Wrangle OpenStreetMap Data

Abhishek Babuji

JULY 19, 2017

1 Introduction

1.1 Map Area

Velachery is a residential area in South Chennai, a metropolitan city in Tamil Nadu, India. This place is special to me since I was born and brought up in this area. Velachery as a whole draws a perfect balance between OLD and NEW Chennai and is a phenomenon in terms of growth.

1.2 CUSTOM Mapzen Metro Extract

The Dataset

2 Problems Encountered in the Map

2.1 Street Names

A lot of street names follow obscure naming patterns. Some are abbreviated, `["St", "St." = "Street"]`, `["Ave" = "Avenue"]`, `["Rd", "Rd." = "Road"]`, `["Extn", "Extn." = "Extension"]`, some are just misspelled, `["Strret", "Strret", "strret" = "Street"]`, irregular alphabet casing, or incorrect formatting.

2.1.1 Audit Street Names

The code below is a part of audit.py that looks for street names based on a mapping. The ones in the expected map are ignored, while the others are added to the `street_types` dictionary.

```
expected = ["Street", "Extension", "Road", "Street", "Avenue"]

def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)


def is_street_name(elem):
    return (elem.attrib['k'] == "addr:street")


def audit(osmfile):
    osm_file = open(osmfile, "r")
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):

        if elem.tag == "way":
            for tag in elem.iter("tag"):
                if is_street_name(tag):
```

```
                audit_street_type(street_types, tag.attrib['v'])
    osm_file.close()
    pprint.pprint(dict(street_types))
    return street_types
```

2.1.2 Observations after auditing Street Names

The irregularities in street names found are listed in the mapping dictionary of data.py with the key representing the poor street naming and the value representing the corrected street name.

```
street_type_mapping = { "St": "Street",
                        "st": "Street",
            "St.": "Street",
            "Ave": "Avenue",
            "Rd": "Road",
            "Rd.": "Road",
            "Extn": "Extension",
            "Extn.": "Extension",
            "Strret": "Street",
            "strret": "Street"
            }
```

2.1.3 Cleaning Street Names

The code below is a part of data.py that does the cleaning based on the pattern seen in the observations of inconsistent street types. Any occurrence of character match found in the `key` of the `street_type_mapping` dictionary is replaced by the `value` of that dictionary.

```
# Map invalid street types to correct ones and strip off non alphabetical characters off of the
    end of street types
# Returns False if the street type is unmappable or is not in expected, meaning it will be
    ignored.
def sanitize_street_type(street_type):
    if not street_type[-1].isalpha():
        return street_type[:-1]

    elif street_type in street_type_mapping.keys():
        return street_type_mapping[street_type]
    else:
        return False
```

2.2 Postal Codes

Postal codes that represent places in Chennai are supposed to be 6 digits long only. Furthermore, postal codes for all regions covered as a part of the map extract are supposed to range between two limits, some leniency can be shown towards the range as the exact boundaries very from source to source, but apart from the range that postal codes are supposed to be limited by, below is the list of some bad postal codes. For the given small extract, we see the following inconsistencies.

2.2.1 Audit Postal Codes

The code below is a part of audit.py that looks for postal codes that have more than 6 digits. This will return all inconsistencies in the postal code present in the data.

```
def audit_postal_code(error_codes, postal_codes, this_postal_code):
    # Append incorrect zip codes to list
    if this_postal_code.isdigit() == False:
        error_codes.append(this_postal_code)
    elif len(this_postal_code) != 6:
        error_codes.append(this_postal_code)
    else:
        postal_codes.update([this_postal_code])

def is_postal_code(elem):
    # Identify element tag as postal code
    return (elem.attrib['k'] == "addr:postcode")

def audit_post(osmfile):
    # Parse osm file for incorrect postal codes
    osm_file = open(osmfile, "r")
    error_codes = []
    postal_codes = 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_postal_code(tag):
                    audit_postal_code(error_codes, postal_codes, tag.attrib["v"])
    return error_codes, postal_codes

bad_list, good_list = audit_post(OSMFILE)
print (bad_list)
```

2.2.2 Observations after auditing Postal Codes

```
['6000036', '600 020.', '6000113', '6000036', '6000036', '6000036', '6000036', '6000042', '6000036',
'6000036', '6000036', '6000036', '6000036', '6000036', '6000036', '6000036', '6000036']
```

1. 7 digit postal codes instead of 6 digits (Extra zeroes added while typing).
2. White space between postal code and period at the end of the postal code.

2.2.3 Cleaning Postal Codes

The code below is a part of data.py that does the cleaning based on the pattern seen in the observations of inconsistent postal codes.

```
# Regular expressions used

zip_code_re = re.compile(r'^[0-9]{1,6}$')
fix_zipcode_state_short = re.compile(r'^[0-9.\s?]{1,8}$')


# Matches all bad zip codes that have around 8 digits including periods and spaces as seen in
    patterns of the bad_list in audit.py We will find a match for such zip codes and edit those
    alone.

# Sanitize zip codes to return a 6 digit zip code.
```

```
def sanitize_zipcode(zip_code):
    zip_code = zip_code.strip()
    m = zip_code_re.search(zip_code)
    if m:
        return zip_code

    elif fix_zipcode_state_short.search(zip_code):
        zip_code = zip_code[:3] + zip_code[4:7]
        return zip_code

# The correct zipcodes fall from the start digit to the 3rd digit appended with 4th digit to the
     7th digit
```

2.3 Database Queries and Statistics

2.3.1 File sizes

```
velachery_chennai.osm:  62.8 MB
velachery_chennai.osm.json:  87.6 MB
```

2.3.2 Number of documents

```
db.osmdata.find().count()

685070
```

2.3.3 Number of nodes

```
db.osmdata.find("type":"node").count()

557226
```

2.3.4 Number of ways

```
db.osmdata.find("type":"way").count()

127836
```

2.3.5 Number of unique users

```
db.osmdata.distinct("created.user").length

393
```

2.3.6 Number of users who have made only 1 post

```
db.osmdata.aggregate([{"$group":{"_id":"$created.user", "count":{"$sum":1}}},
{"$group":{"_id":"$count", "num_users":{"$sum":1}}},
{"$sort":{"_id":1}}, {"$limit":1}])

{ "_id" :  2, "num_users" :  99 }
"_id" in this case represents the number of users
```

2.3.7 Top 10 contributors

```
db.osmdata.aggregate([{"$group":{ "_id":"$created.user", "count":{"$sum":1}}},
{"$sort":{"count":-1}},{"$limit":10}])
```

```
{ "_id" : "maheshrkm", "count" : 109078 }
{ "_id" : "vamshikrishna", "count" : 71822 }
{ "_id" : "harishvarma", "count" : 45638 }
{ "_id" : "anushap", "count" : 39138 }
{ "_id" : "venkatkotha", "count" : 37680 }
{ "_id" : "harisha", "count" : 32044 }
{ "_id" : "saikumard", "count" : 28684 }
{ "_id" : "shivajim", "count" : 28656 }
{ "_id" : "Rahuldhanraj", "count" : 25926 }
{ "_id" : "premkumar", "count" : 25528 }
```

2.4 Additional data exploration using MongoDB queries

2.4.1 Top 10 appearing amenities

```
db.osmdata.aggregate([{"$match":{"amenity":{"$exists":1}}},
{"$group":{"_id":"$amenity",
"count":{"$sum":1}}},
{"$sort":{"count":-1}},
{"$limit":10}])
```

```
{ "_id" : "place_of_worship", "count" : 116 }
{ "_id" : "restaurant", "count" : 105 }
{ "_id" : "school", "count" : 99 }
{ "_id" : "atm", "count" : 50 }
{ "_id" : "hospital", "count" : 48 }
{ "_id" : "bank", "count" : 40 }
{ "_id" : "college", "count" : 35 }
{ "_id" : "pharmacy", "count" : 29 }
{ "_id" : "fuel", "count" : 29 }
{ "_id" : "parking", "count" : 24 }
```

2.4.2 Biggest religion in the area

```
db.osmdata.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"place_of_worship"}},
{"$group":{"_id":"$religion", "count":{"$sum":1}}},
{"$sort":{"count":1}},
{"$limit":1}])
```

```
{ "_id" : "muslim", "count" : 7 }
```

2.4.3 Most popular cuisines

```
db.char.aggregate([{"$match":{"amenity":{"$exists":1}, "amenity":"restaurant"}},
{"$group":{"_id":"$cuisine",
"count":{"$sum":1}}},
{"$sort":{"count":1}},
{"$limit":2}])
```

```
{ "_id" : "marine", "count" : 1 }
{ "_id" : "indian;vegetarian;local", "count" : 1 }
```

2.5 Additional Ideas

1. Consider collecting map data in forms with validation provided for the respective fields. For example, when entering pin-codes, ensure that, they satisfy the necessary conditions such as length of pin-code, start digit and end digit of pin-code. When entering phone numbers, consider checking for correct length of the numbers.

`BENEFITS:` Validation of data can be conducted during data entry itself. No additional investigation needs to be carried out to check for valid data entry.

`ISSUES:` The above idea is contingent on the fact that, there are clear boundaries when it comes to demarcation of areas of a map. For example, the area "Velachery" is considered to be belonging to Kanchipuram district in some sources and Thiruvanmyur district by other sources. This will result in a conflict in start digits of zip-codes. So additional training and personal inspection is required to train the volunteers for accurate data entries.

2. Instead of text entry, adopt a "check" entry from a drop down list for city names.

`BENEFITS:` This ensures that every entry is consistent and does not have any typographical errors, inconsistent letter spacing and inconsistent letter casing!

`ISSUES:` The main issue in implementing drop down check entries is that, sometimes, city names tend of have an alias! My city itself, has an alias. It used to be called Madras and was renamed in 1996 to Chennai! Similarly almost all cities were renamed at some point of time or the other. The drop down entry list must be updated with literally every possible name that the area has received and clearly be coded to represent that one name is in fact the alias of the other and not two separate names. The same problem can be extended to street names also! On a much larger scale, this would require the help of a domain expert and also additional training to be provided to the volunteers.

3. Make address entry a lot more specific by forcing entries to specific part that represent the address. For example, allow entry to a specific space for street name, another separate space for zip code and separate space for plot number. Doing it this way makes the data consistent instead of having different parts of an address or redundant, extra parts appended as a part of street address.

`BENEFITS:` This ensures that every entry is consistent and ensures that the address is not populated with the wrong parts of the address in the wrong places. All address names will follow the same pattern.

`ISSUES:` Erroneous data entries will be major problem. This part of the data entry requires further inspection to ensure that country name has not been entered in the place of state name or zip-code has been entered in the wrong place. The fact that this part requires to be inspected for validation almost makes it counterproductive to have such a data entry scheme. Data inspection needs to be conducted irrespective of the data entry scheme used. Whether the parts of the address needs to be entered in specific spaces, or the address can be entered as a single string, validation has to be performed to ensure correctness of the entry anyway.

3. Allow further details to represent phone numbers. A phone number could be a land-line or a mobile device number.

`BENEFITS:` Further categorization of phone number is now available. We now know if a phone number is a land-line or a mobile device. Knowing this will also help clean data better since there are easy patterns that can be deciphered. For example, all mobile phone numbers are 10 digits long, and all land-line phone numbers tend to be 8 digits long. It is now easier to write functions for phone number validation specific to the type of phone number. Before, a phone numbers could be anything, and could follow any inconsistent pattern (hyphens between STD code and phone number, no spacing between STD code and phone number), but now, it can be ensured that phone numbers now follow a consistent pattern.

`ISSUES:` Just like the previous idea, erroneous data entries will be major problem. Data inspection needs to be conducted irrespective of the phone number entry scheme used. Whether the parts of the phone number needs to be entered in specific spaces (separate space for STD code, separate space of phone number), or the phone number can be entered as a single string, validation has to be performed to ensure correctness of the entry anyway.

2.6 Conclusion

While it looks simply to find patterns in erroneous data of small sizes, larger size data will require better and more robust program coding for efficient auditing. Investigating data is a tedious process. Starting with the right approach is paramount. It is very easy to notice your entire effort go to waste if the methodology used for cleaning is inefficient. Patterns must be deciphered thoroughly or batch by batch for specific map areas depending on the time that can be sacrificed for cleaning. Countless inconsistencies can be found and handled if there were time to spare and a lot of patience to donate.

2.7 References

1. Udacity Forum Mentors: t0mkaka and Myles
2. 1-1 appointment with coach, Khushboo Tiwari