Data Wrangle OpenStreetMaps Data

Laurent de Vito

July 27, 2015

We choose the OSM of Avignon, France, from https://mapzen.com/data/metro-extracts. Proceeding as in the *Data Wrangling with MongoDB* course, we audited and cleaned the data in python before importing it into MongoDB using

```
mongoimport  -d osm -c Avignon --file avignon_france.osm.json
```

In MongoDB, we ran some queries against the data.

1 Problems encountered in the map

We (pretty-) printed a few nodes with tags from the dataset to get a feeling about how the data is formatted before programming various checks in python (build upon what we learned in the course). Obviously, it is not as nicely structured as in the course, and the way data was typed in differs significantly from user to user.

1.1 Addresses

We would expect addresses to be formatted as, e.g.:

```
{'changeset': '24962415',
 'id': '303080419',
 'lat': '43.9493122',
 'lon': '4.8116326',
 'timestamp': '2014-08-23T18:45:25Z',
 'uid': '2277250',
 'user': 'Philippe Dumas',
 'version': '2'}
{'k': 'name', 'v': 'Code Bar'}
{'k': 'phone', 'v': '04 88 07 83 31'}
{'k': 'amenity', 'v': 'pub'}
{'k': 'addr:city', 'v': 'Avignon'}
{'k': 'addr:street', 'v': 'Rue du Portail Matheron'}
{'k': 'addr:postcode', 'v': '84000'}
{'k': 'addr:housenumber', 'v': '2'}
```

However, we ran into a node that has the peculiarity that what pertains to the address is not preceded by "addr:", as in:

```
{'changeset': '15360704',
 'id': '25214701',
 'lat': '43.927',
 'lon': '4.842',33
 'timestamp': '2013-03-14T10:15:50Z',
 'uid': '11068',
 'user': 'windu2b',
 'version': '4'}
{'k': 'name', 'v': u"C C ''Cap-Sud'' G\xe9ant"}
{'k': 'is_in', 'v': 'AVIGNON'}
{'k': 'source', 'v': 'stations.gpl.online.fr'}
{'k': 'address', 'v': '162, Avenue P Semard'}
{'k': 'amenity', 'v': 'fuel'}
{'k': 'fuel:lpg', 'v': 'yes'}
{'k': 'operator', 'v': 'Casino'}
{'k': 'addr:postcode', 'v': '84000'}
```

This example is interesting: The key "is_in" is a substitute for "addr:city", and the key "address" is the concatenation of "addr:housenumber" and "addr:street". We can recast the latter in the form we expect using regular expressions. However, in this case, it would be an overkill to do it programmatically since it turns out that it is the only example of this kind in the OSM dataset: Normally, the key "address" does not include any house number.

Is "address" used as a bad substitute for "addr:street" ? We found out that the above is also an exception. The key "address" is used in place of "addr:street" when the street type is not common, e.g. think of an amenity in a commercial center for instance.

For the treatment of the key "is_in", see Section 1.3.2.

Sometimes, the information is stripped to its bare minimum, as in:

```
{'changeset': '22127250',
 'id': '303080428',
 'lat': '43.9495545',
 'lon': '4.8119055',
 'timestamp': '2014-05-04T15:01:37Z',
 'uid': '1068293',
 'user': 'x0c0',
 'version': '4'}
{'k': 'name', 'v': 'Histoire de pains'}
{'k': 'shop', 'v': 'bakery'}
{'k': 'addr:country', 'v': 'FR'}
```

which is also superfluous. More problematic are nodes that only have house numbers, as in:

```
{'changeset': '9723876',
 'id': '1489711867',
 'lat': '43.9375933',
 'lon': '4.8043063',
 'timestamp': '2011-11-02T17:20:56Z',
 'uid': '333347',
```

```
 'user': 'gaston',
 'version': '1'}
{'k': 'source',
 'v': u'cadastre-dgi-fr source : Direction G\xe9n\xe9rale des Imp\xf4ts - Cadastre. Mise \xe0 jo
{'k': 'addr:housenumber', 'v': '2'}
```

This is in fact logical since this node might be referenced by a way that will indicate the street name, so there is no need for duplicates that are usually another source of errors.

More problematic for the analysis later, is that some ways defining streets do not refer to any city. More on this in Section 2.1.

1.2 Harmonizing street types

We found the following counts for different street types:

```
{u'All\xe9e ': 335,
 'Av ': 0,
 'Av. ': 0,
 'Ave. ': 0,
 'Avenue ': 591,
 'Boulevard ': 217,
 'Ch. ': 4,
 'Chemin ': 542,
 'Cour ': 12,
 'Impasse ': 393,
 'Place ': 126,
 'Route ': 188,
 'Rue ': 1663,
 'Square ': 8,
 'Voie ': 15,
 u'all\xe9e ': 1,
 'av ': 0,
 'avenue ': 1,
 'boulevard ': 1,
 'ch ': 0,
 'ch. ': 0,
 'chemin ': 7,
 'cour ': 0,
 'impasse ': 0,
 'others': 4070,
 'place ': 0,
 'route ': 0,
 'rue ': 7,
 'square ': 0,
 'voie ': 0}
```

As we can see, all street types start with a capital letter, e.g. "Rue" (1663 counts), except for a few ones, e.g. "rue" (7 counts). Harmonizing this is straightforward by simply adapting what was done in the *Data Wrangling with MongoDB* course.

1.3 Checking postal codes

1.3.1 Finding the cities behind postal codes

It is not uncommon that datasets have erroneous postal codes, so we first checked postal codes from tags with keys containing the string "postcode" (it is usually either "postcode" or "addr:postcode").

Obviously a postal code must be a 5-digit integer. Apart from that, it is possible to get the name of the city associated to a postal code. If none is found, the postal code is not valid. We initially made queries against the web site http://www.codeposte.com/ because of its simplicity. This is best explained using a plain example. Suppose we wish to get the name of the city with postal code 84000 (it is the city of Avignon). It suffices to retrieve the html page http://www.codeposte.com/home.php?s_keyword=84000 (the postal code is sent in the URL's query string) and to parse it:

```
def is_postcode_valid(code):
    query_string = { 's_keyword': code }
    r = requests.get("http://www.codeposte.com/home.php?", params=query_string)
    if r.text.find("Aucune commune") != -1: # ~ no city
        return False
    return True
```

However, the web site did not recognize the postal code 84140, issuing that no city had such a postal code. But it is the postal code of the city of Montfavet. To circumvent this problem, we searched for a better web site (that would offer the same kind of service). We found http://www.code-postal.com/search.php[1]. Using the following code[2]:

```
def get_city_name(code):
    r = requests.post("http://www.code-postal.com/search.php",
                      data={'mots_cles': code,
                            'Submit': 'Chercher'
                      })
    soup = BeautifulSoup(r.text)
    td_texts = []
    for td in soup.find_all('td'):
      if td.string != None:
        td_text = td.string.strip()
        if td_text.startswith("Aucune ville"):  # "Aucune ville" = "no city" [found]
          return None
        td_texts.append(td_text)
    if code in td_texts:
        pos = td_texts.index(code)
        city = td_texts[pos+1]
        return city
    else:
        return None
```

[1] This web site is specialized in postal codes from France. Since Avignon and its surroundings are fairly away from other countries, we are confident that any postal code found in the dataset for Avignon pertains to a city in France.

[2] Notice that this snippet is not robust against connections that time out. Furthermore, we ought to check the status code of the request as in

```
    if r.status_code != requests.codes.ok:
        print "bad request"
```

we can easily retrieve the name of the city associated to a given postal code. The advantage of proceeding so is twofold: We further check for the validity of a postal code and we can cross-check the answer against the name of the city specified in the address.

We were able to get a list of very few suspicious postcodes, namely postcodes that are not assigned to any city. We cross-checked some of them using yet another web site: https://www.laposte.fr/particulier/outils/trouver-un-code-postal. It turned out that those postcodes are weird but valid. So, the final course of action was to query against this web site. Despite our efforts, we were not successful at getting results, though the request status was ok, using the following request:

```
r = requests.post("https://www.laposte.fr/particulier/outils/trouver-un-code-postal",
                  data={'VilleCP[filtre]': "ville",
                        'VilleCP[communeCode]': code,
                        "VilleCP[save]": "",
                        "VilleCP[ezxform_token]": "0519c4d10ad5dc69e2a4c7a0564519dc902dfcf9",
                        'Submit': 'Submit'
                  })
```

The failure was caused by the hidden parameter "VilleCP[ezxform_token]" that changed at each new browser session but was identical for a given browser session no matter how many manual queries we did. To mimic what the browser experienced, we created a session, got the main page to extract the token, and eventually made the post request:

```
def get_city_name(code):
    s = requests.Session()

    # getting the token

    r = s.get("https://www.laposte.fr/particulier/outils/trouver-un-code-postal")
    soup = BeautifulSoup(r.text)
    token_element = soup.find(id="VilleCP_ezxform_token")
    token_value = token_element['value']

    # making the requests

    r = s.post("https://www.laposte.fr/particulier/outils/trouver-un-code-postal",
                    data={'VilleCP[filtre]': "ville",
                          'VilleCP[communeCode]': code,
                          "VilleCP[save]": "",
                          "VilleCP[ezxform_token]": token_value,
                          'Submit': 'Submit'
                    })

    if r.text.find(u"0␣R sultat␣pour␣votre␣recherche") != -1:
        return None

    soup = BeautifulSoup(r.text)
    fd_code = False
    for td in soup.find_all('td'):
        if td.string == code:
          fd_code = True
        elif fd_code:
          city_names = []
          for s in td.strings:
              city_names.append(s)
          break;

    city_name,_,_ = city_names[0].partition("␣CEDEX")
```

```
    return city_name
```

The last few lines of the above snippet deserve some comments. For some postcodes, more than a single city was returned. Consider for instance the postcode 84140. The answer to a query with this postcode has the following content:

```
<td>MONTFAVET<br>AVIGNON</td>
```

It is ambiguous, and we presume that the first city is the right one. So we build a list and return the first element. Moreover, some answers to a query are composed of the city name followed by "CEDEX XX" where XX is a number. The python method partition allows us to easily extract the city name from such answers.

All postal codes found in the dataset were valid: They all pertained to a city.

1.3.2 Cross-checking postal codes with cities

By cross-checking the postal code with the city (if found among the tags), we often observed that some primitives (either node or way) mention a postcode ("addr:postcode" available) but no city ("addr:city" unavailable). So, at first sight, we could easily complete the address. However, the cities found from the postal codes are not unicode strings but in plain ASCII coded: All french accents are lost.

How to cope with this issue ? From http://sql.sh/736-base-donnees-villes-francaises, we downloaded a csv-formatted file (without header) listing most french cities, spelled with and without accents! So, we can augment the incomplete address by the (unicoded) city only if we can find the city in the city database[3].

We found a single conflict in the OSM dataset between postal code and city. Care must be exercised: The database of french cities is neither complete nor error-free. For instance, our code reported a false conflict: The city (derived from the postal code) was "Les Angles", but it was erroneously introduced in the city database as "Angles". Furthermore, the city of Montfavet (actually, sort of neighbourhood of Avignon, with its own postal code) was missing.

The presence of a postcode without mention of a city is valid for a node that specifies a city through the key='name'. In this case, re-casting all the information pertaining to the address ("addr:postcode", "addr:city", ...) as a dictionary under "address" is a bit awkward.

We had few primitives that mention a city ("addr:city" available) but no postcode ("addr:postcode" unavailable). Correction could easily be carried out.

For some cities mentioned as value of the key k="is_in" instead of the traditional k="addr:city", the city was incorrectly spelled, e.g. "LE-PONTET" instead of "Le Pontet". Those tags can contribute to complete the address if the city is not specified. In fact, if the key k="is_in" is available, we noted that there were no key k="addr:city" or k="city". Hence those tags with keys k="is_in" could contribute to complete the address if the city they specified can be found in the city database.

[3] In fact, the city names in the csv file were not in utf-8, but e.g. in ISO-8859-1. Since we couldn't find the exact encoding, the encoding in unicode was difficult because the city names must first be decoded. Fortunately, there is an alternative to the module csv: The unicodecsv module is a drop-in replacement for Python 2's csv module which supports unicode strings without a hassle. We worked with unicodecsv as we would have done with csv and we didn't have to worry about string encoding.

1.4 Checking names

A few values of key = "name" have a white space after the character "" as in e.g.

```
{'changeset': '136537',
 'id': '302848805',
 'lat': '43.9507194',
 'lon': '4.8057221',
 'timestamp': '2008-10-07T14:43:05Z',
 'uid': '3114',
 'user': 'chippy',
 'version': '1'}
{'k': 'name', 'v': u"L' Echapp\xe9 Belle"}
{'k': 'amenity', 'v': 'restaurant'}
```

This is not common and we presume that it reflects how the name is depicted on the amenity's front door for instance. However, someone searching for this name would never include that white space, and so the search would fail (we exclude that the search engine would use a smart regular expression). This is easily detected using the following pattern

```
apostrophe_type_re = re.compile(r"^[ld]'␣+\S|␣[ld]'␣+\S", re.IGNORECASE)
```

For a given tag then:

```
if (key == "name") or (key == "address") or (key == "addr:street"):
    m = apostrophe_type_re.search(val)
    if m:
        s = str(m.group())
        new_s = s.replace("␣","")
        if s[0] == "␣":
            new_s = "␣" + new_s
        old_val = val
        val = val.replace(s,new_s)
        print "'%s'␣has␣white␣space␣after␣apostrophe␣->␣replaced␣by␣'%s'" % (old_val,val)
```

2 Overview of the Data

2.1 Basic facts

The size of the uncompressed OSM file was 146.9 Mo, comprising 653175 nodes

```
query = { "type" : "node"}
```

and 114990 ways

```
query = { "type" : "way"}
```

But only 622 ways own a reference to a city.

```
query = { "type" : "way", "address.city": { "$exists" : 1 } }
```

or

```
pipeline = [ { "$match" : { "type" : "way" } },
             { "$match" : { "address.city" : { "$exists" : 1 } } } ]
```

and as few as 87 nodes. But as a way refers to many nodes, we can retrieve all the ways pertaining to a city and count their (unique) nodes. Doing so, we get the total number of nodes pertaining to a city. Using the following pipeline:

```
pipeline = [ { "$match" : { "type" : "way" } },
             { "$match" : { "address.city" : { "$exists" : 1 } } },
             { "$unwind" : "$node_refs" },
             { "$group" : { "_id" : "$node_refs",
                            "count" : { "$sum" : 1 } } },
             { "$sort" : { "count" : -1 } }
             ]
```

from

```
result = aggregate(db, pipeline)
print len(result["result"])
```

we got: 3640. We also deduced that a node can be found in at most 4 ways by simpling printing the first result:

```
{u'_id': u'635076251', u'count': 4}
```

using

```
pprint.pprint( result["result"][0] )
```

2.2 Users

Using

```
pipeline = [ { "$group" : { "_id" : "$created.user",
                            "count" : { "$sum" : 1 } } },
             { "$sort" : { "count" : -1 } }
           ]
```

we got an idea about the top 10 users:

```
{u'_id': u'krysst', u'count': 433268}
{u'_id': u'Luc Alquier', u'count': 189170}
{u'_id': u'J-Louis ZIMMERMANN', u'count': 42919}
{u'_id': u'gaston', u'count': 15585}
{u'_id': u'Herv\xe9 TUC', u'count': 14922}
{u'_id': u'botdidier2020', u'count': 13671}
{u'_id': u'Patchi', u'count': 6469}
{u'_id': u'st1974', u'count': 5596}
{u'_id': u'Gall', u'count': 3850}
```

making up a total of 729095 contributions! Obviously, they have found a way to do it programmatically. The top user cites its source for the ways:

```
u'source': u'cadastre-dgi-fr source : Direction G\xe9n\xe9rale des Imp\xf4ts - Cadastre
```

Apparently, his data comes from an on-line version of a land registry.

2.3 Restaurants

There are 221 restaurants all cities confounded in the database

```
query = { "amenity" : "restaurant"}
```

There are 5 restaurants in Avignon

```
query = { "amenity" : "restaurant", "address.city": "Avignon" }
```

This is quite surprising. In fact, most restaurants are mentioned without any reference to a city. The following

```
query = { "amenity" : "restaurant", "address.city": { "\$exists" : 0 } }
```

returns 216.

3 Other ideas about the datasets

3.1 Limits of cities

It is possible to plot all nodes (as points for instance) that are part of a given city, and for each city assign a color for its nodes, so that the extend of each city is clearly rendered.

Using the following:

```
pipeline = [ { "$match" : { "type" : "way" } },
             { "$match" : { "address.city" : { "$exists" : 1 } } },
             { "$unwind" : "$node_refs" },
             { "$group" : { "_id" : "$node_refs",
                            "count" : { "$sum" : 1 },
                            "cities" : { "$addToSet" : "$address.city" } } },
             { "$sort" : { "count" : -1 } }
             ]
```

we got the nodes and the city they are registered in[4]. In a first pass, we gathered all cities:

```
set([u'Avignon',
     u'Le Pontet',
     u'Montfavet',
     u'Mori\xe8res-l\xe8s-Avignon',
     u'Rochefort-du-Gard',
     u'Sorgues',
     u'Ved\xe8ne',
     u'Villeneuve-l\xe8s-Avignon'])
```

and in a second pass, we plotted the nodes:

[4] We also checked that a node belongs to a single city.

```
#!/usr/bin/env python

import pprint
import matplotlib
matplotlib.use('Qt4Agg')
import matplotlib.pyplot as plt

def get_db(db_name):
    from pymongo import MongoClient
    client = MongoClient('localhost:27017')
    db = client[db_name]
    return db

def make_query(node_id):
    query = { "type" : "node", "id": node_id }
    return query

def query(db,q):
    node_ref = db.Avignon.find_one(q)
    return node_ref

def make_pipeline():
    pipeline = [ { "$match" : { "type" : "way" } },
                 { "$match" : { "address.city" : { "$exists" : 1 } } },
                 { "$unwind" : "$node_refs" },
                 { "$group" : { "_id" : "$node_refs",
                                "count" : { "$sum" : 1 },
                                "cities" : { "$addToSet" : "$address.city" } } },
                 { "$sort" : { "count" : -1 } }
                 ]

    return pipeline

def aggregate(db, pipeline):
    result = db.Avignon.aggregate(pipeline)
    return result

if __name__ == '__main__':
    db = get_db('osm')
    pipeline = make_pipeline()
    result = aggregate(db, pipeline)

    cities = set()
    for item in result["result"]:
        cities.add(item["cities"][0])

    colors = [ 'pink', 'red', 'green', 'blue', 'black', 'yellow', 'cyan', 'magenta' ]

    c = 0
```

```
for city in cities:
    x = []
    y = []
    for node in result["result"]:
if node["cities"][0] == city:
            node_ref = query(db,make_query(node["_id"]))
            x.append( node_ref["pos"][0] )
            y.append( node_ref["pos"][1] )
    plt.scatter(x,y,c=colors[c],s=60,label=city,alpha=1., edgecolors='none')
    c += 1

plt.tick_params(axis='both',which='both',bottom='off',top='off',
                labelbottom='off',labelleft='off',right='off',left='off')
plt.grid(True)
plt.legend()
plt.show()
```

The result is depicted in Figure 1. This is not exactly what we were hoping: There is not clear-cut limits for each city because we have too few nodes referencing a city.

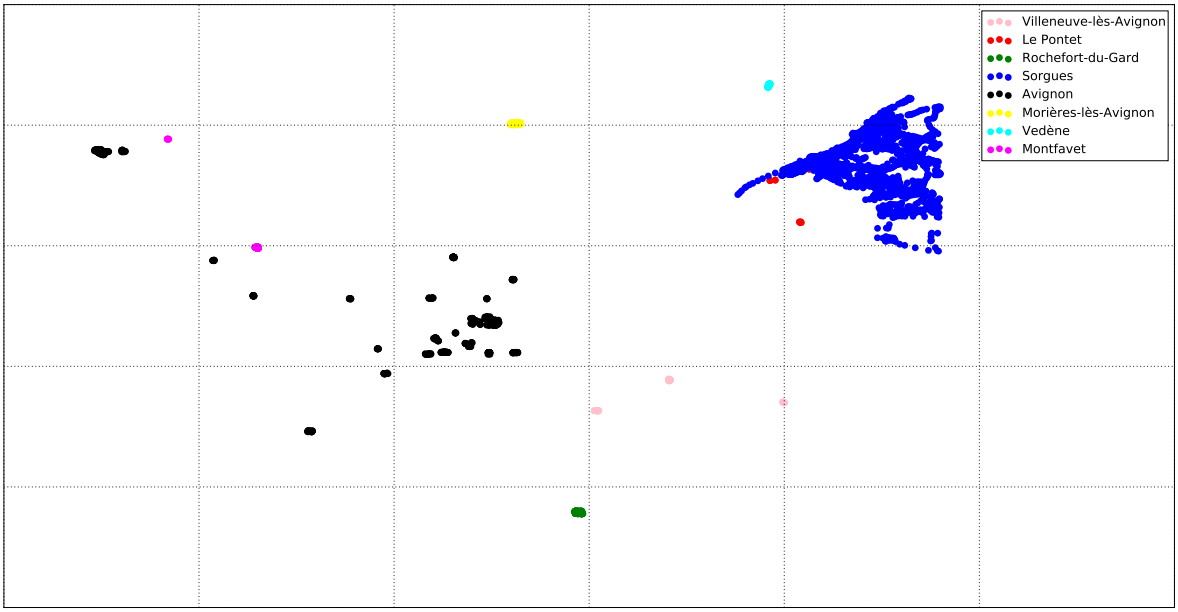

Figure 1: Nodes pertaining to a city in the Avignon OSM dataset.

In the top left corner, we have the city of Montfavet, a neighbourhood of Avignon, with its own postal code. We notice that nodes in Montfavet are either referenced as pertaining to Montfavet or to Avignon (black points close to the small set of pink points).