project_openSTreetMap_MongoDb

December 24, 2017

0.1 OpenStreetMap Project Data Wrangling with MongoDB

Author: Nicolas Luiz Ribeiro Veiga
github: nicolaslrveiga
Linkedin: Nicolas Veiga

0.1.1 Objective

The goal of this document is to apply Data Wrangling tecniques learned on the course(Data Analyst Nanodegree, module 5) over data from OpenStreetMap.
First we are going to select an region of the globe, than we are going to analyse its data, do some cleaning and report all problems found. The last step is to convert this data into JSON format and import it to MongoDB from there we are able to properly query the data.

0.1.2 Introduction

São Paulo is a municipality in the southeast region of Brazil. The metropolis is an alpha global city and the most populous city in Brazil and Americas as well as in the Southern Hemisphere. The municipality is also the largest in the Americas and Earth's 12th largest city proper by population. The city is the capital of the surrounding state of São Paulo, one of 26 constituent states of the republic. It is the most populous and wealthiest city in Brazil. It exerts strong international influences in commerce, finance, arts and entertainment. The name of the city honors the Apostle, Saint Paul of Tarsus. The city's metropolitan area of Greater São Paulo ranks as the most populous in Brazil, the 11th most populous on Earth, and largest Portuguese language-speaking city in the world.

Having the largest economy by GDP in Latin America and the Southern Hemisphere, the city is home to the São Paulo Stock Exchange. Paulista Avenue is the economic core of São Paulo. The city has the 11th largest GDP in the world, representing alone 10.7% of all Brazilian GDP and 36% of the production of goods and services in the state of São Paulo, being home to 63% of established multinationals in Brazil, and has been responsible for 28% of the national scientific production in 2005. With a GDP of US$477 billions, the Sao Paulo city alone could be ranked 24th globally compared with countries. (2016 Estimates).

The metropolis is also home to several of the tallest skyscraper buildings in Brazil, including the Mirante do Vale, Edifício Itália, Banespa, North Tower and many others. The city has cultural, economic and political influence both nationally and internationally. It is home to monuments, parks and museums such as the Latin American Memorial, the Ibirapuera Park, Museum of Ipiranga, São Paulo Museum of Art, and the Museum of the Portuguese Language. The city holds events like the São Paulo Jazz Festival, São Paulo Art Biennial, the Brazilian Grand Prix, São Paulo

Fashion Week and the ATP Brasil Open. The São Paulo Gay Pride Parade rivals the New York City Pride March as the largest gay pride parade in the world. It is headquarters of the Brazilian television networks Band, Gazeta and RecordTV.

São Paulo is a cosmopolitan, melting pot city, home to the largest Arab, Italian, and Japanese diasporas, with examples including ethnic neighborhoods of Mercado, Bixiga, and Liberdade respectively. São Paulo is also home to the largest Jewish population in the country and one of the largest urban Jewish populations in the world. In 2016, inhabitants of the city were native to 196 different countries. People from the city are known as paulistanos, while paulistas designates anyone from the state, including the paulistanos. The city is also known for the size of its helicopter fleet, its architecture, gastronomy, severe traffic congestion and skyscrapers. Wiki

Furthermore the city is spectacular, so to prove all its glamour we can answer some questions drawed from introduction. The questions we are going to answer are: - How tall are the 5 tallest buildings? - With kind of amenity is most common? - With kind of cousine is predominant? - With avenue pass by a large number of subburbs? - How many places to land my helicopter? :) - I am a big fan of coffe places, how many coffe places I have im my disposal?

This document is divided as follows:

- Dataset:
  - Indicates the dataset used.
  - Link to the dataset.
- Data Exploration:
  - Basics statistics about the data, like its lenght.
  - Statistics about the tags in the file.
  - Statistics about the users.
- Data Cleaning:
  - Audicting the tags that are usefull for us to answer the questions above.
  - Problems encontered during audiction.
  - How the problems are going to be fixed.
- Data Wrangling:
  - Convert the data to JSON.
  - Fix the problems reported in Data Cleaning stage.
  - Adds json to MongoDB database.
  - Basic statistics about the database.
- Drawing conclusions:
  - Query the database answering the questions above.
- Additional Ideas
  - Ideas from the process and how it is possible to improve the data.

0.1.3 Dataset

Region choosed: São Paulo, Brazil

```
In [1]: DATASET_PATH = "sao-paulo_brazil.osm"
```

0.1.4 Data Exploration

Getting file size:

```
In [2]: import os

        print("File size im mega bytes:")
        print(os.stat(DATASET_PATH).st_size / 1000000.0)
```

```
File size im mega bytes:
1081.490417
```

Because of the size of the document, initially we worked with a scaled down version of the dataset, only 50 mb. Eventually when all the important features had been audicted, we started processing all the dataset.

```
In [3]: """
        This block of code was originally provided so we can reduce the original dataset. The
        the number of k-th elements that we are going to have in the sampled dataset. k = 100
        a dataset with 10mb of size, this size is suitable for test and debuging.
        """

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

        #set the path to our entire dataset to OSM_FILE and the sample file to SAMPLE_FILE.
        OSM_FILE = DATASET_PATH  # Replace this with your osm file
        SAMPLE_FILE = "sample.osm"

        #Set this variable to the dataset that you want to process.
        DATASET = DATASET_PATH

        k = 100 #arameter: 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:
```

```
        #OSM HEADER, each file contains a header and a footer that identifies the type of
        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'))

        #OSM FOOTER
        output.write('</osm>')
```

Counting and getting all tags in the dataset, this data is going to give a notion of how many primitives we are dealing with and can be further compared with the data that is going to be imported to the database.

```
In [4]: import pprint
        import xml.etree.ElementTree as ET

        def count_tags(filepath):
            """
            This method iterates through all the dataset and counts each time a tag appears.
            input: Dataset path
            output: Dictionary were key is a tag and value is frequency of that tag.
            """
            tags = {}
            tree = ET.parse(filepath)
            root = tree.iter()
            for child in root:
                if child.tag in tags:
                    tags[child.tag] += 1
                else:
                    tags[child.tag] = 1
            return tags

        tags = count_tags(DATASET)
        pprint.pprint(tags)
```

```
{'bounds': 1,
 'member': 80672,
 'nd': 6467712,
 'node': 4763397,
 'osm': 1,
 'relation': 13278,
 'tag': 2157283,
 'way': 649248}
```

Data tags definitions

- node: Represents a point in space, it contains an id, latitude and longitude. Most commonly used to delimit a way. It can also contain tag with key and value pair.

- way : is a ordered list of nodes, normally has at least one tag. Represents ways of going from one place to another. tags nd are reference to node.

- relation: is one of the core data elements that consists of one or more tags and also an ordered list of one or more nodes, ways and/or relations as members which is used to define logical or geographic relationships between other elements.
- member: node or way that belongs to a relation.

- nd : reference to node

- osm : top level tag
- bounds : region bounds
- tag: A tag consists of two items, a key and a value. Tags describe specific features of map elements (nodes, ways, or relations) or changesets. Both items are free format text fields, but often represent numeric or other structured items.

From the tags description, we can see that all features about a node, way or relation are inside tag "tag". So we need to take a look in which features we have access and which feature can help us to answer the questions above.

```
In [6]: def get_keys():
            """
            Each map element(nodes, ways or relations) present their attributes in tags, each
            a value that belongs to that key. This method is going to iterate throught all the
            each map element.
            """
            tags_way = set()
            tags_node = set()
            tags_relation = set()
            for event, elem in ET.iterparse(DATASET, events = ("start",)):
                if elem.tag == "way":
                    for tag in elem.iter("tag"):
                        tags_way.add(tag.attrib['k'])
                if elem.tag == "node":
                    for tag in elem.iter("tag"):
                        tags_node.add(tag.attrib['k'])
                if elem.tag == "relation":
                    for tag in elem.iter("tag"):
                        tags_relation.add(tag.attrib['k'])
                elem.clear()
            #pprint.pprint(tags_way)
            #pprint.pprint(tags_node)
            #pprint.pprint(tags_relation)
            print("Number of features in way:")
            print(len(tags_way))
            print("Number of features in node:")
            print(len(tags_node))
            print("Number of features in relation:")
            print(len(tags_relation))

        get_keys()
```

```
Number of features in way:
684
Number of features in node:
584
Number of features in relation:
294
```

The output was suppressed because there are a lot of different types, instead of each key I am only showing the number of keys each element has. That is a lot of features. Since only a small portion of this features are used constantly to describe a primitive, only listing the tags does not help, we need to count the tags and sort it in away that it is possible to see with features we can take advantage.

Computing how many times each key was used, only showing the top 10:

```
In [7]: def get_count_keys():
            """
            This method counts the amount of times each tag appears and sort it in a decrescen
            output: dictionary where the key is a tag attribute and value is the frequency of
            """
            keys_counter = {}
            for event, elem in ET.iterparse(DATASET, events = ("start",)):
                if elem.tag == "way" or elem.tag == "node" or elem.tag == "relation":
                    for tag in elem.iter("tag"):
                        if tag.attrib['k'] in keys_counter:
                            keys_counter[tag.attrib['k']] += 1
                        else:
                            keys_counter[tag.attrib['k']] = 1
                elem.clear()
            return keys_counter

        #Sort a dictionary by value in decrescent order and print the top 10
        keys_counter = get_count_keys()
        sorter_keys_counter = sorted(keys_counter, key = keys_counter.get, reverse = True)
        for index, key in enumerate(sorter_keys_counter):
            print("%s: %s" % (key, keys_counter[key]))
            if index == 10:
                break

building: 361659
source: 341858
height: 333240
highway: 260072
name: 182318
surface: 87429
oneway: 68367
addr:street: 41876
addr:city: 39255
addr:housenumber: 38031
addr:postcode: 31785
```

oks, now we can pick the features that can help us to answer the question stated in introduction. The features that we are going to use are: - street: Gives the address - name: Name of the place - height: Height of the building - cousine: type of cousine - amenity: type of community facilite - suburb: In a city we can have multiple place with the same adress, suburb subdivides a city. - aeroway: The aeroway key is used to tag physical infrastructure used to support aircraft, air travel, spacecraft and space flight, in particular the elements associated with airports, spaceports and heliports etc. - building: The building key is used to mark areas as a building. - city - postcode

For more information about tag go to: http://wiki.openstreetmap.org/wiki/

Some of this data need cleaning and others no. Features like street, name, height, suburb, city and postcode are open features where the user is free to enter what he things is suitable, because of that a lot of human errors can happen here. Now features like cousine, amenity, aeroway and building already have categories that the user can choose, it is also prune to erros but the chance of having fields with wrong spelling or weird caracters is low.

Now, computing the number of users that we are dealing with, if we have a lot of user that contributed equally the chances ar that the data is going to have a lot of inconsistences.

```
In [8]: def users(filename):
            """
            This method gets all attributes equal to user and count the frequency each user ap
            input: File path
            output: Dictionary where keys are users and value is the amount of times each user
            """
            users = {}
            for _, element in ET.iterparse(filename):
                if  "user" in element.attrib:
                    if element.attrib['user'] in users:
                        users[element.attrib['user']] += 1
                    else:
                        users[element.attrib['user']] = 1
                element.clear()
            return users
```

```
#Sort a dictionary by value in decrescent order and print the top 10
data_users = users(DATASET)
sorter_data_users = sorted(data_users, key = data_users.get, reverse = True)
for index, key in enumerate(sorter_data_users):
    print("%s: %s" % (key, data_users[key]))
    if index == 10:
        break
```

```
Bonix-Mapper: 2802473
AjBelnuovo: 393998
Bonix-Importer: 342773
cxs: 168544
O Fim: 115462
johnmogi: 98863
MCPicoli: 94870
ygorre: 89844
naoliv: 80755
patodiez: 74544
Roberto Costa: 64703
```

Some statistics about user data:

```
In [9]: print("Percentage of data from Bonix-Mapper:")
        print(float(data_users[sorter_data_users[0]])/float(sum(data_users.values()))*100.0)

        print("Percentage of data from the top 5 contributer:")
        print(float(sum(data_users[user] for user in sorter_data_users[:5]))/float(sum(data_use
```

```
Percentage of data from Bonix-Mapper:
51.6497008896
Percentage of data from the top 5 contributer:
70.4626659833
```

From this data we can see that 51% of the data comes from one user, this data probably can be trusted and follows a certain pattern. Also 70% of the data comes from the top 5 contributers, it may contain a little inconsistency beetween than.

0.1.5 Data Cleaning

As stated before, the fields that needs audiction are: - street - height - suburb - city - postcode - name

Audicting street name Audict addr:street from way tag Standartization: All strings are going to be converted to ascii and all letters to lower case, easier way to deal with accents and upper case letters.

```
In [11]: import re
         import collections
         import sys
         import unidecode

         def audict_addr_street():
             """
             This method audicts addr:street, it has a dictionary that contains the expected t
             these types of street are compared against the first word of addr:street, if any
             address is normal, otherwise it has to be fixed.
             output: Streets that needs to be fixed.
             """
             street_first_word_re = re.compile(r"^[a-zA-Z]*", re.UNICODE)
             street_types = collections.defaultdict(set)
             expected_street_types = ["rua", "avenida", "alameda", "via", "estrada", "rodovia"
                                      "passagem", "marginal", "parque", "acesso", "vila", "ponte

             for event, elem in ET.iterparse(DATASET, events = ("start",)):
                 if  elem.tag == "node" or elem.tag == "way" or elem.tag == "relation":
                     for tag in elem.iter("tag"):
                         if (tag.attrib['k'] == "addr:street"):
                             #print(tag.attrib['v'])
                             m = street_first_word_re.findall(unidecode.unidecode(tag.attrib['
                             if m[0]:
                                 m[0] = m[0].lower()
                                 if m[0] not in expected_street_types:
                                     street_types[m[0]].add(unidecode.unidecode(tag.attrib['v']
                     elem.clear()
             return street_types

         audict_addr_street = audict_addr_street()
```

An example of audict_addr_street is:

```
In [16]: audict_addr_street["av"]

Out[16]: {'av cassiano ricardo',
          'av jacu pessego / nova trabalhadores',
          'av sara veloso',
          'av. 9 de abril',
          'av. agenor c. de magalhaes',
          'av. albert bartholome',
          'av. andromeda',
          'av. antonio joaquim de moura andrade',
          'av. arlindo betio',
          'av. augusto zorzi baradel furquim',
          'av. brg. faria lima',
          'av. consolacao, 1290',
          'av. das nacoes unidas',
          'av. francisco nobrega barbosa',
          'av. horacio lafer',
          'av. liberdade',
          'av. marginal',
          'av. mathias lopes',
          'av. yojiro takaoka'}
```

Initially, for the SAMPLE part of the data, only two problems were reported:
- "Av." is apreviation for "Avenida", that is going to be fixed programmaticaly. - "JOSÉ PEREIRA CRUZ", searching for it on google maps we see that is classified as "Rua". This problem is going to be fixed manually, but if there are too many similar cases in the entire dataset, that is going to became a huge time consumption task.

When audicting all the dataset, new problems emerged: - new street types that were not included in the street_tyles list like, marginal, parque, acesso. - soma street types with typos, like: alamedas, rodoanel and rue. - A lot of differences because of accentuation and lower or upper case letters. - Some streets without a type. For all the street types that can be found in Brazil take a look at:
http://coopus.com.br/site/upload/tabelas-ans/lodragouro-tipo.pdf

Solution encontered for the problems: - All new street types that were not in the list and are present in the document lodragouro-tipo.pdf where add to the list. - Street types with typos are going to be ignored. - All characteres are going to be converted to ASCII. - Streets without a type are going to be ignored.

```
In [17]: street_name_problem = collections.defaultdict(set)
         street_name_problem['avenida'].add("av.")
         street_name_problem['alamedas'].add("al.")
         street_name_problem['rua'].add("r.")
         street_name_problem['rodovia'].add("rod.")
         street_name_problem['estrada'].add("estr.")

         def fix_street(street_name, street_name_problem):
             """
             This method receives a street name, removes upper letter and converts each caract
             input: street_name that is the street that is going to be evaluated.
                    street_name_problem is a dictionaty with type of words that needs to be fi
             """
             street_name = unidecode.unidecode(street_name).lower()
             street_type = street_name.split(" ")[0].encode("utf-8")
             for correct_street_types in street_name_problem:
                 for wrong_street_type in street_name_problem[correct_street_types]:
                     if (street_type == wrong_street_type):
                         street_name = street_name.replace(street_type, correct_street_types)
             return street_name
```

Audicting name

```
In [19]: def audict_name():
             """
```

```
                This method audicts name, this variable contains names of streets and places. We 
                audict for street because we already did it for addr:street. Regarding name of pl
                to remove accents and upper case letters.
                output: Dictionary with the frequency each time a name of a place appears in the 
                """
                names = {}
                for event, elem in ET.iterparse(DATASET, events = ("start",)):
                    if  elem.tag == "node" or elem.tag == "way" or elem.tag == "relation":
                        for tag in elem.iter("tag"):
                            if (tag.attrib['k'] == "name"):
                                name = unidecode.unidecode(tag.attrib['v']).lower()
                                if name in names:
                                    names[name] += 1
                                else:
                                    names[name] = 1

                    elem.clear()
                return names

            #limiting the output of audict_name to only show the top 10 with the same name.
            names = audict_name()
            names = sorted(names, key = names.get, reverse = True)
            for index, name in enumerate(names):
                print(name)
                if index == 10:
                    break
```

```
rodoanel mario covas
rua sem denominacao
rodovia raposo tavares
rodovia presidente dutra
avenida corifeu de azevedo marques
rodovia anchieta
itau
bradesco
santander
rua um
banco do brasil
```

The output of audict name was limited because it is a very long list, the ouput contains the top 10 names that appears more in the dataset. Name has the same properties as street, so the same code can be used to fix it.

Audicting height

```
In [21]: def audict_height():
                """
```

```
        This method audicts height, it searchs for number that fit a pattern. The pattern
        a string that only has numbers and one "." to separete from decimal part. Any str
        fit in the pattern is going to be considered a height that needs to be fixed.
        output: Array containing two dictionaries, the first contains the heights that fi
        the second the heights that does not fit in the pattern.
        """
        heights = []
        heights_with_problems = []
        isnumber = re.compile(r"[+-]?\d+(?:\.\d+)?", re.IGNORECASE)

        for event, elem in ET.iterparse(DATASET, events = ("start",)):
            if elem.tag == "way" or elem.tag == "node" or elem.tag == "relation":
                for tag in elem.iter("tag"):
                    if (tag.attrib['k'] == "height"):
                        m = isnumber.findall(tag.attrib['v'])
                        if m:
                            if(m[0] == tag.attrib['v']):
                                heights.append(tag.attrib['v'])
                            else:
                                heights_with_problems.append(tag.attrib['v'])
            elem.clear()
        return [heights, heights_with_problems]

    #Print the heights with problem.
    #pprint.pprint(audict_height()[1])
```

The output of audict height was supressed, the results where summarized below.

Height: The default unit is meters. If the height is measured in a different unit, the unit abbreviation is appended to the value, separated by a space. initially we only had two problem with height: - Values with unit. - Values that are not in meters.

When audicting all the dataset - Measures with "," as separator and others with ".". - Building with multiple parts.

Solution for the problems listed above: - Remove any num numerical character - Keep all measures in meters - Convert "," to "." - Convert the data to a list of floats.

```
In [22]: def fix_height(height_from_data):
             """
             This method receives a string that represents a height and fix test it agains the
             the audiction fase, if the problem is present we fix it.
             input: String that represents a height.
             output: Array of heights in float, it is an array because a building can have mul
             """
             heights = []
             height_from_data = height_from_data.split(";")
             for height in height_from_data:
                 height = height.replace(" ","").replace(",",".").replace("m","")
                 if "'" in height:
                     height = float(height.replace("'",""))
```

```
                height = height * 0.3048
            else:
                height = float(height)
            heights.append(height)
        return heights
```

Audicting city

```
In [23]: def audict_city():
             """
             This method audicts city, it basically counts each occurency of city. That is use
             in the data. We also remove upper case letters and convert the strings to utf-8
             output: Dictionary where keys are the cities and values is the amount of times ea
             """
             cities = {}

             for event, elem in ET.iterparse(DATASET, events = ("start",)):
                 if elem.tag == "way" or elem.tag == "node" or elem.tag == "relation":
                     for tag in elem.iter("tag"):
                         if (tag.attrib['k'] == "addr:city"):
                             city = unidecode.unidecode(tag.attrib['v']).lower()
                             if city in cities:
                                 cities[city] += 1
                             else:
                                 cities[city] = 1
                     elem.clear()
             return cities

         #To run audict_city, run the code below.
         #pprint.pprint(audict_city())
```

Problems encontered for cities: - Lowercase vs uppercases - utf-8 to ascii - A lot of misspeling words and wrong names, this data can not be fixed programmatically - sao paulol - 2 inputs with postal code instead of city name, to fix it the postal code is going to be replaced by the city name. - 06097-100 - same cities with state initials on the name, to fix it everything after "-" and "," is going to be ignored. - every caracter that is not a letter is going to be removed - campo0 limpo paulista - city name with "sp" are going to be removed without regular expression. - varzea paulista, sp

```
In [24]: dict_postal_code_to_city = {}
         dict_postal_code_to_city["06097-100"] = "osasco"
         dict_postal_code_to_city["08589-000"] = "itaquaquecetuba"

         def fix_city(city_name, dict_postal_code_to_city):
             """
             This method fix city based on the problems found in the audiction. It removes unw
             write spaces. It also converts CEP to city names.
             input: city_name that contains the city name.
                    dict_postal_code_to_city it is a dictionary that contains CEP and its resp
             """
```

```
        city_name = unidecode.unidecode(city_name).lower()
        city_name = city_name.replace("-sp","")
        city_name = city_name.replace(", sp","")
        city_name = city_name.replace(" - sp","")
        if city_name in dict_postal_code_to_city:
            city_name = dict_postal_code_to_city[city_name]
            return city_name
        copy_city_name = city_name
        for char in city_name:
            if not char.isalpha() and char != " ":
                copy_city_name = copy_city_name.replace(char, "")
        return copy_city_name
```

Audicting postcode

```
In [26]: def audict_postcode():
        """
        This method audicts postcode. It compares agains a pattern that represents a post
        XXXXX-XXX. If postal code does not fits the pattern, it is printed.
        output: all postal codes that follows the pattern.
        """
        postcode = {}
        postal_code_re = re.compile(r'(\d{5})([ ])?([-])?([ ])?(\d{3})', re.IGNORECASE)

        for event, elem in ET.iterparse(DATASET, events = ("start",)):
            if elem.tag == "way" or elem.tag == "node" or elem.tag == "relation":
                for tag in elem.iter("tag"):
                    if (tag.attrib['k'] == 'addr:postcode'):
                        m = postal_code_re.search(tag.attrib['v'])
                        if m:
                            m = m.groups()
                            postal_code = m[0] + "-" + m[4]
                            if postal_code in postcode:
                                postcode[postal_code] += 1
                            else:
                                postcode[postal_code] = 1
                        else:
                            print(tag.attrib['v'])
            elem.clear()
        return postcode

    #To run audict_postcode, run the code below.
    #audict_postcode = audict_postcode()
```

The postal code that does not match the regular expressions are going to be dropped. Problems with postcode: - some has hifen others no - 12.216-540 - 061 5000 - some have white spaces - "." instead of "-" - misplacement of the separator - wrong number of digits - in this case the data is going to be ignored.

All the problem reported were because the string did not match the standart for postal code: https://www.correios.com.br/para-voce/precisa-de-ajuda/o-que-e-cep-e-por-que-usa-lo/estrutura-do-cep

```
In [27]: def fix_postal_code(postal_code):
             """
             This method receives a postal code and return it in the pattern XXXXX-XXX. If the
             postal code is not 9, it returns None, invalidating the data.
             input: String containing the postal code.
             output: String following the postal code pattern.
             """
             postal_code_re = re.compile(r'(\d{5})([ ])?([-])?([ ])?(\d{3})', re.IGNORECASE)
             m = postal_code_re.search(postal_code)
             if m:
                 m = m.groups()
                 postal_code = m[0] + "-" + m[4]
                 return postal_code
             else:
                 copy_postal_code = postal_code
                 for char in postal_code:
                     if not char.isdigit():
                         copy_postal_code = copy_postal_code.replace(char, "")
                 copy_postal_code = copy_postal_code[:5] + "_" + copy_postal_code[5:]
                 if len(copy_postal_code) != 9:
                     return None
                 else:
                     return copy_postal_code
```

Audicting suburb

```
In [29]: def audict_suburb():
             """
             This method audicts suburb. It iterates thought all the addr:suburb and counts al
             of each suburb, it also converts all upper letters to lower letters and converts
             goal is to visualize all suburb so we can isolate the problems that we need to fi
             output: Dictionary where key is suburb and values is how often each key appeared.
             """
             suburbs = {}

             for event, elem in ET.iterparse(DATASET, events = ("start",)):
                 if elem.tag == "way" or elem.tag == "node" or elem.tag == "relation":
                     for tag in elem.iter("tag"):
                         if (tag.attrib['k'] == "addr:suburb"):
                             suburb = unidecode.unidecode(tag.attrib['v']).lower()
                             if suburb in suburbs:
                                 suburbs[suburb] += 1
                             else:
                                 suburbs[suburb] = 1
```

```
            elem.clear()
        return suburbs

    #To run audict_suburb, run the code below.
    #pprint.pprint(audict_suburb())
```

Problems founded: - Some abreviation like parque to pq. - vl da saude - Inconsistency because of accents and upper case letters. - Miss spelling words. - bras cubas and braz cubas - One case of a postal code in the data. - 02924-000

Solutions: - Convert the abreviations. - Convert all the characters to ASCII and lower case. - Searching on google the suburb that corresponds to the postal code is Vila Arcádia. Since it is only one entry, the data is going to be fixed programatically.

```
In [30]: suburb_problem = collections.defaultdict(set)
         suburb_problem['parque'].add("pq.")
         suburb_problem['jardim'].add("jd.")
         suburb_problem['vila'].add("vl")

         dict_postal_code_to_city = {}
         dict_postal_code_to_city[ "02924-000"] = "vila arcadia"

         def fix_suburb(suburb, suburb_problem):
             """
             This method fix suburb. It receives a string and compares if is a postal code, if
             converts it to its respective surburb name. It also test to see if the type of th
             abreviated, if it is abreviated we convert it to its full name.
             input: suburb is a string containing the name of the suburb.
                    suburb_problem is a dictionary that contains abreviation of suburb types.
             output: String containing the suburb name fixed.
             """
             if suburb in dict_postal_code_to_city:
                 return dict_postal_code_to_city[suburb]
             suburb = unidecode.unidecode(suburb).lower()
             suburb_type = suburb.split(" ")[0].encode("utf-8")
             for correct_suburb_types in suburb_problem:
                 for wrong_suburb_type in suburb_problem[correct_suburb_types]:
                     if (suburb_type == wrong_suburb_type):
                         suburb = suburb.replace(suburb_type, correct_suburb_types)
             return suburb
```

0.1.6 Data Wrangling

```
In [31]: """
         The task here is to wrangle the data and change its format to JSON so it can be inser
         For example, for primitives like nodes the output should be like:

         {
         "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"
         }

         Features like address have a class of keys like adreess:street, for address only the 

         {...
         "address": {
             "housenumber": 5158,
             "street": "North Lincoln Avenue"
         }
         "amenity": "pharmacy",
         ...
         }

         - for "way" specifically:

           <nd ref="305896090"/>
           <nd ref="1719825889"/>

         is going to be turned into
         "node_refs": ["305896090", "1719825889"]

         - for "relations"specifically:
           <member ref="188909336" role="outer" type="way" />
           <member ref="42846055" role="outer" type="way" />
           <member ref="42845680" role="outer" type="way" />
           <member ref="42847596" role="outer" type="way" />

           the output is going to be a list of dictionaries like:
           "member": [{"ref":"188909336, "role":"outer" "type":"way"},{...}...]
         """
```

```python
from exceptions import TypeError, ValueError

def audict_tag(key, value):
    """
    This function receives a tag that is the key and a value that belongs to that key
    the tag is one of our attributes of interrest that are: height, street, name, cit
    suburb. If a tag is a tag of interrest, the value is going to be fixed by the fun
    the audiction fase.
    input: key that is an attribute.
           value that represents the attribute.
    output: value fixed.
    """
    if key == "height":
        value = fix_height(value)
    if key == "street":
        value = fix_street(value, street_name_problem)
    if key == "name":
        value = fix_street(value, street_name_problem)
    if key == "city":
        value = fix_city(value, dict_postal_code_to_city)
    if key == "postcode":
        value = fix_postal_code(value)
    if key == "suburb":
        value = fix_suburb(value, suburb_problem)
    return value

def add_tag_to_dict(dictionary, key, value):
    """
    This method adds a tag to a dictionary. If the key presents "." in the name, it i
    removed because mongoDb does not accept dictionary keys with dots in the name.
    input: dictionary tag is going to receive the new attribute and its value
           key to be add to the dictionary
           value that represents the key.
    """
    value = audict_tag(key, value)
    dictionary[key.replace(".","")] = value

def process_addr_key(tag, dictionary):
    """
    Method that receives a tag that belongs to the addr class of attributes and proce
    The method is going to remove the "addr:" parte of the attribute name and send th
    to the dict.
    input: tag that contains an attribute key and its value
           dictionary that is going to receive the new tag.
    """
    add_tag_to_dict(dictionary, tag.attrib['k'].split(":")[1], tag.attrib['v'])

def process_tag(tag, dictionary):
    """
    This method  receives a tag and checks to see if it belongs to the "addr" class,
    otherwise it sends it to add_tag_to_dict directly.
    input: tag with key and value
           dictionary that the tag is going to be inserted.
    """
    if (":" in tag.attrib['k']):
        if (tag.attrib['k'].split(":")[0] == "addr"):
            if "addr" not in dictionary:
                dictionary["addr"] = {}
            process_addr_key(tag, dictionary["addr"])
        else:
            add_tag_to_dict(dictionary, tag.attrib['k'].replace(":", "_"), tag.attrib
    else:
        if tag.attrib['k'] != "addr":
            add_tag_to_dict(dictionary, tag.attrib['k'], tag.attrib['v'])

def wrangle_node(elem):
    """
    Wrangle node receives a element that is a node, creates a dictionaty and process
    input: node element
    output: dictionary with all the node items and tags in it.
    """
    node_dict = {}
    node_dict["primitive"] = "node"

    for name, value in elem.items():
        node_dict[name] = value

    for tag in elem.iter("tag"):
        process_tag(tag, node_dict)
    return node_dict

def wrangle_way(elem):
    """
    Wrangle way receives a element that is a node, creates a dictionaty and process a
    input: way element
    output: dictionary with all the way items, nd and tags in it.
    """
    way_dict = {}
    way_dict["primitive"] = "way"

    for name, value in elem.items():
        way_dict[name] = value

    for nd in elem.iter("nd"):
        for name, value in nd.items():
            if name in way_dict:
                way_dict[name].append(value)
            else:
                way_dict[name] = []
                way_dict[name].append(value)

    for tag in elem.iter("tag"):
        process_tag(tag, way_dict)
    return way_dict

def wrangle_relation(elem):
    """
    Wrangle relation receives a element that is a relation, creates a dictionaty and 
    relation.
    input: relation element
    output: dictionary with all the relation items, members and tags in it.
    """
    relation_dict = {}
    relation_dict["primitive"] = "relative"

    for name, value in elem.items():
        relation_dict[name] = value

    member_list = []
    for nd in elem.iter("member"):
        member_dict = {}
        for name, value in nd.items():
            member_dict[name] = value
        member_list.append(member_dict.copy())
    if len(member_list) != 0:
        relation_dict["member"] = member_list

    for tag in elem.iter("tag"):
        process_tag(tag, relation_dict)
    return relation_dict

def wrangle(elem):
    """
    This methods receives a element, checks to see if it is a way, node or relation a
    processed.
    input: element
    output: Dictionary with the element attributs in it.
    """
    if elem.tag == "way":
        json = wrangle_way(elem)
        return json
    if elem.tag == "node":
        json = wrangle_node(elem)
        return json
    if elem.tag == "relation":
        json = wrangle_relation(elem)
        return json
    return None

def insert_json(json, db):
    """
    Insert json to database
    """
    db.open_street_map.insert_one(json)

def query_open_street_map_dataset(db):
    print("Number of inputs:")
    print(db.open_street_map.find().count())

    print("Number of inputs that are nodes:")
    print(db.open_street_map.find({"primitive":"node"}).count())

    print("Number of inputs that are ways:")
    print(db.open_street_map.find({"primitive":"way"}).count())

    print("Number of inputs that are relations:")
    print(db.open_street_map.find({"primitive":"relative"}).count())


if __name__ == "__main__":
    # Connecting to local database, if there is a database with the same name drops a
    from pymongo import MongoClient
    client = MongoClient("mongodb://localhost:27017")
    client.drop_database("open_street_map")
    db = client.open_street_map

    #Process every element of the dataset
    for event, elem in ET.iterparse(DATASET, events = ("start",)):
        json = {}
        json = wrangle(elem);
        try:
            if json is not None:
                insert_json(json, db)
        except:
            print(json)
        elem.clear()
    query_open_street_map_dataset(db)
```

```
Number of inputs:
5425923
Number of inputs that are nodes:
4763397
```

```
Number of inputs that are ways:
649248
Number of inputs that are relations:
13278
```

```
In [32]: print db.command("dbstats")
```

```
{u'storageSize': 410177536.0, u'ok': 1.0, u'avgObjSize': 242.3584610397162, u'views': 0, u'db'
```

Done, all the nodes, ways and relations are in a database. Two problems during the process: - there is a feature called addr only, because of this variable the code was crashing because it was trying to store a dict in a variable type string. - MongoDb does not accept keys with ".", there is one feature with "." that was making the code to crash, solution for it was to remove all "." from keys.

All the erros above were cought handling the errors using try except statements.

Some statistics about the MondoDb collection: - It's size is significantlly smaller than the original osm file. * original: 1081.490417 mb * collection: 410148864 bytes, what gives us 410 mb - The number of nodes, ways and relations in the collection are the same as in the osm file.

Data exploration using MondoDb queries Finally we can easily answer the questions states in the beginning. - How tall are the 5 tallest buildings?

```
In [33]: pipeline = [{"$unwind": "$height"},
                      {"$match":{"building":{"$ne":None}, "name":{"$ne":None}}},
                     {"$sort":{"height":-1}},
                     {"$limit":5}]
         query_result = db.open_street_map.aggregate(pipeline)
```

```
In [34]: the_tallest_buildings = []
         for data in query_result:
             the_tallest_buildings.append(data)
```

```
In [35]: for data in the_tallest_buildings:
             print("name: %s, building: %s, height: %d" % (data["name"], data["building"], data
```

```
name: mirante do vale, building: yes, height: 131
name: edificio copan, building: apartments, height: 118
name: grande sao paulo, building: yes, height: 114
name: instituto do cancer do estado de sao paulo octavio frias de oliveira, building: hospital
name: cbi esplanada, building: commercial, height: 100
```

Second the dataset, The building "Mirante do Vale" is the highiest building in São Paulo, with 131 meters. A quick search on google confirms that this is the tallest building, but second the wikipedia, it has 170 meters.wikipedia

- With kind of amenity is most common?

```
In [36]: pipeline = [{"$match":{"amenity":{"$ne":None}}},
                      {"$group":{"_id":"$amenity", "count":{"$sum":1}}},
                      {"$sort":{"count":-1}},
                     {"$limit":5}]
         query_result = db.open_street_map.aggregate(pipeline)
```

```
In [37]: amenities = []
         for data in query_result:
             amenities.append(data)
```

```
In [38]: for data in amenities:
             print(data)
```

```
{u'count': 3694, u'_id': u'parking'}
{u'count': 1867, u'_id': u'fuel'}
{u'count': 1635, u'_id': u'restaurant'}
{u'count': 1553, u'_id': u'school'}
{u'count': 1041, u'_id': u'bank'}
```

The query return that we have 3694 parking lots in São Paulo.

- With kind of cousine is predominant?

```
In [39]: pipeline = [{"$match":{"cuisine":{"$ne":None}}},
                      {"$group":{"_id":"$cuisine", "count":{"$sum":1}}},
                      {"$sort":{"count":-1}},
                     {"$limit":5}]
         query_result = db.open_street_map.aggregate(pipeline)
```

```
In [40]: cousines = []
         for data in query_result:
             cousines.append(data)
```

```
In [41]: for data in cousines:
             print(data)
```

```
{u'count': 306, u'_id': u'regional'}
{u'count': 192, u'_id': u'burger'}
{u'count': 162, u'_id': u'pizza'}
{u'count': 82, u'_id': u'japanese'}
{u'count': 73, u'_id': u'fish_and_chips'}
```

Second the data we have 306 restaurants the offers "regional" food.

- With avenue pass by a large number of subburbs?

```
In [42]: pipeline = [{"$match":{"addr.street":{"$regex":"^(avenida)"}, "addr.suburb":{"$ne":No
                      {"$group":{"_id":{"street":"$addr.street", "suburb":"$addr.suburb"}, "cou
                      {"$group":{"_id":"$_id.street", "count":{"$sum":1}}},
                      {"$sort":{"count":-1}},
                      {"$limit":10}]
         query_result = db.open_street_map.aggregate(pipeline)

In [43]: avenues = []
         for data in query_result:
             avenues.append(data)

In [44]: for data in avenues:
             print(data)
```

```
{u'count': 9, u'_id': u'avenida robert kennedy'}
{u'count': 9, u'_id': u'avenida das nacoes unidas'}
{u'count': 6, u'_id': u'avenida brigadeiro faria lima'}
{u'count': 5, u'_id': u'avenida sao joao'}
{u'count': 5, u'_id': u'avenida anna costa'}
{u'count': 5, u'_id': u'avenida dom pedro i'}
{u'count': 5, u'_id': u'avenida antonio piranga'}
{u'count': 5, u'_id': u'avenida santos dumont'}
{u'count': 5, u'_id': u'avenida moinho fabrini'}
{u'count': 5, u'_id': u'avenida getulio vargas'}
```

Second the dataset, we have two avenues that goes by 9 suburbs each, we have "avenida robert kennedy" and avenida das nacoes unidas. A quick serch on google returns that the avenue "Avenida Sapopemba" goes by 27 suburbs.wikipedia
For this question I used regex to match the first street name as "avenida" than i had to use two group statements to get the answer i wanted.

- How many places to land my helicopter?

```
In [45]: pipeline = [{"$match":{"aeroway":"helipad"}},
                     {"$group":{"_id":"helipad", "count":{"$sum":1}}}]
         query_result = db.open_street_map.aggregate(pipeline)

In [46]: aeroways = []
         for data in query_result:
             aeroways.append(data)

In [47]: for data in aeroways:
             print(data)
```

```
{u'count': 455, u'_id': u'helipad'}
```

There is 455 helipads in São Paulo region.

- I am a big fan of coffe places, how many coffe places I have im my disposal?

```
In [48]: pipeline = [{"$match":{"$or":[{"amenity":"cafe"},{"cuisine":"coffee_shop"}], "addr.st
                     {"$group":{"_id":"coffe_places","count":{"$sum":1}}}]
         query_result = db.open_street_map.aggregate(pipeline)
```

```
In [49]: coffe_places = []
         for data in query_result:
             coffe_places.append(data)
```

```
In [50]: for data in coffe_places:
             print(data)
```

```
{u'count': 120, u'_id': u'coffe_places'}
```

There is 120 coffe places where i can go. To answer this questions I have to use the or operand in match statement because a coffe shop can be defined either on amenity as cafe or in cuisine as coffee_shop. wiki

0.1.7 Conclusion and additional ideas

The purpose of that project was to obtain a dataset, clean it, wrangle it and awnswer some questions with data from the dataset. During the process all problems were documented in this notebook. Some data that were audited were not used because it could restrict even more the results if they were included in the queries to the database, features like city and postal code. That could happen because a lot of data is missing from the dataset. The main conclusion about the dataset is that it is incomplete, some questions answered did not match with results from other sources.

One idea to fix some of the problems is to use the postal code and geolocation. There are some datasets that gives us geolocation, postal code and address used by the Brazilian post office, the benefits of these datasets is that it comes from a goverment agency and are more reliable. Another advantage is that geolocation is linked with an address and postal code, with this information we can fill gaps in the dataset or use it as reference to check how reliable the information is. The downside of using it is that it does not give information like the type of building, its name and other information that we use and are very common in our day to day life. Another problem with this is in the case of new entries that are not present in the post office dataset, it is not possible to tell if the new information is correct or not.

Another idea is to use a spell check to fix some inputs with wrong caracters that endup making the cleaning process harder. One benefit is that we would have more homogeneous data making the process of queries more relatiable. Also these features could be tigh together with the dataset from the post office, that way we could use it as reference when the user is typing a new entry offering tips like the type of street and the postal code based on geolocation. The disavantage of this method is that it is necessary to have a way to say if the word is wrong or not.

Moreover, we have that very few users contributed to the dataset, 70% of the data comes from 5 users. To improve the quality of the data and the number of users, google could offer a reward the top users. It could also use other users to verify the data, a lot of people use google maps service to move around the city and each time the user gets to where he wants to go, google could ask if the location is the right one. Also on my analytics, I could have used geolocation over a map. A lot of features could use this technique to give an idea of where things are located. An example could be to plot the geolocation data by users, that can give an idea of the area covered by each one.

```
In [ ]:
```