P3_Diogo_Cosin.md

OpenStreetMap Case Study

Map Area

São Paulo, SP, Brazil

I have chosen São Paulo because it is my hometown and I would like to know some statics and facts about it.

The XML OSM file was extracted from MapZen. São Paulo OpenStreetMap can be find here.

Data Wrangling

Street Type Audit

In Brazil, addresses respect the pattern:

[Street Type] [Street Name], [Street Number]

For example, one of the most famous avenues in São Paulo, Paulista Avenue, in portuguese is written:

Avenida Paulista, where "Avenida" is the street type meaning Avenue.

So in order to detect the address street type in the XML file data, it was used the following regular expression written in Python.

```
street_type_re = re.compile(r'^\S+\.?', re.IGNORECASE)
```

The regex above detects the first word in address street, because it must be the street type. Also, only the first letter should be capitalized.

The following function was used to match the street type in a street address tag. Moreover, this function will also return a dictionary containing all street types detect during the auditing process and the number of occurrences for it.

```
def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        street_types[street_type] += 1
```

After executed the street type auditing process, we can obtain the following street types dictionary:

```
1ª: 1
Acesso: 3
Acost.: 1
Al.: 16
Alameda: 334
Alfonso: 1
Angelo: 1
Antonio: 1
antonio: 1
armando: 1
Av: 4
av.: 2
Av.: 19
avenida: 1
Avenida: 7006
Azevedo: 1
Barroco: 1
```

```
Bueno: 2
Cajaíba: 1
Calçadão: 4
Campinho: 1
Cantídio: 1
Castro: 1
Complexo: 1
Conselheiro: 4
Coronel: 3
Corredor: 1
Doutor: 4
Dr.: 1
Escapatória: 1
Espírito: 1
Estrada: 274
estrada: 1
Fernando: 1
Garcia: 2
Guarara: 1
Hermínio: 1
Ignacio: 1
JOSÉ: 1
José: 1
Ladeira: 1
Largo: 51
lucidalva: 1
Luiz: 1
Manoel: 1
Marginal: 3
Maria: 1
Marina: 2
Marquês: 1
Nívia: 1
O: 1
Oito: 1
Oscar: 1
Parati: 1
Parque: 9
Passagem: 5
Pateo: 1
Paulo: 1
Pires: 1
Praça: 229
praça: 1
presidente: 1
R: 2
R.: 10
r.: 1
Rocha: 1
Rodoanel: 2
Rodovia: 142
Rua: 14735
RUa: 1
rua: 14
RUA: 1
Rue: 1
sao: 3
Tavares: 1
Travessa: 96
Tupinambás: 1
Vale: 1
VERMELHO: 1
Via: 15
Viaduto: 4
Viela: 5
Álvares: 1
Ângela: 1
```

Many street types returned are not adherent to the standard for different reasons:

- Over abbreviated. Examples: R. Av.
- Incorrect capitalization. Examples: RUA, RUa, avenida.
- Logically invalid street type. Examples: Rocha, Pires and Parati are not street types, but only compound the street name, meaning that some streets do not present any street type.

Street Type Cleaning

The over abbreviated and incorrect capitalization were corrected using a mapping dictionary, as shown bellow.

```
mapping = { "Al.": "Alameda",
            "Av.": "Avenida",
            "Av": "Avenida",
            "avenida": "Avenida",
            "av.": "Avenida",
            "estrada": "Estrada",
            "praça": "Praça",
            "R": "Rua",
            "R.": "Rua",
            "r.": "Rua",
            "RUa": "Rua",
            "RUA": "Rua",
            "rua": "Rua",
            "Rue": "Rua"}
```

The remaining street addresses with no street types were discarded once it is not possible to assert what is the correct street type.

Postcode Audit

In Brazil, it's used a eight-digit form for postcodes. Soon, the standardized postcode must have the following regex pattern:

```
postcode_re = re.compile(r'\d{5}-\d{3}', re.IGNORECASE)
```

Running a postcode auditing process, we detected some inconsistent postcodes.
Many postcodes do not present any kind of separator between the first 5 digits and the last 3 digits. Others separates the digits using space instead of hyphen, and others even use dots. Bellow you will find some inconsistent pattern postcodes examples.

```
"02634020" # no separator.
"12.216-540" # using dot.
"06016 008" # space as separator.
```

Besides the inconsistent pattern cases, some postcodes presented invalid length. Bellow you will find some examples.

```
"3221-390" # 7-digits postcode.
"077780-000" # 9-digits postocode.
```

Postcode Cleaning

The inconsistent postcode pattern cases were corrected through a string reshaping process and the invalid length postcodes were discarded given that it is not possible to programmatically detect what is the correct postcode in a simple way.

The following function was used to programmatically update the inconsistent postcodes before recording it on MongoDB.

```
def upadate_postcode(postcode):
    postcode_correct_re = re.compile(r'\d+', re.IGNORECASE)  #regex to match only numeric characters in postcode.
    m = postcode_correct_re.findall(postcode)
    postcode_updated = ''.join(m)[:5] + "-" + ''.join(m)[5:]
    if len(postcode_updated) == 9:
        return postcode_updated
    else:
        print "Invalid postcode lenght {}. Postcode {} will be discarded." .format(len(postcode_updated), postcode)
        return None
```

Data facts

MongoDB Document

After auditing and cleaning processes, data was reshaped in a JSON format and recorded in a MongoDB database. Bellow you will find the schema used for the documents. If a tag does not contain any content for a specific field, this field is not inserted in the MongoDB document.


```
{
"id": "2406124091",
"type: "node",
"visible":"true",
"created": {
          "version":"2",
          "changeset":"17206049",
          "timestamp":"2013-08-03T16:43:42Z",
          "user":"linuxUser16",
          "uid":"1219059"
        },
"pos": [41.9757030, -87.6921867],
"address": {
          "housenumber": "5157",
          "postcode": "60625",
          "street": "North Lincoln Ave"
        },
"amenity": "restaurant",
"cuisine": "mexican",
"name": "La Cabana De Don Luis",
"phone": "1 (773)-271-5176"
}
```


Volumetry

XML Data File size: 762.9 MB

Some MongoDB Queries

Using MongoDB shell, let's make some queries in our database.

Number of Nodes

```
> db.node.find({type: "node"}).count()

3375664
```

Number of Ways

```
> db.node.find({type: "way"}).count()

468711
```

Number of Unique Users

```
> db.node.distinct("created.user").length

2077
```

Most Common Streets

Here we can partially assure that the street type cleaning was correct if this query won't return any inconsistent street type.

```
> db.node.aggregate([
...                       { $match: { "address.street": {$exists: true}} },
```

```
...                         { $group: { _id: "$address.street", count: { $sum: 1} } },
...                         { $sort: { count: -1 } },
...                         { $limit: 15 }
...                 ])

{ "_id" : "Avenida Conceição", "count" : 526 }
{ "_id" : "Rua Oriente", "count" : 243 }
{ "_id" : "Rua Xavantes", "count" : 203 }
{ "_id" : "Avenida Paulista", "count" : 197 }
{ "_id" : "Rua Maria Marcolina", "count" : 182 }
{ "_id" : "Avenida Brigadeiro Luís Antônio", "count" : 168 }
{ "_id" : "Rua Silva Teles", "count" : 162 }
{ "_id" : "Avenida Corifeu de Azevedo Marques", "count" : 162 }
{ "_id" : "Rua Catumbi", "count" : 158 }
{ "_id" : "Rua Voluntários da Pátria", "count" : 156 }
{ "_id" : "Rua Agostinho Gomes", "count" : 141 }
{ "_id" : "Avenida Guilherme", "count" : 140 }
{ "_id" : "Rua Miller", "count" : 139 }
{ "_id" : "Avenida Sapopemba", "count" : 138 }
{ "_id" : "Rua Reverendo Israel Vieira Ferreira", "count" : 138}
```

Most Common Amenities

```
> db.node.aggregate([
...                     { $match: { amenity: {$exists: true}} },
...                     { $group: { _id: "$amenity", count: { $sum: 1} } },
...                     { $sort: { count: -1 } },
...                     { $limit: 15 }
...                 ])

{ "_id" : "parking", "count" : 2939 }
{ "_id" : "fuel", "count" : 1775 }
{ "_id" : "restaurant", "count" : 1264 }
{ "_id" : "school", "count" : 1207 }
{ "_id" : "bank", "count" : 929 }
{ "_id" : "fast_food", "count" : 758 }
{ "_id" : "place_of_worship", "count" : 688 }
{ "_id" : "pharmacy", "count" : 470 }
{ "_id" : "pub", "count" : 382 }
{ "_id" : "hospital", "count" : 319 }
{ "_id" : "bar", "count" : 292 }
{ "_id" : "bicycle_rental", "count" : 275 }
{ "_id" : "telephone", "count" : 245 }
{ "_id" : "police", "count" : 231 }
{ "_id" : "bench", "count" : 203 }
```

Most Common Cuisines

```
> db.node.aggregate([
...                     { $match: { cuisine: {$exists: true}} },
...                     { $group: { _id: "$cuisine", count: { $sum: 1} } },
...                     { $sort: { count: -1 } },
...                     { $limit: 15 }
...                 ])

{ "_id" : "regional", "count" : 255 }
{ "_id" : "burger", "count" : 160 }
{ "_id" : "pizza", "count" : 129 }
{ "_id" : "fish_and_chips", "count" : 75 }
{ "_id" : "japanese", "count" : 71 }
{ "_id" : "sandwich", "count" : 41 }
{ "_id" : "italian", "count" : 33 }
{ "_id" : "coffee_shop", "count" : 25 }
{ "_id" : "chinese", "count" : 13 }
{ "_id" : "arab", "count" : 12 }
{ "_id" : "international", "count" : 12 }
{ "_id" : "brazilian", "count" : 11 }
{ "_id" : "steak_house", "count" : 10 }
{ "_id" : "ice_cream", "count" : 9 }
{ "_id" : "sushi", "count" : 8 }
```

Surprise here. I didn't imagine Fish and Chips was so popular in Sao Paulo.

Additional Ideas

Postcode and Position Data Cross-validation

Same postcodes should have nearby locations. So, a programmatically cross-validation between these two fields should detects inconsistent postcode and position values. For instance, the MongoDB database was queried in order to find all nodes with the same specific postcode.

```
> db.node.find({"address.postcode": "01311-300"}, {"_id": 0, "address.postcode": 1, "pos": 1})

{ "pos" : [ -23.5563104, -46.6619509 ], "address" : { "postcode" : "01311-300" } }
{ "pos" : [ -23.5585041, -46.6600857 ], "address" : { "postcode" : "01311-300" } }
{ "pos" : [ -23.559279, -46.6592171 ], "address" : { "postcode" : "01311-300" } }
{ "pos" : [ -23.5584401, -46.6601503 ], "address" : { "postcode" : "01311-300" } }
```

In this case, the position returned are consistent given that they describe nearby locations.

Reputation Points for Users

Some sites already use this kind of mechanics. For example, Stackoverflow's users must have minimum reputation points in order to edit posts. Using the same idea, the data audit process could reward users that have inserted good quality data. This way, users with good reputation should have more priority in editing OpenStreetMap tags.

Mapping Street Types using Machine Learning

For the street type data audit and update process, the mapping dictionary was filled totally with human interference. This may mean a potential problem as new invalid street types may still be inserted by users. In this case, the dictionary will be not adapted for new inconsistent values, demanding a human edition.
As a solution, it might be possible to implement a unsupervised machine learning algorithm able to detect new inconsistent street types and programmatically relate it to the corrects ones through the mapping dictionary.

Conclusion

In this project, a XML OSM file has been audited and transformed in order to be inserted in a MongoDB database. Many street types and postcodes were cleaned efficiently and programmatically, however it was necessary great human interference in defining the expected values and patterns for street type and postcodes. This can harm the created solution as new inconsistent values may still be inserted by OpenStreetMaps's users, meaning that the audit process should be adjusted in order to deal with the new inconsistent values. A machine learning might be used in this case.