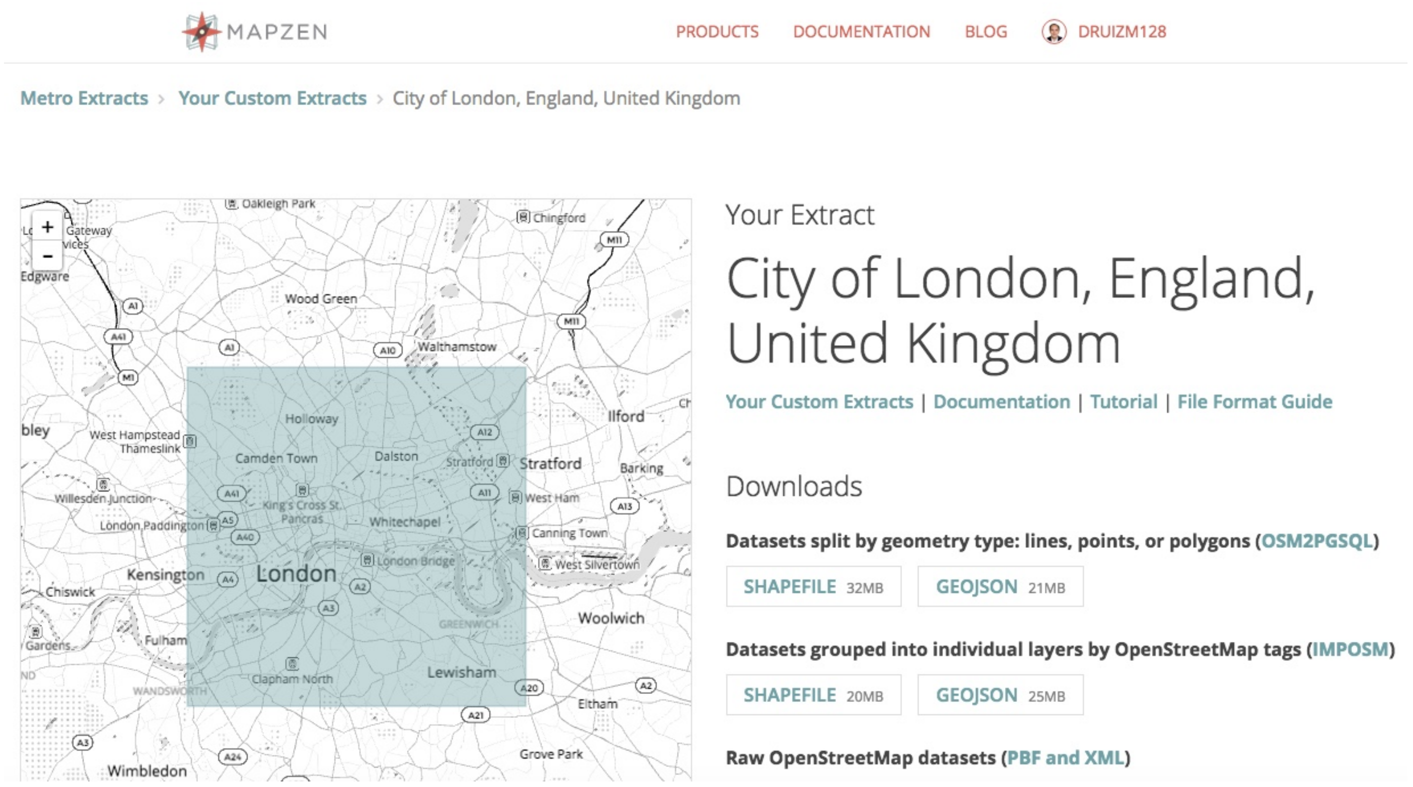
A case study of data wrangling using the City of London Open Street Map data

Introduction:

The following document explores the City of London in the United Kingdom using the information available in the Open Street Map. The purpose of this analysis is to use Data Wrangling techniques to load, audit, clean and analyze a big dataset using Python and SQLite. The city of London OSM dataset is used as a case study.


MAPZEN
PRODUCTS
DOCUMENTATION
BLOG
DRUIZM128
Metro Extracts > Your Custom Extracts > City of London, England, United Kingdom
Your Extract
City of London, England, United Kingdom
Your Custom Extracts | Documentation | Tutorial | File Format Guide
Downloads
Datasets split by geometry type: lines, points, or polygons (OSM2PGSQL)
SHAPEFILE 32MB
GEOJSON 21MB
Datasets grouped into individual layers by OpenStreetMap tags (IMPOSM)
SHAPEFILE 20MB
GEOJSON 25MB
Raw OpenStreetMap datasets (PBF and XML)

The organization of this documents is as follows:

1. Description of auditing process
2. Description of cleaning plan and process
3. Data overview
4. Additional ideas
5. Finally, the conclusion of this case study is presented.

The Python libraries used to complete this project are the following.

In [2]:
```
# Audit and Write CSV
import xml.etree.cElementTree as ET
import pprint
import re
from collections import defaultdict
# Audit file
import csv
import codecs
import cerberus
import sqlite3
# Analisys and graphs
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

1. Description of auditing process:

The OSM file is large and contains a wide array of information from nodes and ways. The efforts of the auditing process were focused in auditing the following information with the purpose of showing how to handle different problems:

- Data types
- Node coordinates (latitude and longitude)
- Postal code format
- Street names
- Subway stations

I conducted this auditing process using the measurements of data quality as needed:

- validity
- accuracy
- completeness
- consistency
- uniformity

a) Data types

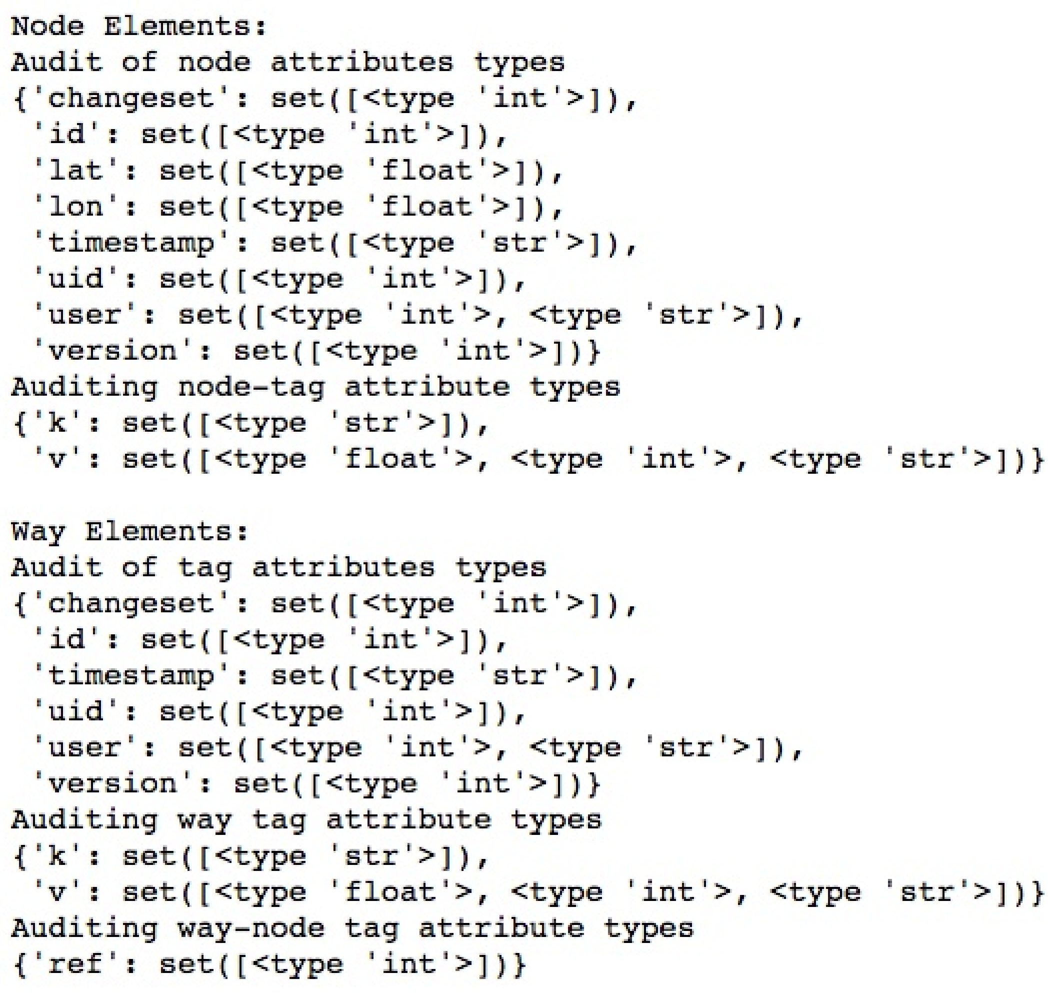
The data types were checked for each element attribute in nodes and ways using the following code.

In [ ]:
```
# A function to audit the type of the attributes of an element
def audit_attribute_type(types_dictionary, attributes):

    for attribute in attributes:
        value = attributes[attribute]
        if value == "NULL" or value == "" or value == None:
            types_dictionary[attribute].add(type(None))
        elif value.startswith("{") and value.endswith("}"):
            types_dictionary[attribute].add(type([]))
        elif is_number(value):
            try:
                int(value)
                types_dictionary[attribute].add(type(1))
            except ValueError:
                float(value)
                types_dictionary[attribute].add(type(1.1))
        else:
            types_dictionary[attribute].add(type("a"))
```

In the following figure it can be seen that with the exception of user and value (v) attributes, all the fields have one data type. In the case of user this is not a problem as the value represents a key and all entries can be treated as strings.

In the case of the node-tag and way-tag value attribute, types need to be treated on a case by case basis, as they represent different information. For example, some values represent postal codes, labels, hours, names, maximum speed allowed, apartment numbers, etc.



```
Node Elements:
Audit of node attributes types
{'changeset': set([<type 'int'>]),
 'id': set([<type 'int'>]),
 'lat': set([<type 'float'>]),
 'lon': set([<type 'float'>]),
 'timestamp': set([<type 'str'>]),
 'uid': set([<type 'int'>]),
 'user': set([<type 'int'>, <type 'str'>]),
 'version': set([<type 'int'>])}
Auditing node-tag attribute types
{'k': set([<type 'str'>]),
 'v': set([<type 'float'>, <type 'int'>, <type 'str'>])}

Way Elements:
Audit of tag attributes types
{'changeset': set([<type 'int'>]),
 'id': set([<type 'int'>]),
 'timestamp': set([<type 'str'>]),
 'uid': set([<type 'int'>]),
 'user': set([<type 'int'>, <type 'str'>]),
 'version': set([<type 'int'>])}
Auditing way tag attribute types
{'k': set([<type 'str'>]),
 'v': set([<type 'float'>, <type 'int'>, <type 'str'>])}
Auditing way-node tag attribute types
{'ref': set([<type 'int'>])}
```

b) Auditing coordinates

To verify that the coordinates were correct, first it was checked that the data type was float value considering the information previously presented. As the latitude and longitude both have a float type, there are no coordinates in another format.

Furthermore, I verified that the coordinates were in a reasonable range. For this a square area around London was defined. A function was built to verify that the coordinates in the OSM fall inside the area, otherwise they were stored in a dictionary for further analysis.

The code that was used is the following.

```
In [ ]: # Data structure used to store wrong coordinates
        coord_out_area = {}

        # A function to audit coordinates
        def audit_coordinates(coord_out_area, element_attributes):
            node_id = element_attributes['id']
            lati = float(element_attributes['lat'])
            longi = float(element_attributes['lon'])
            # Evaluates if the latitude and longitude fall outside the area of i
        nterest
            if not(lati > 51.4425602 and lati < 51.5785612) or not (longi > -0.2
        1698 and longi < 0.0164795):
                coord_out_area[node_id] = (lati,longi)
```

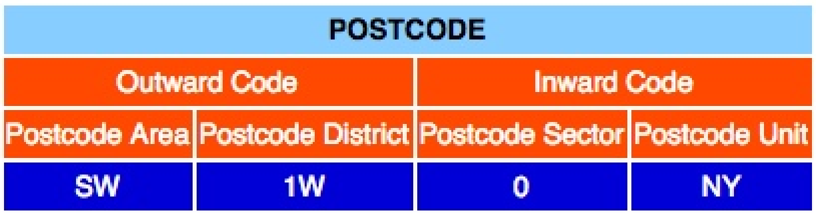
For this dataset all coordinates were valid. If I had defined a smaller rectangular area maybe focusing in a particular neighborhood of the City many values would have been out of the range.

c) Auditing postal codes

According to the Post codes in the United Kingdom (https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom#Listings_and_availability) article the syntax of the postal code in London can be described as follows:

- Alphanumeric
- Variable in length ranging from six to eight characters (including a space) long
- Each post code is divided into two parts separated by a single space:
    - **Outward code:** includes the postcode area *(one or two-letter)* and the postcode district *(one or two digits)*
    - **Inward code:** includes the postcode sector *(one number)* and the postcode unit *(two letters)*.
- Additionally, the post code area *for London* corresponds to a division of the city in EC (East Central), WC (West Central), N (North), E (East), SE (South East), SW (South West), W (West), and NW (North West).

For example:



POSTCODE			
Outward Code		Inward Code	
Postcode Area	Postcode District	Postcode Sector	Postcode Unit
SW	1W	0	NY

Therefore, I used Python's regular expressions to build the patterns finding out that some of the post codes were incomplete. Some just included the outward part (i.e. E1 or SW12), making it impossible to fix. Moreover, I was able to identify a special case pattern of recent post codes that the Royal Mail have authorized and that do not correspond 100% to the pattern previously described.

I used the following code:

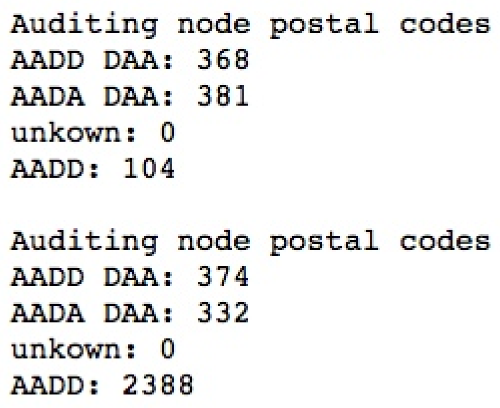
In [ ]:
```
# Regular Expressions
# AADD DDA pattern
postal_code_type_1_re = re.compile(r'([A-Z]{1,2}[1-9]{1,2}\s[0-9][A-Z]
{1,2})')
# AADA DAA pattern
postal_code_type_2_re = re.compile(r'([A-Z]{1,2}[1-9][A-Z]\s[0-9][A-Z]
{1,2})')
# AADD
postal_code_type_3_re = re.compile(r'[A-Z]{1,2}[1-9]{1,2}')

# Data structure
postal_code_types = defaultdict(set)
# Counter
counter_postal_code_types = {'AADD DAA': 0, 'AADA DAA': 0, 'AADD': 0, 'u
nkown': 0}

# Audit potal code function
def audit_postal_code(postal_code_types, postal_code):
    if postal_code_type_1_re.match(postal_code):
        postal_code_types['AADD DAA'].add(postal_code)
        counter_postal_code_types['AADD DAA'] += 1
    elif postal_code_type_2_re.match(postal_code):
        postal_code_types['AADA DAA'].add(postal_code)
        counter_postal_code_types['AADA DAA'] += 1
    elif postal_code_type_3_re.match(postal_code):
        postal_code_types['AADD'].add(postal_code)
        counter_postal_code_types['AADD'] += 1
    else:
        postal_code_types['unknown'].add(postal_code)
        counter_postal_code_types['unkown'] += 1
```

As a result, it was found that in nodes 749 postal codes were in the correct format and 104 were incomplete because they just had the outward code. Similarly, 706 postal codes in way elements were correct and 2388 were incomplete for the same reason.

Those postal codes that only contain the outward code are not possible to be fixed because we lack information to assign and inward code.

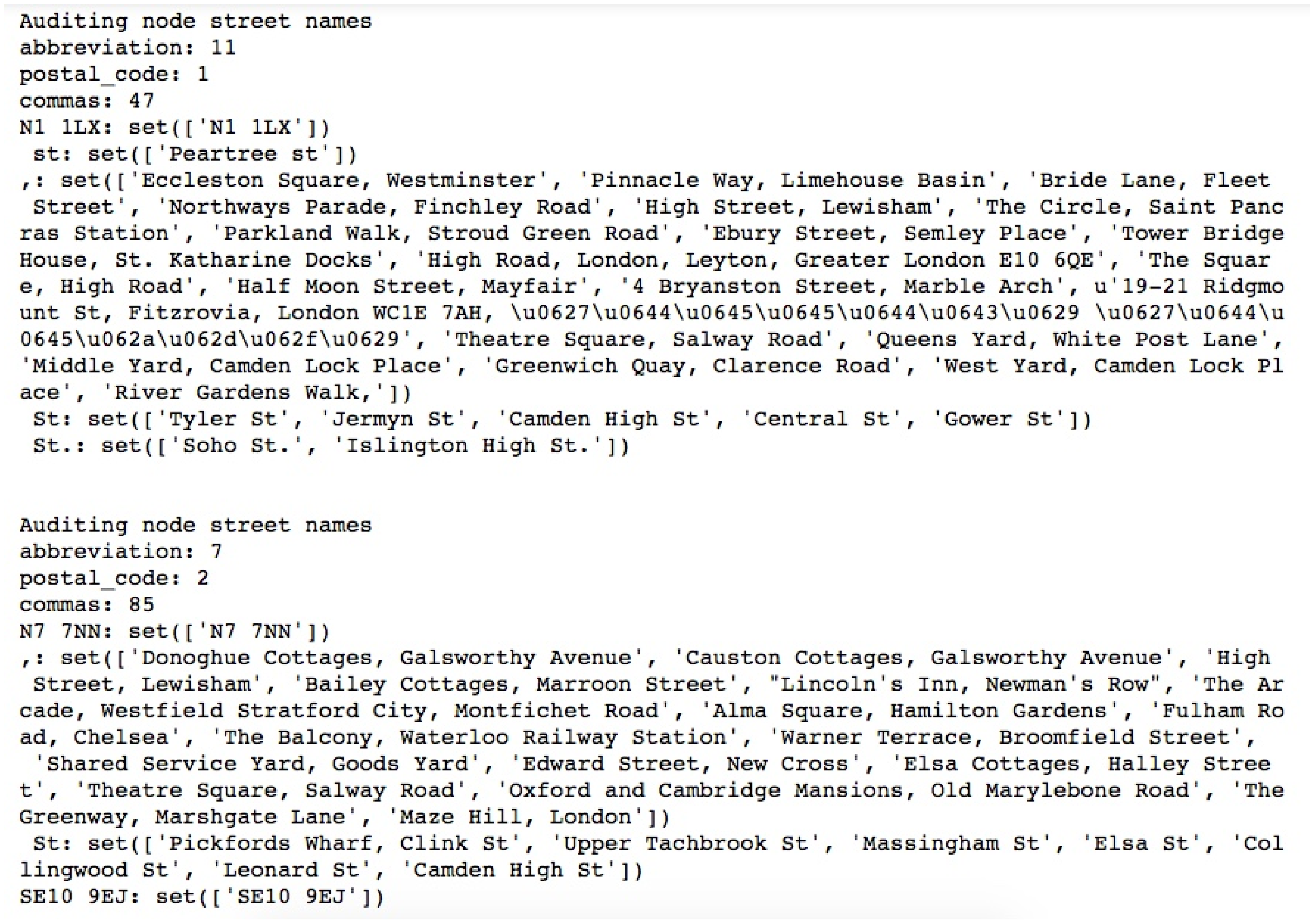

```
Auditing node postal codes
AADD DAA: 368
AADA DAA: 381
unkown: 0
AADD: 104

Auditing node postal codes
AADD DAA: 374
AADA DAA: 332
unkown: 0
AADD: 2388
```

d) Auditing street names

It could be said that the street values in the OSM have been previously cleaned as almost 99% of the names appear in the correct format. Meaning that there are no abbreviations for streets, avenues, roads, etc.

However, there is a 1% in both node and way elements with street names that contain abbreviations, commas and postal codes. The following image shows the result of street names audit.



```
Auditing node street names
abbreviation: 11
postal_code: 1
commas: 47
N1 1LX: set(['N1 1LX'])
 st: set(['Peartree st'])
,: set(['Eccleston Square, Westminster', 'Pinnacle Way, Limehouse Basin', 'Bride Lane, Fleet
 Street', 'Northways Parade, Finchley Road', 'High Street, Lewisham', 'The Circle, Saint Panc
ras Station', 'Parkland Walk, Stroud Green Road', 'Ebury Street, Semley Place', 'Tower Bridge
House, St. Katharine Docks', 'High Road, London, Leyton, Greater London E10 6QE', 'The Squar
e, High Road', 'Half Moon Street, Mayfair', '4 Bryanston Street, Marble Arch', u'19-21 Ridgmo
unt St, Fitzrovia, London WC1E 7AH, \u0627\u0644\u0645\u0645\u0644\u0643\u0629 \u0627\u0644\u
0645\u062a\u062d\u062f\u0629', 'Theatre Square, Salway Road', 'Queens Yard, White Post Lane',
'Middle Yard, Camden Lock Place', 'Greenwich Quay, Clarence Road', 'West Yard, Camden Lock Pl
ace', 'River Gardens Walk,'])
 St: set(['Tyler St', 'Jermyn St', 'Camden High St', 'Central St', 'Gower St'])
 St.: set(['Soho St.', 'Islington High St.'])


Auditing node street names
abbreviation: 7
postal_code: 2
commas: 85
N7 7NN: set(['N7 7NN'])
,: set(['Donoghue Cottages, Galsworthy Avenue', 'Causton Cottages, Galsworthy Avenue', 'High
 Street, Lewisham', 'Bailey Cottages, Marroon Street', "Lincoln's Inn, Newman's Row", 'The Ar
cade, Westfield Stratford City, Montfichet Road', 'Alma Square, Hamilton Gardens', 'Fulham Ro
ad, Chelsea', 'The Balcony, Waterloo Railway Station', 'Warner Terrace, Broomfield Street',
 'Shared Service Yard, Goods Yard', 'Edward Street, New Cross', 'Elsa Cottages, Halley Stree
t', 'Theatre Square, Salway Road', 'Oxford and Cambridge Mansions, Old Marylebone Road', 'The
Greenway, Marshgate Lane', 'Maze Hill, London'])
 St: set(['Pickfords Wharf, Clink St', 'Upper Tachbrook St', 'Massingham St', 'Elsa St', 'Col
lingwood St', 'Leonard St', 'Camden High St'])
SE10 9EJ: set(['SE10 9EJ'])
```

The code used for street auditing is the following.

In [ ]:
```
# Regular expressions
street_type1_re = re.compile(r'st', re.IGNORECASE)
street_type2_re = re.compile(r',', re.IGNORECASE)
postal_code_type_1_re = re.compile(r'([A-Z]{1,2}[1-9]{1,2}\s[0-9][A-Z]
{1,2})')
postal_code_type_2_re = re.compile(r'([A-Z]{1,2}[1-9][A-Z]\s[0-9][A-Z]
{1,2})')
postal_code_type_3_re = re.compile(r'[A-Z]{1,2}[1-9]{1,2}')
street_types = defaultdict(set)

# Counter
type_counter = {'abbreviation': 0, 'commas': 0, 'postal_code': 0}

# Function to audit streets
def audit_street(street_types, street_name):
    m1 = street_type1_re.search(street_name)
    m2 = street_type2_re.search(street_name)
    m3 = postal_code_type_1_re.search(street_name)
    m4 = postal_code_type_2_re.search(street_name)
    m5 = postal_code_type_3_re.search(street_name)
    if m1:
        street_type = m1.group()
        street_types[street_type].add(street_name)
        type_counter['abbreviation'] += 1
    elif m2:
        street_type = m2.group()
        street_types[street_type].add(street_name)
        type_counter['commas'] += 1
    elif m3:
        street_type = m3.group()
        street_types[street_type].add(street_name)
        type_counter['postal_code'] += 1
    elif m4:
        street_type = m4.group()
        street_types[street_type].add(street_name)
        type_counter['postal_code'] += 1
    elif m5:
        street_type = m5.group()
        street_types[street_type].add(street_name)
        type_counter['postal_code'] += 1
```

e) Auditing tube stations in the City of London

As a final exercise to verify the quality of the data, the topic of the tube stations in the City of London was chosen.

A query to the node tags was executed where key attribute equaled station and value attribute equaled subway. As a result, 21 records were found. As the records did not detail the name of the station, another query was made to add the latitude and longitude to the dataset so they could be further compared with Google Maps.

In [3]:
```
db = sqlite3.connect("city_of_london.db")
c = db.cursor()
query = ''' SELECT sub.id, sub.type, sub.key, sub.value, nodes.lat, node
s.lon, nodes.user
            FROM nodes JOIN (SELECT id, type, key, value
                                        FROM node_tags
                                        WHERE key="station" and value="subwa
y"
                                        ORDER BY key) as sub
            ON nodes.id = sub.id;'''

c.execute(query)
rows = c.fetchall()
db.close()


idk = []
typ = []
key = []
val = []
lat = []
lon = []
usr = []
for row in rows:
    idk.append(row[0])
    typ.append(row[1])
    key.append(row[2])
    val.append(row[3])
    lat.append(row[4])
    lon.append(row[5])
    usr.append(row[6])
print 'Results for subway stations to the City of London OSM'
pd.DataFrame({'id':idk, 'type': typ, 'key': key, 'value': val, 'latitud
e': lat, 'longitude': lon, 'user': usr})
```

```
Results for subway stations to the City of London OSM
```

Out[3]:

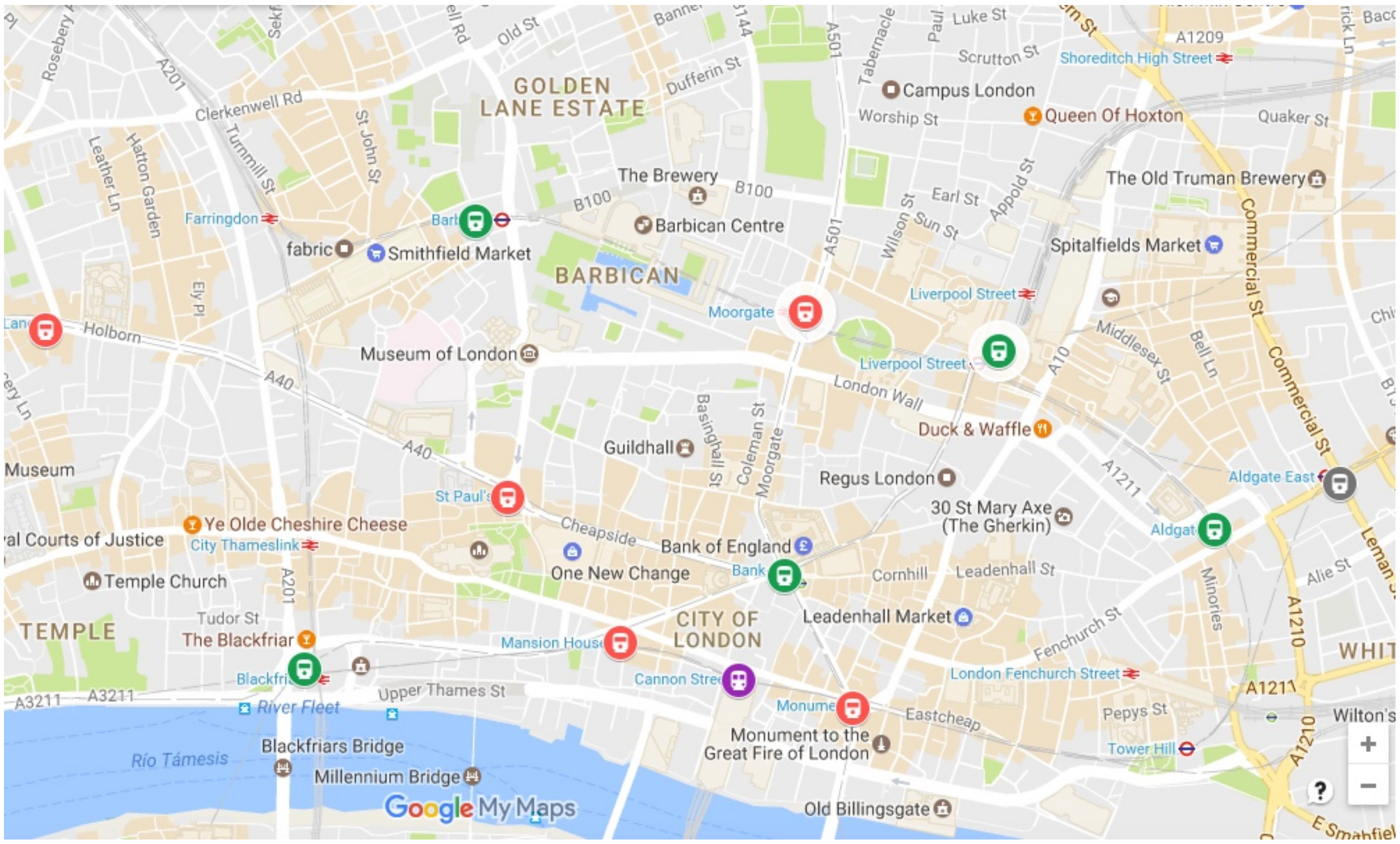
	id	key	latitude	longitude	type	user	value
0	26652919	station	51.4626925	-0.1147275	regular	lcmortensen	subway
1	186194628	station	51.5587178	-0.1078741	regular	jesus_army	subway
2	1264468143	station	51.5225862	-0.1567349	regular	bjohas	subway
3	1264482797	station	51.5157049	-0.1758402	regular	Med	subway
4	1389887370	station	51.5654371	-0.1349977	regular	jesus_army	subway
5	1637578440	station	51.5133798	-0.0889552	regular	jesus_army	subway
6	1670707904	station	51.5322944	-0.1059515	regular	jesus_army	subway
7	1670707917	station	51.5394276	-0.1426923	regular	Paul The Archivist	subway
8	1721881886	station	51.526845	-0.0248534	regular	jesus_army	subway
9	1741760056	station	51.5268726	-0.0248416	regular	jesus_army	subway
10	1807593699	station	51.5010936	-0.093406	regular	Med	subway
11	1825191028	station	51.515128	-0.0718502	regular	trigpoint	subway
12	2337594752	station	51.5272716	-0.0550277	regular	bjohas	subway
13	3076573910	station	51.520169	-0.0984809	regular	bjohas	subway
14	3076573913	station	51.514299	-0.149002	regular	bjohas	subway
15	3370671078	station	51.5112839	-0.1284197	regular	Jindo	subway
16	3637436407	station	51.5176805	-0.0823701	regular	bjohas	subway
17	3637449659	station	51.5113812	-0.0903632	regular	bjohas	subway
18	3638795618	station	51.5030015	-0.1139812	regular	bjohas	subway
19	3763522921	station	51.5115854	-0.1037671	regular	bjohas	subway
20	3788735981	station	51.5142477	-0.0757186	regular	bjohas	subway

In order to verify the accuracy and completeness of this subset the data was compared against Google Maps results.

From the 21 results of this query only 5 corresponded to City of London subway stations. Only one of them was missclassified because it was a train station, the rest 15 records belong to stations outside the City of London.

It is worth to point out that the dataset misses to include 5 tube stations.

Therefore, for this sort of query the data is incomplete, and there are some issues with classification.



A way to improve the data is to revalidate data points. It is possible that the missing tube stations were classified under different tag keys.

2. Description of cleaning plan and process

I decided to clean the postal codes and the street names.

a) Cleaning postal codes

To clean the postal codes, the regular expressions in the audit process were used. A function was defined to evaluate four patterns:

- If the postal code fulfilled the **AADD DAA** or **AADA DAA** patterns, the function returned the postal code value as it was correct.
- If the postal code had this pattern **AADD**, the function returned a string saying that the **"*postal code was incomplete*"**.
- For **any other case** in postal code the function returned a string saying **"*not a postal code*"**.

```
In [ ]: # Function that updates postal code value
        def update_postal_code(postal_code):
            # Pattern AADD DAA A = letter A-Z ; D = digit 0-9
            if postal_code_type_1_re.match(postal_code):
                return postal_code
            # Pattern AADA DAA
            elif postal_code_type_2_re.match(postal_code):
                return postal_code
            # Pattern AADD
            elif postal_code_type_3_re.match(postal_code):
                return 'Postal code incomplete'
            # Any other string different than a postal code
            else:
                return 'Not a postal code'
```

The following query shows results for the cleaned postal codes

In [4]:
```
# SQL Query for Postal Codes
db = sqlite3.connect("city_of_london.db")
c = db.cursor()
query = '''select distinct type, key, value
           from way_tags
           where key = "postal_code"
           order by value desc
           limit 10;'''
c.execute(query)
rows = c.fetchall()

print "Postal codes in Way tags:"
for row in rows:
    print row
    #print row[0], ": ", row[1]

db.close()
```

```
Postal codes in Way tags:
(u'regular', u'postal_code', u'WC2V 4ET')
(u'regular', u'postal_code', u'WC2R 3JF')
(u'regular', u'postal_code', u'WC2R 2PG')
(u'regular', u'postal_code', u'WC2R 1JA')
(u'regular', u'postal_code', u'WC2R 1DA')
(u'regular', u'postal_code', u'WC2R 1AY')
(u'regular', u'postal_code', u'WC2R 1AP')
(u'regular', u'postal_code', u'WC2R 0NH')
(u'regular', u'postal_code', u'WC2R 0HS')
(u'regular', u'postal_code', u'WC2R 0ET')
```

b) Cleaning street names

To clean the streets, the regular expressions in the audit process were also used. A function to evaluate four cases was defined.

- The **first case** substituted all abbreviations **St or St located at the end of the string** for the complete name Street.
- The **second case** looked for **values that instead of a street name had an address**, such as: "*Warner Terrace, Broomfield Street*". The function splitted the string into a list using the comma as a separator. It evaluated each of the items in the list created to find an expected value such as Street, Avenue or Road. If found, it returned that item in the list, in our example: "*Broomfield Street*". If the expected value was not found it returned the original street value.
- The **third case** removed all those street values containing a **postal code**.
- The **fourth case** just returned all the **correct street names**.

In [ ]:
```
# Function that updates street value
def update_street(street_name):
    #Case 1: Abbreviations
    if street_type1_re.search(street_name):
        street_name = re.sub( street_type1_re , ' Street',street_name)
        return  street_name
    #Case 2: Complete address
    elif street_type2_re.search(street_name):
        street_list = street_name.split(',')
        for street_item in street_list:
            for expected_item in expected:
                if expected_item in street_item:
                    street_name = street_item.strip()
        return street_name
    #case 3: Postal code
    elif postal_code_re.search(street_name):
        return ''
    #case 4: Any normal case
    else:
        return street_name
```

The following SQL query shows results for the cleaned street names.

In [5]:
```
# SQL Query for Street Name
db = sqlite3.connect("city_of_london.db")
c = db.cursor()
query = '''select distinct type, key, value
           from way_tags
           where key = "street"
           order by value
           limit 10;'''
c.execute(query)
rows = c.fetchall()

print "Street names in Way tags:"
for row in rows:
    print row
    #print row[0], ": ", row[1]

db.close()
```

```
Street names in Way tags:
(u'addr', u'street', u'')
(u'addr', u'street', u'1 Portsmouth Street')
(u'addr', u'street', u'150 Vicarage Road')
(u'addr', u'street', u'17 Old Court Place')
(u'addr', u'street', u'18-22 Damien Street')
(u'addr', u'street', u'1st Avenue')
(u'addr', u'street', u'2 Blackheath Vale')
(u'addr', u'street', u'263')
(u'addr', u'street', u'3rd Avenue')
(u'addr', u'street', u'400 Oxford Street')
```

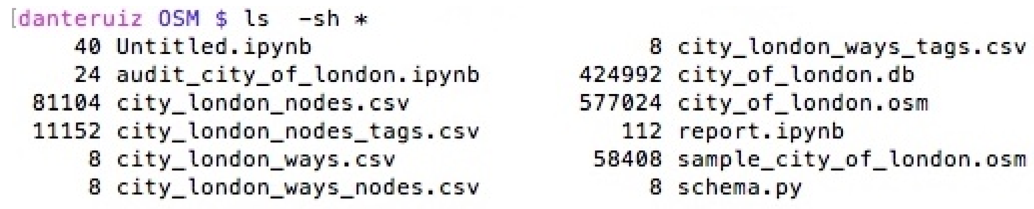
3. Data overview

This section presents statistics about the dataset.

3.1 Size of the file

The following figure shows the files used in this analysis and their size in memory.

- The open street map file of the City of London is: **563.5 MB**
- The SQLite database containing the tables of the parsed OSM file is: **415 mb**



```
[danteruiz OSM $ ls  -sh *
     40 Untitled.ipynb                      8 city_london_ways_tags.csv
     24 audit_city_of_london.ipynb     424992 city_of_london.db
  81104 city_london_nodes.csv          577024 city_of_london.osm
  11152 city_london_nodes_tags.csv        112 report.ipynb
      8 city_london_ways.csv            58408 sample_city_of_london.osm
      8 city_london_ways_nodes.csv          8 schema.py
```

3.2 Number of unique users

Considering the nodes and ways tables, the number of **unique users 3,111** that have contributed to the OSM database for this metropolitan.

In [6]:
```
db = sqlite3.connect("city_of_london.db")
c = db.cursor()
query = ''' SELECT COUNT(uid) FROM
                (SELECT uid FROM nodes
                 UNION
                 SELECT uid FROM ways) as TOTAL_USERS;'''
c.execute(query)
rows = c.fetchall()

print "Unique useres: " + str(rows[0][0])
db.close()
```

```
Unique useres: 3111
```

3.3 Number of nodes and ways

The number of node and way entries are **1,135,372** and **213,993** respectively.

```
In [7]: db = sqlite3.connect("city_of_london.db")
        c = db.cursor()
        query = ''' SELECT COUNT(id)
                    FROM nodes;'''
        c.execute(query)
        rows = c.fetchall()

        print "Number of nodes in the Nodes table: " + str(rows[0][0])
        db.close()
```

```
Number of nodes in the Nodes table: 1135372
```

```
In [10]: db = sqlite3.connect("city_of_london.db")
         c = db.cursor()
         query = ''' SELECT COUNT(id)
                     FROM ways;'''
         c.execute(query)
         rows = c.fetchall()

         print "Number of ways in the Ways table: " + str(rows[0][0])
         db.close()
```

```
Number of ways in the Ways table: 213993
```

3.4 number of chosen type of nodes, like cafes, shops etc.

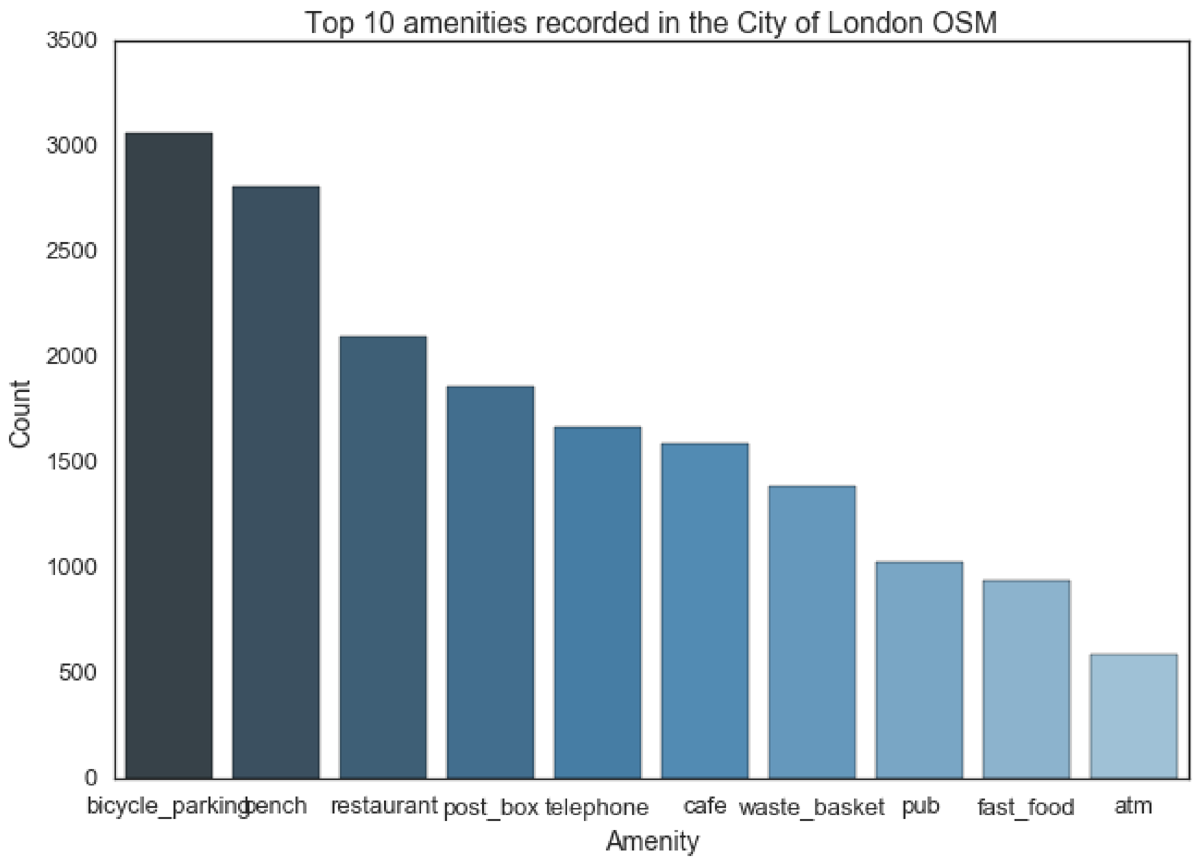
A query to get the **top 10 amenities** recorded in the OSM shows that ***bicycle parking, benches, restaurants, post-boxes, telephones, cafes, waste-baskets, pubs, fast food restaurants and atms dominate***. However, the distribution of these is uneven as in this top 10 bicycle parkings account to 3067 entries and atms to 589.

```
In [8]: db = sqlite3.connect("city_of_london.db")
        c = db.cursor()
        query = ''' SELECT value, COUNT(value) as total
                    FROM node_tags
                    WHERE key = "amenity"
                    GROUP BY value
                    ORDER BY total DESC
                    LIMIT 10;'''
        c.execute(query)
        rows = c.fetchall()

        total = []
        amenity = []
        for row in rows:
            amenity.append(row[0].encode('UTF-8'))
            total.append(row[1])
        db.close()
```

In [9]:
```
%pylab inline
amenity_top10 = pd.DataFrame({'amenity':amenity, 'total': total})
sns.set(style="white", context="talk")
sns.barplot('amenity','total', data = amenity_top10, palette = ("Blues_
d"))
plt.title('Top 10 amenities recorded in the City of London OSM')
plt.ylabel('Count')
plt.xlabel('Amenity')
plt.show()
```

```
Populating the interactive namespace from numpy and matplotlib
```



4. Additional ideas

In this section some ideas that are worth considering and some ideas of how to improve the quality of the data are discussed.

The behaviour of users contributing to build the City of London OSM

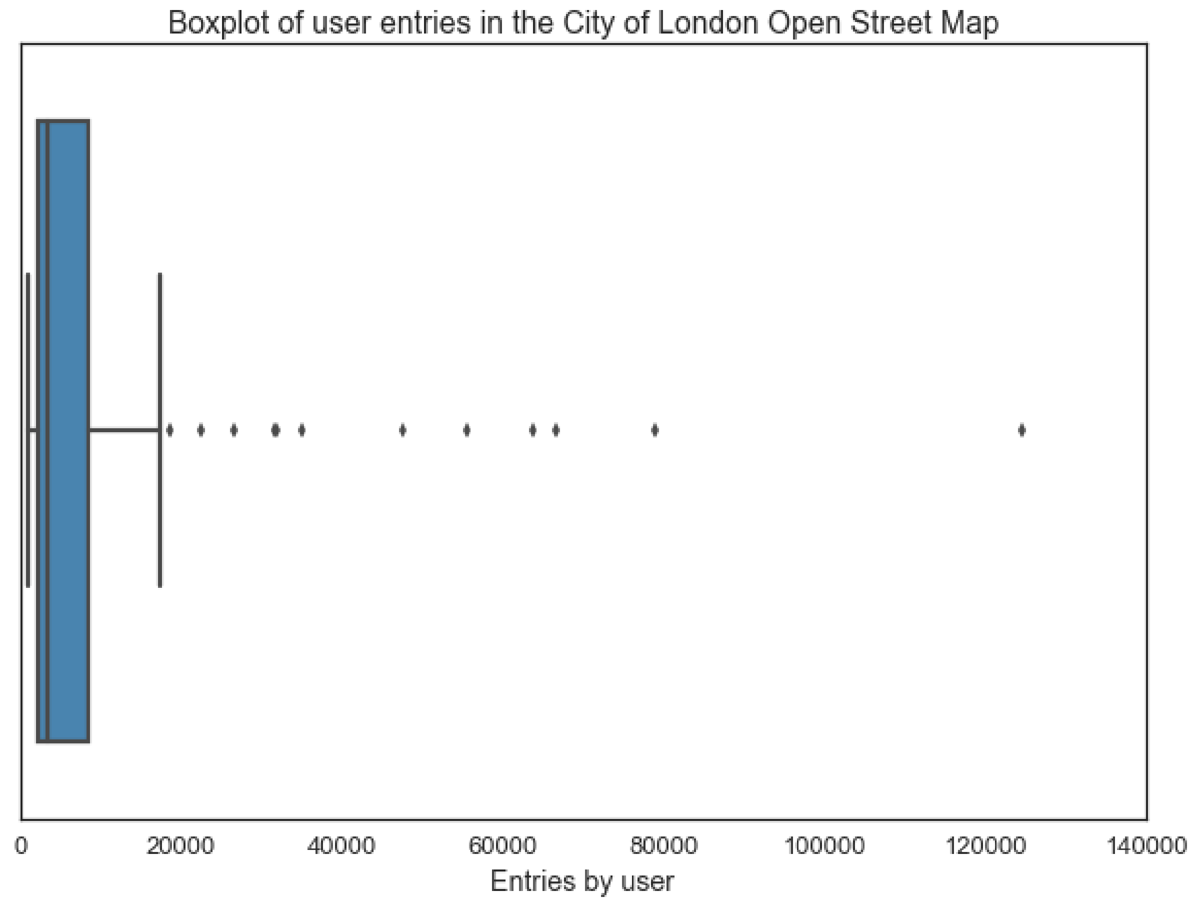
The City of London OSM has 3111 unique users contributing to update this map. Half of the users had made maximum 4 contributions. However, if we just consider the top 100 users we can see that the minimum number of contributions has been 1037 entries and the maximum is of 124,476 entries. Therefore, it can seen that most of the features contained in this OSM is concentrated in very few users, some with hundred thousand entries (See stats and boxplot below).

In [10]:
```
db = sqlite3.connect("city_of_london.db")
c = db.cursor()
query = ''' SELECT nodes.user, count(*) as total
            FROM nodes
            GROUP BY nodes.user
            ORDER BY total DESC
            LIMIT 100;'''
c.execute(query)
rows = c.fetchall()
db.close()

entries = []
for row in rows:
    entries.append(int(row[1]))
```

In [11]:
```
%pylab inline
entries = pd.Series(entries)
print entries.describe()
sns.set(style="white", context="talk")
sns.boxplot(entries, palette = ("Blues_d"))
plt.title("Boxplot of user entries in the City of London Open Street Ma
p")
plt.xlabel("Entries by user")
plt.show()
```

```
Populating the interactive namespace from numpy and matplotlib
count       100.000000
mean      10248.260000
std       18359.049122
min        1037.000000
25%        2231.500000
50%        3466.500000
75%        8598.250000
max      124476.000000
dtype: float64
```



Considering that most entries in the OSM dataset are done by a small group of users, the quality of the data relies on the criteria that this group of people used. Ideally, the more contributions from different users the greater possibility to minimize the risk of populating the map with wrong or biased data.

User information and validity

One of the limitations of the Open Street Map is that the only information about the identity of the user is its name. This makes it difficult to assess the accuracy of the data because we still do not know if the data was entered by an independent user, a government, company or organization. Particularly when the user entered thousands of points.

The following query shows the results for the users that entered the information of the bicycle parking spaces, the amenity with more points in the City of London OSM. It can be seen that all points were entered by user "mcld". This raises questions such as how accurate is his data, and what criteria he used.

```
In [12]: db = sqlite3.connect("city_of_london.db")
         c = db.cursor()
         query = ''' SELECT nodes.user, subquery.key, count(nodes.user)
                     FROM nodes JOIN (SELECT *
                                      FROM node_tags
                                      WHERE value = "bicycle_parking") as subquery
                     ON nodes.id = subquery.id
                     GROUP BY subquery.key;'''
         c.execute(query)
         rows = c.fetchall()

         print "Users that added the bicycle parkings to the OSM"
         print rows
         db.close()
```

```
Users that added the bicycle parkings to the OSM
[(u'mcld', u'amenity', 3067), (u'JeanFred', u'transportation', 2)]
```

Verification data

It would be useful to add a field for each node and tag that could contain information about who and how many times the data has been verified by other users. This would help to increase the reliability or the validity of the data.

Form data validation

Some of the mistakes found in street names and postal codes have to do with the user entering information where it doesn't. Ensuring that the machine verifies what the user is entering will improve considerably the quality of the data.

I also reckon that this dataset has already been cleaned, as there were not many format mistakes as were expected.

Accuracy of the dataset

To obtain this OSM for the City of London, the metropolitan dataset already constructed in the Open Street Map website was used. It was a surprise taht the area defined for the City of London is larger than the City of London Borough that was expected and smaller to the metropolitan area of London. In this sense, the dataset is incomplete if someone wants to make an analysis of the metropolitan area of London, and too big if the user wants to analyze the City of London Borough. Maybe this has to be fixed or explained before downloading the data.

Conclusion

In this case study Python and SQLite were used to wrangle the City of London Open Street Map. As this dataset contains lots of information the data types, node coordinates, postal code format, street names and subway stations were chosen to assess the quality of the dataset.

As a result of the auditing process it can be inferred that the dataset has previously been cleaned, as the format of the street names and the format of the postal codes were 99% correct. For the resting 1% the case study suggests a way to audit and clean the street names and postal codes that are in the incorrect format. Bearing this in mind, the quality of the data in terms of validity and quality is good.

Another, exercise was to audit the coordinates which all are in the correct format. As to how accurate are these points the exercise of auditing the subway stations showed that for a sample of points their location are correct. However, there are subway station points that remain to be added, and some that need to be reclassified. Therefore, the completeness of the dataset remains to be a data quality issue. Of course this result might or might not be a concern depending on the purpose for which the data wants to be used.

It also was pointed out that the contribution of users to populate the dataset does not have a normal distribution pattern. In fact, there are only few users that account to more than 10,000 thousand contributions to the map. While this can be positive in terms of the amount of information that the OSM map might contain, it also raises the question as how accurate are the criteria that these users apply to map points. Furthermore, as there is no way to know more about the users, it is also difficult to assess their authority when it comes to topics such as the localization of public services and public infrastructure.

Bearing these in mind, it can be concluded that the OSM for the City of London is valuable information, with an appropriate degree of data quality. However, data wrangling plays a key role to assess how accurate, valid, complete, consistent and uniform it is, for the purpose for which the data wants to be used.

Appendix SQL

For this exercise the CSV files that were produced with the data wrangling coded were uploaded to a SQLite database. This is the SQL code was used to create and populate the database.

In [ ]:
```
sqlite3 city_of_london.db
.mode csv
.import city_of_london_nodes.csv nodes
.import city_of_london_nodes_tags.csv node_tags
.import city_of_london_ways.csv ways
.import city_of_london_ways_tags.csv way_tags
.import city_of_london_ways_nodes.csv way_nodes
```