OpenStreetMap Data Case Study

Map Area

Sevilla, Andalusia, Spain

- https://www.openstreetmap.org/relation/342563

This map is of my hometown, so I can easily check for data accuracy.

Files used

1. Data.py It is used to parse the Open Street Map XML and create the following 5 tables on csv
    1. nodes.csv: contains the node information
        - id: primary key
        - user: user name of the creator
        - uid: user Id
        - version
        - lat: latitude
        - lon: longitude
        - timestamp
        - changeset
    2. nodes_tags.csv: contains the tags asociated to the aforementioned nodes
        - ind: primary key
        - id:foreign key to the node id
        - key:the catalogue of tag
        - value:
        - type: type of tag
    3. ways.csv: contains the way information
        - id: primary key
        - user: user name of the creator
        - uid: user Id
        - version
        - timestamp
        - changeset
    4. ways_tags.csv: contains the tags asociated to the aforementioned nodes
        - Ind: primary key
        - id:foreign key to the way id
        - key:the catalogue of tag
        - value:
        - type: type of tag

5. ways_nodes.csv: contains the tags asociated to the aforementioned nodes
    - id:foreign key to the way id
    - node_id: foreign key to the node id
    - position: position in the list of nodes belonging to the way

The Ind primary keys on the tags are added in order to update only these rows in the database when changes are made

2. QSL data.py: It is used to create the SQL database (named OSMSevilla.db) where the tables are a mirror of the aforementioned csv tables.

3. OSM Queries.sql: It is used to perform queries to OSMDB.db using sqlite. This queries are then transferred to the OSM audit.py to select only the problematic data to be treated. Only the problematic rows are treated to be corrected when possible.

4. OSM audit.py: It is used to perform changes to the OSMDB.db in order to correct the problems found.
    1. It performs queries to the DB to select only the problematic rows (when possible) and imports them into dataframes
    2. This dataframes are transformed and corrected
    3. the corrected dataframes are then exported to the database for update

Problems encountered in the Map

The first encountered problem was handling Unicode strings as the Spanish alphabet uses several special characters which where not correctly handled when converted to UTF-8 by DictWriter. Fortunately Python 3's built-in csv module supports Unicode natively by selecting encoding='utf-8-sig' when opening the csv. It worked!

Once ready, I screened a small sample size area around my house (so I know all the places around) and I run it against the data.py file done during the lesson exercises obtaining the DBTest.db

Once I got an idea of the problems I could found, I created the OSMSeville.db from the complete map of Seville (87MB) using the data.py and SQL data.py files.

Note that in SQL data.py I added an index as primary key to the nodes_tags and ways_tags tables so I can perform the update of selected rows by index.

Once, my database was ready, I followed the steps below to systematically screen the values of the tags:

1. First, I filtered the tags by address type, since it should be the most regular. Here below I list the problems per found per key:
    1. Country
        1. No issues, all values to "ES"
    2. State
        1. State key should be replaced by Country
    3. Province
        1. No issues, all values to "Sevilla"
    4. City

        1. Some values of Sevilla are not correctly written or are in another language
        2. Some values included neighborhoods as for example " Sevilla / Casco Antiguo / San Bartolomé but seems structured and easy to undstand
    5. Postcode
        1. No issues, all postal codes are consistent
    6. Zip
        1. Zip key should be replaced by Postalcode
    7. Street
        1. Some abbreviations are found, although rare.
    8. House number
        1. There are some numbers with access letter associated, finding 3 acc A and 3A
        2. Some contains a list in 2 different forms: "1;2;3;4;5" "and 1-2-3-4-5".

2. Then I took a look to the rest of tags types. Here below I list the problems per found per key:

    1. Phone
        1. Some keys where of type contact and other type regular
    2. Color
        1. Some colors are in HEX (#D6B290) and others as a color noun.

1.2 and 1.6 Replace and consolidate tag keys

In the file audit.py I created the functions to update any chosen column in accordance with a given dictionary.

In this case we will remove some redundant tag keys and change them by more appropriate ones.

The query filters the results to only treat the desired keys ("zip" and "state")

```
def update_name(name, mapping):
    # YOUR CODE HERE
    for k, v in mapping.items():
        if re.search(rf"{k}", name, re.IGNORECASE):
            name = name.replace(k,mapping[k])
    return name

def update_tags(df, column, mapping):
    df2 = df
    dummy = "a"
    for i, row in df.iterrows():
        dummy = row[column]
        df2[column][i]= update_name(dummy, mapping)
    return df2

query_tag_keys = """
SELECT ind, key, value
FROM nodes_tags
where (key = "zip") or (key = "state");
"""
mapping_keys = {
             "zip": "postcode",
             "state": "country",
            }

tag_keys = pd.read_sql_query(query_tag_keys, conn)

Corrected_keys = update_tags(tag_keys, "key", mapping_keys)
```

As it can be seen, both update_tags and update_name functions can be used for any correction. Only the query and mapping changes.

Therefore, for the next corrections only the query and mapping will be given as the other functions are reused.

Finally the following function is used to update the database with the information contained in the corrected dataframe

```
def update_table_from_dataframe(conn, df, table):
    """ update a table from the dataform
    :param conn: Connection object
    :param df: dataform withe the info to update
    :param table: table to be updated
    :return:
    """
    try:
        cursor=conn.cursor()
        # creating column list for insertion
        cols = ",".join([str(i) for i in df.columns.tolist()])
        # Insert DataFrame recrds one by one.
        for i,row in df.iterrows():
            sql = "REPLACE INTO " + table + " ("+cols+") VALUES
"+str(tuple(row))+";"
            print(sql)
            cursor.execute(sql)
            # the connection is not autocommitted by default, so we must commit to
save our changes
            conn.commit()

    except Error as e:
        print(e)
```

In order to be able to update the database easily, I decided to perform two diferent queries: one on the nodes_tags table and the other on the ways_tags tables.

Note that we can also perform any of the shown queries to the combined nodes_tags and ways_tags tables as they share the same columns.

This can be done by performing the a UNION subquery and then performing the desired query on the subquery.

Thus we need only to replace

```
SELECT ind, key, value
FROM nodes_tags
where (key = "zip") or (key = "state");
```

by the following query including the UNION subquery

```
SELECT tags.ind, tags.key, tags.value
FROM (SELECT * FROM nodes_tags UNION
      SELECT * FROM ways_tags) as tags
where (tags.key = "zip") or (tags.key = "state");
```

From now on we will only include the query on nodes_tags for simplicity.

1.4 standardize Sevilla city

After screening of the city names, I created a dictionary of Sevilla misspelings to replace them by "Sevilla".

```
SELECT ind, key, value
FROM nodes_tags
where key = "city" and type = "addr" ;
```

```
Corrected_city = update_tags(city, "value", mapping_city)

mapping_city = {
             "Sevila": "Sevilla",
             "Seville": "Sevilla",
             "Sevillla": "Sevilla",
             "41008": "Sevilla",
             "41010": "Sevilla"
            }
```

1.7 remove street abbreviations

After screening of the street names, I created a dictionary of abbreviations to replace them by the full noun.

```
SELECT ind, key, value
FROM nodes_tags
where (key = "street") and type = "addr" ;
```

```
Corrected_street = update_tags(street, "value", mapping_street)

mapping_street = {
```

```
                 "C/": "Calle",
                 "Av.": "Avenida",
                 "AVDA.": "Avenida",
                 "Ctra.": "Carretera",
                 "CTRA.": "Carretera",
                 "CRTA.": "Carretera",
                 "CR": "Carretera"
                }
```

1.8 standardize house numbers

The following changes are made:

- remove the acc abbreviation
- change ";" for "-" in the number list

```
SELECT ind, key, value
FROM nodes_tags
WHERE key = "housenumber" and type = "addr" and (value LIKE '%;%' or value LIKE
'%acc%');
```

```
Corrected_housenumbers = update_tags(housenumbers, "value", mapping_housenumbers)

mapping_housenumbers = {
                 ";": "-",
                 " acc": "",
                }
```

2.1 standardize phone numbers

In the file audit.py the following changes are made:

- tag type is changed for phone numbers from regular to contact

~~~
SELECT ind, key, value, type
FROM nodes_tags
where (key = "phone") and type = "regular";

```python
Corrected_phone = update_tags(phone, "type", mapping_phone)

mapping_phone = {
                 "regular": "contact"
                }
~~~

2.2 standardize colors

The follwoing changes are made:

- the color values that are expresed in nouns are changed into HEX codification

```
SELECT ind, key, value
FROM nodes_tags
where (key = "colour") and not value LIKE '%#%';
```

```
Corrected_color = update_tags(color, "value", mapping_color)

mapping_color = {
             "white": "#ffffff",
             "black": "#000000",
             "red": "#ff0000",
             "yellow": "#FFFF00",
             "blue": "#0000FF"
            }
```

Other checks performed

Postal Codes

Once the "zip" key is changed to "postcode", all postcodes are checked for consistency

```
SELECT tags.value, COUNT(*) as count
FROM (SELECT * FROM nodes_tags
      UNION
      SELECT * FROM ways_tags) tags
WHERE tags.key='postcode' and not value LIKE '%41%'
GROUP BY tags.value;
```

Only one postcode was found inconsistent, that should be removed

```
    #   value   count
   1    18360   1
```

Cities included

```
SELECT tags.value, tags.key, tags.type, COUNT(*) as count
FROM (SELECT * FROM nodes_tags UNION ALL
      SELECT * FROM ways_tags) tags
```

```
WHERE tags.key = "city" and not tags.value LIKE "%Sevilla%"
GROUP BY tags.value
ORDER BY count DESC;
```

These are the other cities included, which are correct since they limit with Seville:

```
#|value|key|type|count
1|Tomares|city|addr|27
2|Camas|city|addr|25
3|San|Juan|de|Aznalfarache|city|addr|16
4|Castilleja|de|la|Cuestacity|addr|11
5|Alcalá|de|Guadaíra|city|  addr|6
6|Mairena|del|Aljarafe|city|addr|5
7|Alcalá|de|Guadaira|city|addr|3
8|Coca|de|la|Piñeracity|addr|3
9|Dos|Hermanas|city|addr|3
10|Montequinto|city|addr|3
11|camas|city|addr|3
12|Gelves|city|addr|1
```

Data Overview and Additional Ideas

This section contains basic statistics about the dataset and some additional ideas about the data in context.

File sizes

```
Seville map.osm ......... 87 MB
OSMSeville.db .......... 53 MB
nodes.csv ............. 29 MB
nodes_tags.csv ........ 4 MB
ways.csv .............. 3 MB
ways_tags.csv ......... 5 MB
ways_nodes.cv ......... 12 MB
```

Number of nodes

```
SELECT COUNT(*) FROM nodes;
```

355969

Number of ways

```
SELECT COUNT(*) FROM ways;
```

56897

Number of unique users

```
SELECT COUNT(DISTINCT(e.uid))
FROM (SELECT users, uid FROM nodes UNION ALL SELECT user, uid FROM ways) e;
```

1046

Top 10 contributing users to nodes and ways

```
SELECT e.user, COUNT(e.uid) as count
FROM (SELECT uid, user FROM nodes UNION ALL SELECT uid, user FROM ways) e
GROUP BY e.user
ORDER BY count DESC
LIMIT 10;
```

```
#   user    count
1   erlenmeyer  104084
2   Juan Pedro Ruiz 50931
3   BoomEngine  33827
4   cirdancarpintero    28737
5   nonopp  15810
6   avm 14962
7   nanino90    11968
8   Alberto Molina  11090
9   Ro5597  8529
10  AtomMapper  7619
```

Top 10 contributing users to tags (both for nodes and ways)

```
SELECT total.user, count(total.ind) as count
FROM (select w.user, w.ind FROM (SELECT ways.user, ways_tags.ind
FROM ways_tags JOIN ways WHERE ways_tags.id = ways.id) as w UNION select n.user,
n.ind FROM (SELECT nodes.user, nodes_tags.ind
FROM nodes_tags JOIN nodes WHERE nodes_tags.id = nodes.id) as n) as total
GROUP by total.user
Order by count DESC
Limit 10;
```

```
#   user    count
1   erlenmeyer  45023
```

```
2   Juan Pedro Ruiz 26770
3   nyuriks 19686
4   BoomEngine  13496
5   mapman44    10263
6   avm 10127
7   sanchi  6944
8   Ro5597  6395
9   CristinaDC  5687
10  AtomMapper  5480
```

Number of users appearing less than 10 times

```
sqlite> SELECT COUNT(*)
FROM
    (SELECT e.user, COUNT(*) as num
     FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways) e
     GROUP BY e.user
     HAVING num<10)  u;
```

642

Additional Ideas

Contributor statistics

The contributions follow a pareto distibution

- Top user contribution percentage (“erlenmeyer”) 25.21%
- Combined Top 10 users contribution 69.65%
- Combined user with less than 10 contributions: 61.38% of all users

Which indicates the presence of bots in the 10 biggest contributors.

Additional Data Exploration

Top 10 keys

Let's see what kind of information we can find appart from addr info

```
SELECT tags.key, tags.type, COUNT(*) as num
FROM (SELECT * FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags
WHERE not tags.type = "addr"
GROUP BY tags.key
ORDER BY num DESC
LIMIT 10;
```

Obtaining the following table

```
#   key type    num
1   highway regular 27465
2   building    regular 21897
3   natural regular 18217
4   name    regular 17266
5   source  regular 12679
6   levels  building    7886
7   oneway  regular 7881
8   amenity regular 7275
9   surface regular 4899
10  leaf_type   regular 4595
```

As we can see there are three main categories

1. Ways (highway, oneway)
2. buildings (building, levels, amenity)
3. nature (natural, leaf_type)

1. ways

This category is all about types of ways, traffic signals... useful by boring for this exercise.

```
SELECT tags.key, tags.type, tags.value, COUNT(*) as num
FROM (SELECT * FROM nodes_tags UNION ALL SELECT * FROM ways_tags) tags
WHERE not tags.type = "addr" and (tags.key ="highway" or tags.key ="oneway")
GROUP BY tags.value
ORDER BY num DESC
LIMIT 5;

#   key type    value   num
1   oneway  regular yes 7060
2   highway regular residential 5964
3   highway regular crossing    4319
4   highway regular footway 3571
5   highway regular service 2685
```

From now on, we will not consider the way tags and only query on the node tags

1. buildings

This category is all about types of buildings... useful by boring for this exercise but also about amenity types, which is better

In the sake of brevety I do not show the query, similar to the one done for ways except that the key filter is set to buildings, levels and amenity.

Let's focus on amenity

```
SELECT key, value, COUNT(*) as num
FROM nodes_tags
WHERE  key ="amenity"
GROUP BY value
ORDER BY num DESC
LIMIT 5;

#   key value   num
1   amenity bench   1214
2   amenity bar 654
3   amenity restaurant  636
4   amenity parking 524
5   amenity bicycle_parking 378
```

As we can see there are a lot of bars and restaurant in Seville, which makes sense since it is a quite touristic city.

If we focus ocn the restaurants

Most popular cuisines (restaurant, fast food, cafe...)

```
SELECT nodes_tags.value, COUNT(nodes_tags.value) as num
FROM nodes_tags JOIN nodes
WHERE nodes_tags.id=nodes.id and nodes_tags.key='cuisine'
GROUP BY nodes_tags.value
ORDER BY num DESC
LIMIT 5;

#   value   num
1   regional    70
2   spanish 58
3   pizza   39
4   tapas   26
5   burger  22
```

Most popular cuisines only for restaurants

```
SELECT nodes_tags.value, COUNT(*) as num
FROM nodes_tags JOIN (SELECT DISTINCT(id) FROM nodes_tags WHERE
value='restaurant') i
WHERE nodes_tags.id=i.id and nodes_tags.key='cuisine'
GROUP BY nodes_tags.value
ORDER BY num DESC
LIMIT 5;
```

```
#   value   num
1   regional    59
2   spanish 40
3   pizza   19
4   tapas   19
5   italian 13
```

As we can see, the local food is very popular

let's see how many including everything that sounds like spanish, local, tapas...

```
SELECT COUNT(nodes_tags.value) as num
FROM nodes_tags JOIN nodes
WHERE nodes_tags.id=nodes.id and nodes_tags.key='cuisine' and (nodes_tags.value
LIKE "%spanish%" or nodes_tags.value LIKE "%local%" or nodes_tags.value LIKE
"%regional%" or nodes_tags.value LIKE "%tapas%");
```

We found 189 amenities, which is almost half of the total 414.

3. nature

This category is all about nature. Perfoming a similar query than for ways, we can find that most of the information is realted to trees (17770 trees), incluidng its type of leaf.

Suggestions for improving the data or its analysis

As can be seen there is a huge ammount of different information on the tags about very diverse subjects. I think that its structure into key, value, type is too simple and may not be enough.

As a suggestion I would create the concept of datasets. A dataset is a group of fields that always go together since they all descrive the same concept.

For example address could be a dataset. The dataset Address would contain the datafields: country, state, province, city, postalcode, street, housenumber...

The dataset tree for example could contain the datafields: genus, leaf_cycle, leaf_type...

In the tags we would only refer to the dataset ID and then you could retreive the information of the dataset filds through a join.

For example:

Node_tag:

tag.key= dataset_tree, tag.value = 123456, tag.type = dataset

DATASET:

dataset_tree.id = 123456, dataset_tree.genus = citrus, dataset_tree.leaf_cycle= evergreen, dataset_tree.leaf_type = broadleaved

1. Benefits

1. In this way we would have different dataset tables, all structured in the same way with predefined fields, helping to improve consitency.
2. If we were interested only in the city threes we would only consult the table of the trees dataset and via a join with the nodes we could locate them in the map.
3. It would make very easy to add data to the map since the datafields information could be stored and mantained in a different website and we only would need to create the dataset tags on OSM. When interested in consulting the datafields of a given dataset in the map, OSM would make a query to the third party website API and get the datafields through the dataset ID.
4. The datasets can be a closed list so when people want to contribute they will already have the list of possible datasets and they fields, making easier to standardize information

2. Anticipated Problems

1. People could introduce information as regular tags if they do not realize that a dataset for this pourpose already exists, having the same type of information in two diferent tables (the tag table and the dataset table)
2. The dataset maintained in third party websites could dissaperar or even change IDs, making the dataset tags obsolete or wrong.

Conclusion

I was actualy impressed by the amount of information and its consistency, completness and accuracy. Regarding completness and consistency, I guess OSM checks for it. Regarding accuracy I guess some people from Seville already followed this course! I also noticed that the information is sometimes very specific, like the one related to the trees and tree leafs or buildings style and color. It seems that some people are using Open Street Map to perform its invertigations or classifications, which I found a great idea.