Wrangling OpenStreetMap Data

Sam Cumarasamy
Data Analyst Nanodegree

INTRODUCTION

This report is to discuss the data wrangling of the XML file of the Openstreetmap data. I've taken the Map of Sydney, Australia as that's where I live currently. As a result, I'm more familiar with the data and would be able to identify any bad entries. Also, there was no requirement to

The file was downloaded from the site Mapzen and uncompressed, the file "Sydney_australia.osm" was 320.5MB in size.

OVERVIEW OF OSM DATA FOR SYDNEY

CONVERSION OF XML TO JSON FORMAT

The OSM file was first converted into a JSON file after which the JSON file was then imported into a mongodb database. The file "json_create.py" is combination of python code that was used in the case study for wrangling Openstreet map. In the python file, the following function was used from the SQL exercise of the previously mentioned case study:

```
def get_element(osm_file, tags=('node', 'way', 'relation')):
    """Yield element if it is the right type of tag"""

    context = 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()
```

The above function was used to process the 320.5MB OSM file without any discernable impact on computer resources and this was processed in a timely manner. The above function only processes the three tags that "node", "way" and "relation" which was required for the purposes of this project.

Like in the case study, each XML record in the OSM file were processed into the following format:

```
{
"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"
}

As per what was done in the case study, the following things were done to the OSM data:

- Three types of top-level tags were processed: "node", "way" and “relation”
- All attributes of "node", "way" and “relation” were turned into regular key/value pairs, except for the attributes in the CREATED array were added under a key "created"
- Attributes for latitude and longitude were added to a "pos" array. The values inside "pos" array were converted into floats.
- If the second level tag "k" value starts with "addr:", it was added to a dictionary "address"
- If the second level tag "k" value didn’t’ start with "addr:", but contained ":", it was split it into a two-level dictionary like with "addr:".
- If there was a second ":" that separated the type/direction of a street, the tag was ignored.

Take for example:

<tag k="addr:housenumber" v="5158"/>

<tag k="addr:street" v="North Lincoln Avenue"/>

<tag k="addr:street:name" v="Lincoln"/>

<tag k="addr:street:prefix" v="North"/>

<tag k="addr:street:type" v="Avenue"/>

<tag k="amenity" v="pharmacy"/>

were turned into:

{...

"address": {

"housenumber": 5158,

"street": "North Lincoln Avenue"

}

"amenity": "pharmacy",

...

}

For the “way” tags, tags specifically named “nd” were treated in the following manner along:

<nd ref="305896090"/>

<nd ref="1719825889"/>

were turned into

"node_refs": ["305896090", "1719825889"]

In regards to the “relation” tags, the member tags were rolled into a list as per:

<member type="way" ref="61265623" role="from"/>

<member type="way" ref="173092678" role="to"/>

were turned into

"member": [{"ref": "61265623", "role": "from", "type": "way"}, {"ref": "173092678", "role": "to", "type": "way"}]

BRIEF OVERVIEW OF DATA

All the data is stored in the collection “Sydney” in the database called “OSM”. Also for the purposes of cleaning up the data for both the city and suburb fields, a collection called

"postcodes" was created in the "OSM" database. The data for the "postcodes" collection was taken from the link http://download.geonames.org/export/dump/AU.zip.

The file AU.txt (found in zip file) was converted into a CSV file using excel. From there, both the Sydney OSM file and the postcodes CSV file were imported into MongoDB via the script "wrangle.py".

The stats of the "Sydney" collection is as follows:

➔ db.sydney.stats({scale: 1024})

{

"ns" : "OSM.sydney",

"size" : 404792,

"count" : 1625186,

"avgObjSize" : 255,

"storageSize" : 129176,

"capped" : false,

"nindexes" : 6,

"totalIndexSize" : 62648,

"indexSizes" : {

"_id_" : 15824,

"id_1" : 18116,

"address.city_1_address.suburb_1" : 7160,

"address.suburb_1_address.city_1" : 7160,

"address.street_1" : 7212,

"address.postcode_1" : 7176

},

"ok" : 1

}

The breakdown of the tag types that were imported are as follows:

➔ db.sydney.aggregate([{"$group": {"_id": "$tag_type", "count": {"$sum": 1}}}, {"$sort": {"count": -1}}])

- {"_id": "node", "count": 1425174.0}
- {"_id": "way", "count": 195262.0}
- {"_id": "relation", "count": 4750.0}

ANALYSIS OF THE ADDRESS FIELD

The python program “analysis.py” was written to perform analysis across the address field for the collection Sydney in the OSM database. All the analysis in the subsequent sections are based on the output of the python program “analysis.py”.

The address field was found to have issues where data wrangling could be used to fix the majority of the data in this field. The number of records in the database that had an address field were:

- ➔ db.sydney.find({"address": {"$exists": true}}).count()
  - 19569.

POSTCODE ANALYSIS

Of the records that had an “address” field, 7738 records were found to hold a postal code. Of these records, 7687 records were found to be good records ie the postal code consisted of 4 digits within the range of 2000 – 2999, which was the correct range for postcodes for the state of NSW.

Aside from that, there were issues found. The following issues were found with the address.postcode field:

- ➔ 43 records were found to have postal codes along with the State abbreviation e.g. “NSW 2745”, “NSW 2234” etc. These records could be fixed by stripping the characters leaving behind the 4-digit postcode which is the correct format.
- ➔ 7 records were found whereby they didn’t fall within the postcode range of 2000 – 2999 (‘NSW 1460’, ‘1640’), were just incomplete (e.g. ‘2???’, ‘200’) or just characters (e.g. 'Phillip Street').

STREET TYPE ANALYSIS

The number of records with a street address:

- db.sydney.find({"address.street": {"$exists": true}}).count()
  - 18446

There were two issues found field after applying the following regular expression against the “address.street” field:

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

1. The first issue that was found was that there were street names that were considered incomplete. They were considered incomplete as the derived street types were not considered as one would consider as either standard abbreviations or complete street types. Some of the non-conforming street types found were:
   - u'Wolli': 74,

- u'Berith': 25,
- u'Broadway': 25,
- u'Square': 18,
- A couple of records had Chinese characters as street types like ‘2000 澳洲’ and ‘2026 澳洲.

2. The second issue found with the street types was that the street types were abbreviated. Examples were:
    a. Ave
    b. st
    c. Pl
    d. Streets
    e. Streett

SUBURB AND CITY ANALYSIS

The issues found for both suburb and city data could be split into 6 scenarios which were as follows:

Scenario	Case
1	Had a value of “Sydney” as city and as well as a value for suburb. Some of the address fields were found to be missing postcodes. {"$match": {"$and": [{"address.city": "Sydney"}, {"address.suburb":{"$exists": True}}]}}
2	Has Sydney as part of the value of city and has a postcode but had no suburb. {"$match": {"$and": [{"address.city": {"$regex": re.compile(r"(sydney)", re.IGNORECASE)}}, {"address.postcode": {"$exists": True}}, {"address.suburb":{"$exists": False}}]}}
3	Has a value for the city whose value was actually a Suburb and subsequently the suburb field didn’ot exist {"$match": {"$and": [{"address.city": {"$regex": re.compile(r"^((?!sydney).)*$", re.IGNORECASE)}}, {"address.suburb": {"$exists": False}}]}}
4	The city field doesn't exist but there was a value for the suburb. {"$match":{"$and": [{"address.city": {"$exists": False}}, {"address.suburb":{"$exists": True}}]}}, {"$group": {"_id": {"city": $address.city", "suburb": "$address.suburb", "postcode": $address.postcode"}, "count": {"$sum": 1}}}, {"$sort": {"suburb": 1}}
5	The value of the field city doesn't contain the value of 'Sydney' and has a value for the suburb field

	{"$match":{"$and": [{"address.suburb": {"$exists": True}}, {"address.city": {"$regex": re.compile(r"^((?!sydney).)*$", re.IGNORECASE)}}]}}
6	The address field was incomplete. There were records where there were no values for both city and suburb fields i.e. they just had only address.street populated. Alternatively, there were just address.street and address.postcode
7	There were suburban names that were all Upper case or just lower case e.g. “CROYDON PARK”. The normal standard way of naming was to capitalise the first letter and then lower case for the rest.

FIXING THE ADDRESS FIELD

Fixing the data involved 3 stages:

1. Clean the postcode to be the correct format
2. Update the street types to fit within the OSM standard ie full street types such Street, Avenue etc
3. Fix the suburb and city, such that all address fields have a complete set of suburb and city data. As part of this process, the postcode data was aligned with its corresponding suburb.

FIXING THE POSTCODE – FIX_POSTCODE.PY

In order to fix the postcode data, all postcode data were identified with the following function:

```
def postcode_query():
    pipeline = [{"$match": {"address.postcode":{"$exists": True}}},
                {"$group": {"_id": {"postcode": "$address.postcode"}, "count":{"$sum": 1}, "id_list": {"$push": "$id"}}}, {"$sort": {"postcode": 1}}]
    return pipeline
```

With the data that was identified using the above function, they were all filtered using regular expressions as per below:

- Filter postcodes that contain non-digit characters at the start or end of the string.

postcode_re = re.compile(r"^\D+\s+|\s+\D+$", re.IGNORECASE)

- Filter properly formatted postcodes

postcode_digit = re.compile(r"\d+", re.IGNORECASE)

- Filter postcodes that contained only characters

postcode_letters = re.compile(r"^(\s[a-zA-Z]+\s*)+$", re.IGNORECASE)*

The main function below:

```
def process_doc(db, pipeline):
    for doc in db.sydney.aggregate(pipeline):
        if doc[“_id”][“postcode”] is None:
            continue
        elif (postcode_letters.search(doc[“_id”][“postcode”])):
            pc = doc[“_id”][“postcode”]
            result = db.sydney.update_many({“address.postcode”: pc}, {“$push”: {“to_review”: “address_postcode”}})
elif (int(postcode_digit.search(doc[“_id”][“postcode”]).group()) < 2000) or (int(postcode_digit.search(doc[“_id”][“postcode”]).group()) > 2999): # This is to capture any postcodes that weren’t in the correct range
            pc = doc[“_id”][“postcode”]
            result = db.sydney.update_many({“address.postcode”: pc}, {“$push”: {“to_review”: “address_postcode”}})
elif postcode_re.search(doc[“_id”][“postcode”]):
            pc_orig = doc[“_id”][“postcode”]
            pc = postcode_re.sub(“”, pc_orig).strip() #Removing any non-digit characters
            result = db.sydney.update_many({"address.postcode": pc_orig}, {"$set": {"address.postcode": pc}})
```

As a result, I’m now left with the following postcodes which now need to be fixed manually.

Postcode: 'NSW 1460' - Record Count: 1

Postcode: '1640' - Record Count: 2

Postcode: '200' - Record Count: 1

Postcode: 'Phillip Street' - Record Count: 1

FIXING STREET TYPES – FIX_STREET.PY

The following query function was used to select all the streets from the database:

```
def fix_street(db, id_data, st):
    st_type = type_fix[st] #Street type in the mapping table
    for i in id_data:
        docs = db.sydney.find({"id": i})
        for d in docs:
            street_name = d["address"]["street"]
            fixed_st = street_type_re.sub(st_type, street_name) #replacing the street type with that in the mapping table using the regular expression
            db.sydney.update_many({"id": i}, {"$set": {"address.street": fixed_st}})
```

In the above function, each record that has a street name defined is processed. Using the regular expression “street_type_re” (as shown earlier), the street type is extracted from the field “address.street”. The street type is compared against a mapping table and then replaced with the correct street type. If the street type doesn’t exist in the mapping table, the record is flagged with a new field “to_review”. As stated earlier, streets that were incomplete were flagged for manual updates.

FIXING SUBURB AND CITY – FIX_SUBURB_CITY.PY

The first kind of records to fix was where there were missing the “address.city” field but had an “address.suburb” field. To do this the following update statement was run to set the “address.city” field to the value of “Sydney:

```
result = db.sydney.update_many({"$and": [{"address.city": {"$exists": False}}, {"address.suburb":{"$exists": True}}]}, {"$set": {"address.city": "Sydney"}})
```

To help with the next set of data cleanups, a new collection called “postcodes” was created and was used to extract the proper suburb names along with their corresponding postcodes. The data came from the site geonames.com The function is below:

```
def find_postcode(db, suburb):
    postcode_rec = db.postcodes.find_one({"$and": [{"suburb": re.compile(suburb, re.IGNORECASE)}, {"postcode": {"$gte": "2000"}}, {"postcode": {"$lte": "2999"}}]})
    if postcode_rec is None:
        return None, None
```

```
    else:
        return postcode_rec["suburb"], postcode_rec["postcode"]
```

Using the above function, it was possible to clean up the records that had a value for “address.city” field of “Sydney” and a valid value for the “address.suburb”. The cleanup was to make sure that all fields that had values in the address.city and address.suburb fields had correct postcodes in order to complete address field. This update process also made sure that the names had the first letter as uppercase and the rest of the letters lower case e.g. “Eastwood”.

```
for doc in db.sydney.aggregate(pipeline):
        suburb = doc["_id"]["suburb"]
        sb, pc = find_postcode(db, suburb)
        if ((sb is None) and (pc is None)):
            result = db.sydney.update_many({"$and": [{"address.city": "Sydney"}, {"address.suburb": suburb}]}, {"$push": {"to_review": "address_city_suburb"}}) #This captures any potential misspellings and invalid suburb names
        else:
            result = db.sydney.update_many({"$and": [{"address.city": "Sydney"}, {"address.suburb": suburb}]}, {"$set": {"address.suburb": sb, "address.postcode": pc}}) #This captures any potential misspellings and invalid suburb names
```

Then the next process was then to clean up the all the address.city fields that had a suburb as a value and these same records had no “address.suburb” values set. So to fix that, the “address.city” field was set to “Sydney” and the “address.suburb” field was updated with the previous value of “address.city”. Then by joining with the postcodes collection, the postcodes were all fixed up at the same time:

```
for doc in db.sydney.aggregate(pipeline):
        city = doc["_id"]["city"]
        sb, pc = find_postcode(db, city)
            if ((sb is None) and (pc is None)):
            result = db.sydney.update_many({"$and": [{"address.city": city}, {"address.suburb": {"$exists": False}}]}, {"$push": {"to_review": "address_city_suburb"}}) #This captures any potential misspellings and invalid suburb names
            else:
            result = db.sydney.update_many({"$and": [{"address.city": city}, {"address.suburb": {"$exists": False}}]}, {"$set": {"address.city": "Sydney", "address.suburb": sb,
```

"address.postcode": pc}}) #All correct names are set appropriately along with the correct postcodes

FURTHER EXPLORATION OF THE DATA

As seen earlier the number of records in the “Sydney” collection was:

- ➔ db.sydney.count()
  - 1625186

USER CONTRIBUTIONS TO THE DATASET

The number of users who have contributed to this dataset are:

- ➔ db.sydney.distinct("created.user").length
  - 2193

The top ten user contributors to the dataset are:

- ➔ db.sydney.aggregate({"$group": {"_id": {"user": "$created.user"}, "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 10})

{ "_id" : { "user" : "balcoath" }, "count" : 117416 }

{ "_id" : { "user" : "inas" }, "count" : 90999 }

{ "_id" : { "user" : "TheSwavu" }, "count" : 74180 }

{ "_id" : { "user" : "ChopStiR" }, "count" : 61108 }

{ "_id" : { "user" : "aharvey" }, "count" : 57928 }

{ "_id" : { "user" : "Leon K" }, "count" : 49041 }

{ "_id" : { "user" : "cleary" }, "count" : 44296 }

{ "_id" : { "user" : "Rhubarb" }, "count" : 40939 }

{ "_id" : { "user" : "AntBurnett" }, "count" : 37852 }

{ "_id" : { "user" : "Warin61" }, "count" : 37318 }

Looking at the user contributions to the dataset, user “balcoath” contributed to about 7.2% of the data. The next highest contribution of data which is by “inas”, is at around 5.6%. There doesn’t seem to be any lopsided contribution to the dataset and so there is no suggestion here of any gamification or any sense of bots going overboard in creating records.

It was found that there was a field “created_by” which stored the name of the software with which the users have created or updated the dataset with. Not all the records hold this field however it was thought to be interesting to see.

Number of records with the “created_by” field:

- ➔ db.sydney.find({"created_by": {"$exists": true}}).count()
  - 12551

Here are the top 5 pieces of software used for creating/updating the records:

- ➔ db.sydney.aggregate({"$match": {"created_by": {"$exists": true}}},{"$group": {"_id": {"software": "$created_by"}, "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 5})

{ "_id" : { "software" : "JOSM" }, "count" : 11250 }

{ "_id" : { "software" : "YahooApplet 1.0" }, "count" : 212 }

{ "_id" : { "software" : "almien_coastlines" }, "count" : 189 }

{ "_id" : { "software" : "Potlatch 0.8a" }, "count" : 122 }

{ "_id" : { "software" : "Merkaartor 0.10" }, "count" : 86 }

AMENITY AND SHOP OBSERVATIONS

The top ten amenities in this dataset are as follows:

- ➔ db.sydney.aggregate({"$match": {"amenity": {"$exists": true}}}, {"$group": {"_id": {"amenity": "$amenity"}, "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 10})

{ "_id" : { "amenity" : "parking" }, "count" : 3984 }

{ "_id" : { "amenity" : "bench" }, "count" : 1334 }

{ "_id" : { "amenity" : "school" }, "count" : 1070 }

{ "_id" : { "amenity" : "restaurant" }, "count" : 1030 }

{ "_id" : { "amenity" : "cafe" }, "count" : 869 }

{ "_id" : { "amenity" : "toilets" }, "count" : 800 }

{ "_id" : { "amenity" : "drinking_water" }, "count" : 746 }

{ "_id" : { "amenity" : "fast_food" }, "count" : 670 }

{ "_id" : { "amenity" : "place_of_worship" }, "count" : 617 }

{ "_id" : { "amenity" : "bicycle_parking" }, "count" : 502 }

It was interesting to note that car parks (“parking”) took the number one spot for the amount of records under amenity in the Sydney collection. That could be taken to mean that there was a strong interest in knowing where all the car parking spaces were in Sydney.

In terms of the amenity “restaurant”, because I love eating, the top cuisines were found to be:

➔ db.sydney.aggregate({"$match": {"amenity": {"$exists": true}, "amenity": "restaurant"}}, {"$group": {"_id": "$cuisine", "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 5})

{ "_id" : null, "count" : 498 }

{ "_id" : "thai", "count" : 72 }

{ "_id" : "chinese", "count" : 65 }

{ "_id" : "italian", "count" : 57 }

{ "_id" : "pizza", "count" : 43 }

For the restaurant dataset, the majority of records didn’t have the “cuisine” field set, however, it’s interesting to note that the data matched the general observations for eating out here in Sydney. Thai food is the highest followed closely by Chinese in terms of the popular types of food.

In terms of “fast_food”, it’s no surprise as to what’s the most favourite past time:

➔ db.sydney.aggregate({"$match": {"amenity": {"$exists": true}, "amenity": "fast_food"}}, {"$group": {"_id": "$name", "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 10})

{ "_id" : "McDonald's", "count" : 98 }

{ "_id" : null, "count" : 67 }

{ "_id" : "KFC", "count" : 58 }

{ "_id" : "Subway", "count" : 53 }

{ "_id" : "Red Rooster", "count" : 22 }

{ "_id" : "Hungry Jacks", "count" : 21 }

{ "_id" : "Oporto", "count" : 18 }

{ "_id" : "Domino's Pizza", "count" : 15 }

{ "_id" : "McDonalds", "count" : 13 }

{ "_id" : "Pizza Hut", "count" : 12 }

In the above resultset, the main highlight is the lack of data validation. There are two results for “McDonalds” ie “McDonald’s” and “McDonalds”.

Looking at the data for shops:

➔ db.sydney.aggregate({"$match": {"shop": {"$exists": true}}}, {"$group": {"_id": {"shop": "$shop"}, "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 10})

{ "_id" : { "shop" : "supermarket" }, "count" : 471 }

{ "_id" : { "shop" : "convenience" }, "count" : 349 }

{ "_id" : { "shop" : "clothes" }, "count" : 183 }

{ "_id" : { "shop" : "mall" }, "count" : 180 }

{ "_id" : { "shop" : "alcohol" }, "count" : 142 }

{ "_id" : { "shop" : "hairdresser" }, "count" : 122 }

{ "_id" : { "shop" : "bakery" }, "count" : 116 }

{ "_id" : { "shop" : "car_repair" }, "count" : 109 }

{ "_id" : { "shop" : "department_store" }, "count" : 86 }

{ "_id" : { "shop" : "bicycle" }, "count" : 78 }

Digging into further the top result of supermarkets:

➔ db.sydney.aggregate({"$match": {"shop": {"$exists": true}, "shop": "supermarket"}}, {"$group": {"_id": {"name": "$name"}, "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 10})

{ "_id" : { "name" : "Woolworths" }, "count" : 112 }

{ "_id" : { "name" : "Coles" }, "count" : 75 }

{ "_id" : { "name" : "Aldi" }, "count" : 49 }

{ "_id" : { "name" : "IGA" }, "count" : 40 }

{ "_id" : { "name" : null }, "count" : 34 }

{ "_id" : { "name" : "ALDI" }, "count" : 14 }

{ "_id" : { "name" : "Franklins" }, "count" : 6 }

{ "_id" : { "name" : "Foodworks" }, "count" : 3 }

{ "_id" : { "name" : "Harris Farm Markets" }, "count" : 3 }

{ "_id" : { "name" : "7/11" }, "count" : 3 }

Again, the above results show the issue of data integrity and validation. There are two results for Aldi being “Aldi” and “ALDI”. It means that Aldi will have 63 stores as opposed to 49.

Even when the address fields were cleaned up, there were still further issues found with the data. Here’s a couple of examples of records with incorrect data that were found in a random manner:

{

```
"name" : "Sydney Ice Arena",
"leisure" : "sports_centre",
"address" : {
        "suburb" : "Bella Vista",
        "city" : "Sydney"
    },
    "id" : "134201234",
    "tag_type" : "way"
}
{
     "shop" : "mall",
     "building" : "retail",
"name" : "Norwest Market Town",
    "visible" : null,
     "address" : {
         "suburb" : "Bella Vista",
         "city" : "Sydney"
     },
     "id" : "227891709",
     "tag_type" : "way"
 }
```


In the above 2 records (which were edited for readability purposes), the “address.suburb” field has a value of “Bella Vista”. This is incorrect. Those buildings actually are located in the suburb of “Baulkham Hills. This shows that further audit of the data would be required to make sure that the data is correct.

SUMMARY/CONCLUSION

It’s been shown in this dataset, that there are issues with both data integrity and validation. Take for example the values of “McDonald’s” and “McDonalds” and also “ALDI” and “Aldi”. If one was doing a search, it’s going to give potentially different results. In summary, throughout this exercise of wrangling, the following issues were found with the data:

- Postcodes weren't validated to being a 4-digit code as well as being in the range of 2000-2999 for NSW.
- Street names having invalid street types.
- Suburbs being misspelt or being written to the wrong field (ie city). It was even missing in other cases.
- An issue was found whereby two places ie the ice rink and a marketplace were in the wrong suburb.
- Incomplete data was prevalent. Take for example the restaurant information. Of 1030 records where the amenity was a restaurant, 498 records didn't hold cuisine information. This would be very useful information for any venture wanting to use such information.

So, what can be done? Here are some suggestions as to improve the situation:

- There has to be a handful of software that would need to be certified by the OpenStreetMap foundation. Certification is to be determined by how well it validates and preserves data integrity.
- Software should be able to integrate with the local business registry such as ASIC (Australian Securities & Investments Commission) to identify correct locations for businesses being entered into the map.
- Information such as suburbs, city and postcode can be validated against Australia Post using integration.
- Using the free APIs with Google Maps, Here Maps and maybe Apple Maps, geo location information can be confirmed.

The disadvantages with the above proposals are that the software used to enter the data could become:

- Quite complicated and therefore may introduce costs in to what's considered free.
- Fragmented in the sense that different types of software for different countries. It will require different integrations to validate the data etc.
- A slow process due to the care that will need to be taken to enter in good data. It might also result in more volunteers required to help clean the data.

Overall, it can be done and needs to be done. This is because OpenStreetMaps is a worthwhile venture and it will be good as the data it holds. If bad data is prevalent then the whole mapping exercise will prove to be useless.