 11
Northeast 28        NW 19               South 27            W 13
Southwest 53        NE 5                East 15             N. 3
Southeast 16        SW 45               West 26             S. 1
southwest 1         SE 1                N 13                E. 2
N.W. 1              sw 1                S 13
```

Based on this information, I decided to go with the most common naming convention: using the full direction name (e.g., “Northwest”) rather than an abbreviation.

I used a very similar process to that of the street names from Lesson 6. It was so similar that I ended up consolidating and refactoring the code into some common functions for auditing and fixing the data. For example, while I kept the “audit_street_type” function, I also created an “audit_direction” function and put common functionality into “audit_address_part”, shown below:

```
def audit_address_part(regex_obj, types, street_name, expected):
    a = regex_obj.search(street_name)
    if a:
        address_part = a.group().strip()
        if address_part not in expected:
            types[address_part].add(street_name)

def audit_street_type(street_types, street_name):
    audit_address_part(street_type_re, street_types, street_name, expected_st)

def audit_direction(dir_types, street_name):
    audit_address_part(direction_re, dir_types, street_name, expected_dir)
```

Erroneous Postal Codes

Postal codes in the United States have either five digits, or five-plus-four digits. In South Florida, they all should start with “33”. I wrote a function called “audit_zips”, with a helper function “audit_zip” to suss out bad zip codes.

```
zip_re = re.compile(r'^33\d{3}($|(-\d{4})$)', re.IGNORECASE) # Zip looks for
5-digits starting with 33, allows for +4

def audit_zip(bad_zips, zip_code):
    z = zip_re.search(zip_code)
    if not z:
        bad_zips.add(zip_code)

def audit_zips(osmfile):
    osm_file = open(osmfile, "r")
    bad_zips = 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_zip(tag):
                    audit_zip(bad_zips, tag.attrib['v'])
    osm_file.close()
    return bad_zips
```

For the invalid results I found, I either had to strip out a leading state abbreviation (“FL”) or look up the property address by a manual Google search (since there were less than 10) to get the right value for the “mapping” array that my newly-created “update_zip” function uses. The original “bad” zip codes and their resulting cleaned up zips are shown below:

```
 33314  => 33314                           FL 33312 => 33312
 FL 33487-3536 => 33487-3536               FL-33140 => 33140
```

```
FL 33431 => 33431                       FL 33134 => 33134
FL 33433 => 33433                       FL 33016 => 33016
FL 33026 => 33026                       3331 => 33313
361-0529 => 33431                       FL 33012 => 33012
FL 33033 => 33033                       11890 => 33181
FL 33126 => 33126                       FL 33140 => 33140
Fl 33186 => 33186                       FL33401 => 33401
FL  33351 => 33351                      (561) 795-4333 => 33411
FL 33431-4403 => 33431-4403             FL 33322 => 33322
FL => 33185                             FL-33139 => 33139
FL 33166 => 33166                       3447 => 33407
0 => 33326                              FL 33131 => 33131
```

To clean the data in Python, I moved the enhanced “update_street_name” and “update_zip” to the part of the shape element function that assigns values to the address fields. I also took the opportunity to ensure that all addresses use “FL” as the state. The resulting code looks like this:

```
if k_array[0] == "addr":
      if len(k_array) == 2:
      if is_street_name(tag):
            val = update_street_name(tag.attrib["v"], st_mapping, dir_mapping)
            elif is_zip(tag):
                  val = update_zip(tag.attrib["v"], zip_mapping)
            elif is_state(tag):
                  val = "FL"
            else:
                  val = tag.attrib["v"]
            address_info[k_array[1]] = val
```

To see more detail, all of the code from Lesson 6 that I enhanced to include direction name and zip code auditing/cleaning is included in this project submission.

Data Overview

File Sizes

Size of zipped OSM file (miami_florida.osm.bz2): 23.7 MB

Size of unzipped OSM file (miami_florida.osm): 353.9 MB

Size of JSON file (miami_florida.osm.json): 390.4 MB

Unique Users

```
pipeline = [{ "$group" : { "_id": "$created.user"}},
            { "$group" : { "_id":1, "count": {"$sum" : 1}}}]
agg = db.miami.aggregate(pipeline)
for a in agg:
    print "Number of unique users: " + "{:,}".format(a["count"])
```

Number of unique users: 1,014

Top 10 Users by Number of Entries (with Percent of Overall Total)

```
ct = db.miami.count() # Total count of collection
pipeline = [{ "$group" : { "_id": "$created.user",
                  "count" : { "$sum" : 1 }}},
            { "$sort" : { "count" : -1 }},
            { "$limit" : 10 }]
agg = db.miami.aggregate(pipeline)
for a in agg:
    pct = float(a["count"])/ct
    print a["_id"] + ": " + "{:,}".format(a["count"]) + ", " +
"{:.1%}".format(pct) + " of total"
```

```
grouper: 299,770, 17.9% of total
woodpeck_fixbot: 236,866, 14.2% of total
Latze: 137,610, 8.2% of total
freebeer: 78,721, 4.7% of total
carciofo: 72,054, 4.3% of total
bot-mode: 62,272, 3.7% of total
NE2: 59,798, 3.6% of total
westendguy: 49,731, 3.0% of total
Seandebasti: 48,397, 2.9% of total
georafa: 39,831, 2.4% of total
```

The entries in this map appear to be reasonably well distributed across a large number of users, with just two handling more than 10% of the total map collection.

Number of Nodes and Ways

```
nodes = db.miami.find({"type":"node"}).count()
print "Number of nodes: " + "{:,}".format(nodes)
ways = db.miami.find({"type":"way"}).count()
print "Number of ways: " + "{:,}".format(ways)
```

```
Number of nodes: 1,477,183
Number of ways: 194,130
```

Top 10 Leisure Categories

```
pipeline = [{"$group" : {"_id" : "$leisure",
                        "count" : { "$sum" : 1 }}},
            {"$sort" : { "count" : -1}},
            {"$limit" : 10}]
agg = db.miami.aggregate(pipeline)
for a in agg:
    print a["_id"], a["count"]
```

```
None 1668086
pitch 1901
park 1008
swimming_pool 703
golf_course 103
playground 95
```

```
sports_centre 73
stadium 56
marina 41
track 39
```

You can tell from the data that South Florida residents like to swim, golf and play outside.

Named Locations in Coral Gables (My City)

```
cg = db.miami.find({"address.city":"Coral Gables"})
for c in cg:
    if "name" in c:
        pprint.pprint(c["name"])
```

```
u'University of Miami Police Department'
u'Miami Spine & Posture Clinic'
u'Fritz & Franz Bierhaus'
u"Swensen's"
u'Law Office of Ferdie and Lones Chartered'
u'Biltmore Golf Course  ,'
u'Granada Golf Course  ,'
u'Village of Merrick Park'
u'Otto G. Richter Library'
u'Holiday Inn Coral Gables'
u'7-Eleven'
u'Coral Gables Art Cinema'
u'Books and Books Bookstore/Cafe'
…. (+ more that I chose not to list)
```

I am happy to see some of my favorite local places appear in the list above, including the Biltmore Golf Course, Fritz & Franz Bierhaus, and Books and Books Bookstore/Cafe.

Boundaries of OSM Download Area

Since I downloaded the file, I wanted to know the boundaries for how far it extends North-South and East-West. That's how I created the image shown at the top of this document.

```
pipeline = [{"$unwind" : "$pos" },
            {"$group" : {"_id" : "$_id",
                        "lat" : { "$first" : "$pos" }}},
            {"$project" : {"_id":0, "lat":1}},
            {"$group" : {"_id" : "$_id",
                        "minLat" : { "$min" : "$lat"},
                        "maxLat" : { "$max" : "$lat"}}}]
agg = db.miami.aggregate(pipeline)
for a in agg:
    print "Min Latitude: " + str(a["minLat"])
    print "Max Latitude: " + str(a["maxLat"])
pipeline = [{"$unwind" : "$pos" },
            {"$group" : {"_id" : "$_id",
                         "lon" : { "$last" : "$pos" }}},
            {"$project" : {"_id":0, "lon":1}},
            {"$group" : {"_id" : "$_id",
                         "minLon" : { "$min" : "$lon"},
                         "maxLon" : { "$max" : "$lon"}}}]
agg = db.miami.aggregate(pipeline)
for a in agg:
    print "Min Longitude: " + str(a["minLon"])
    print "Max Longitude: " + str(a["maxLon"])
```

```
Min Latitude: 25.2910022
Max Latitude: 26.9119981
Min Longitude: -80.682989
Max Longitude: -79.8064041
```

Additional Idea

I queried the MongoDB collection to count the number of entries by city, and got surprising results:

```
pipeline = [{"$group" : {"_id" : "$address.city",
                         "count" : { "$sum" : 1 }}},
            {"$sort" : { "count" : -1}},
            {"$limit" : 10 }]
agg = db.miami.aggregate(pipeline)
for a in agg:
    print a["_id"], a["count"]
```

```
None 1652418
Weston 18251
Miami 336
Fort Lauderdale 188
Miami Beach 137
Wellington 102
Royal Palm Beach 77
Boca Raton 68
Pelican Lake 57
West Palm Beach 53
```

Though Miami and Fort Lauderdale are the largest cities in South Florida, the relatively small community of Weston had by far the most entries in the collection. Probing further, I queried the users to find out who was inputting such a high volume of data for Weston.

```
pipeline = [{"$match" : { "address.city" : "Weston"}},
            {"$group" : { "_id" : "$created.user",
                         "count" : { "$sum" : 1 }}},
            {"$sort" : { "count" : -1}}
agg = db.miami.aggregate(pipeline)
for a in agg:
    print a["_id"], a["count"]
```

```
dataabierta 18237
ernestocd 8
Auction123 1
bladdiaz 1
adjuva 1
MountainAddict 1
grouper 1
mentor 1
```

The results show that just one user, "dataabierta"[1] is responsible for almost all of the Weston's entries. Another query shows that none of dataabierta's entries are named sites:

```
cg = db.miami.find({"address.city":"Weston"})
for c in cg:
    if "name" in c:
        print c["name"], c["created"]["user"]
```

```
Alejandro's by Alejandro (Hair Salon) mentor
Fix Apple Now grouper
165 bladdiaz
Ultimate Software ernestocd
Ultimate Software ernestocd
Auction123.com Auction123
Falcon Cove Middle School ernestocd
Everglades Elementary School ernestocd
Sawgrass Recreation Park adjuva
```

In looking further at the data, it appears that dataabierta supplied the specific street address for every property in Weston, probably through some automated, "bot" approach

It struck me that the map would be more complete if the rest of the cities in South Florida had that level of detail. To improve the overall data set, it would probably be useful to apply dataabierta's methodology to the other cities and towns. However, it's possible that Weston's street address location data is more readily available than other localities, making the methodology difficult to apply to other places.

[1] "dataabierta" is Spanish for "open data"