# Open Street Map Data Cleaning and SQL Import

When I moved to Philadelphia in 2003 to attend UPenn, I had spent the previous 18 years growing up in rural Northern Nevada, where my family has lived since just before 1900. As Philly was my entry in the wider, urban world of the East Coast, the town, its food, its art, its history and culture will always hold special charm.

This project scrapes an Open Street Map (OSM) XML file and imports the data to a SQL databse that we can query to explore the map elements in the Philadelphia metro area. The first link below indicates the boundaries of the XML file, and to save time, the XML file was downloaded from mapzen from the second. The OSM XML for the Philly metro area tips the scales at 594 MB.

Philadlphia, PA - City of Brotherly Love

- https://www.openstreetmap.org/relation/188022
- https://mapzen.com/data/metro-extracts/metro/philadelphia_pennsylvania/

## Dataset Overview

There are over 2.8 million node tags, over 261 thousand way tags and only about 4000 relation tags. There are also over 1.7 million tags nested below node, way and relation elements, though only 7433 node tags and 9698 way tags with addr:street attributes.

### Most Common Node Tag Types

```
('created_by', 198001), ('highway', 15142), ('name', 12695),
('amenity', 7866), ('addr:street', 7433), ('addr:housenumber',
7284), ('ele', 5987), ('addr:city', 5686), ('gnis:feature_id',
4406), ('power', 4014), ('addr:state', 3477), ('addr:postcode',
3354), ('gnis:created', 3304), ('gnis:county_id', 3183),
('gnis:state_id', 3180), ('source', 3009), ('crossing', 2466),
('railway', 2130), ('shop', 1866), ('building', 1831), ('place',
1630), ('is_in', 1592), ('gnis:ST_num', 1580), ('gnis:County',
1580), ('gnis:County_num', 1580), ('gnis:ST_alpha', 1580),
('gnis:id', 1579), ('gnis:Class', 1579), ('import_uuid', 1578)
```

### Number of Nodes, Ways and Associated Tags

```
sqlite> SELECT COUNT(*) FROM Nodes
2811847
sqlite> SELECT COUNT(*) FROM Ways;
```

```
261503
sqlite> SELECT COUNT(*) FROM Way_Tags;
1369705
sqlite> SELECT COUNT(*) FROM Node_Tags;
334095
sqlite> SELECT COUNT(*) FROM Way_Nodes;
3387554
```

### Unique Users

```
sqlite> SELECT COUNT(DISTINCT(i.uid))
   ...> FROM (SELECT uid FROM Nodes UNION ALL SELECT uid FROM Ways) i;
1822
```

### Top Ten Users

```
SELECT nw.user, COUNT(*) as num
FROM (SELECT user FROM Nodes UNION ALL SELECT user FROM Ways) nw
GROUP BY nw.user
ORDER BY num DESC
LIMIT 10;
```

```
dchiles|799894
woodpeck_fixbot|571834
NJDataUploads|297483
kylegiusti|106160
Matt1993|92378
choess|86123
crystalwalrein|77304
Louise Belcher|65258
eugenebata|62532
bot-mode|44946
```

### Social Services

```
SELECT DISTINCT Node_Tags.value, COUNT(*) num
FROM Node_Tags
WHERE Node_Tags.key = 'social_facility'
GROUP BY  Node_Tags.value
ORDER BY num DESC;
```

```
food_bank|272
soup_kitchen|3
```

```
assisted_living|1
food_pantry|1
group_home|1
```

### Top 20 Shop Types

```
SELECT DISTINCT Node_Tags.value, COUNT(*) c
FROM Node_Tags
WHERE Node_Tags.key = 'shop'
GROUP BY  Node_Tags.value
ORDER BY c DESC
LIMIT 20;
```

```
convenience|266
supermarket|157
hairdresser|144
clothes|137
car_repair|79
farm|72
alcohol|68
beauty|54
furniture|49
bakery|48
dry_cleaning|36
department_store|35
doityourself|35
laundry|32
books|29
car|29
jewelry|25
mobile_phone|24
bicycle|23
gift|23
```

### Power Infrastructure

```
SELECT DISTINCT Node_Tags.value, COUNT(*) num
FROM Node_Tags
WHERE Node_Tags.key = 'power'
GROUP BY  Node_Tags.value
ORDER BY num DESC;
```

```
tower|3615
pole|373
```

```
generator|17
station|4
transformer|4
sub_station|1
```

### Top Zipcodes

```
SELECT tags.value, COUNT(*) as count
FROM (SELECT * FROM Node_Tags
      UNION ALL
      SELECT * FROM Way_Tags) tags
WHERE tags.key='postcode'
GROUP BY tags.value
ORDER BY count DESC
LIMIT 15;

Top zipcode is in NJ...
08068|2091
19061|1396
19106|394
19130|391
08043|358
19103|344
19107|308
19104|249
08003|196
19128|181
19428|174
19102|124
19123|124
19147|119
18914|117
19087|107
18901|105
19146|105
18974|100
08054|93
19072|66
19027|51
```

### Top Cities

```
SELECT tags.value, COUNT(*) as count
FROM (SELECT * FROM Node_Tags UNION ALL
```

```
      SELECT * FROM Way_Tags) tags
WHERE tags.key LIKE '%city'
GROUP BY tags.value
ORDER BY count DESC;

Philadelphia|3523
Pemberton|2092
Trenton city|1659
Lawrence twp|1157
Upper Chichester;Marcus Hook;Chichester|967
Boothwyn|844
Marcus Hook|374
Voorhees|211
Cherry Hill|207
Conshohocken|173
Voorhees Township|146
Ewing twp|137
Hopewell twp|121
Chalfont|119
Trenton|111
Ogden|104
1|102
Wayne|100
Doylestown|98
Linwood|95
Mount Laurel|92
Wilmington|87
New Jesery|84
Warminster|83
Lower Chichester;Marcus Hook;Chichester|74
```

### Top Ten Ammenities

```
SELECT value, COUNT(*) as num
FROM Node_Tags
WHERE key='amenity'
GROUP BY value
ORDER BY num DESC
LIMIT 10;

school|1665
restaurant|904
place_of_worship|699
fire_station|457
fast_food|377
```

```
social_facility|280
parking|243
bench|235
bank|217
cafe|210
```

### Religious Institutions

```
SELECT Node_Tags.value, COUNT(*) as num
FROM Node_Tags
    JOIN (SELECT DISTINCT(id) FROM Node_Tags WHERE value='place_of_worship') i
    ON Node_Tags.id=i.id
WHERE Node_Tags.key='religion'
GROUP BY Node_Tags.value
ORDER BY num DESC
LIMIT 5;
```

```
christian|638
jewish|33
muslim|4
buddhist|2
Baptist|1
```

### Total Restaurants

```
SELECT COUNT(*)
FROM Node_Tags
    JOIN (SELECT DISTINCT(id) FROM Node_Tags WHERE value='restaurant') i
    ON Node_Tags.id=i.id
WHERE Node_Tags.key='cuisine';
```

```
482
```

### Number of Different Kinds of Restaurants

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

```
pizza|102
italian|53
chinese|46
mexican|37
american|35
sandwich|23
burger|15
asian|14
japanese|12
indian|11
greek|7
spanish|6
steak_house|6
vietnamese|6
chicken|5
```

### Plentiful Wooder Ice

```
SELECT * FROM Node_tags WHERE value LIKE '%water ice%';
```

```
288055176|name|Rita's Water Ice|regular|27400
288055176|cuisine|Water Ice|regular|27402
796858057|name|Rita's Water Ice|regular|265005
975303670|name|Rita's Water Ice|regular|266273
1129804118|name|Petrucci's Ice Cream and Water Ice|regular|273989
1687402740|housename|Rita's Water Ice|addr|282285
1740624913|name|Cabana Water Ice|regular|283457
2002984943|housename|Rita's Water Ice|addr|285015
2343299340|name|Ritas Water Ice|regular|289362
2699346738|name|Rita's Water Ice|regular|294372
2795611956|name|Rita's Water Ice|regular|298445
2818610758|name|Rita's Water Ice|regular|298747
2876251700|name|John's Water Ice|regular|300036
3342901422|name|Rita's Water Ice|regular|302492
3454225867|name|Georges Water Ice|regular|305664
4307284690|name|Rita's Water Ice|regular|316506
4313660216|name|Gracie's Water Ice|regular|316561
4331629329|name|Rita's Water Ice|regular|317801
4352921245|name|Georgio's Water Ice|regular|326844
4358135189|name|Rita's water ice|regular|327638
4358135189|en|Rita's water ice|name|327639
```

### All Address Fields

```
SELECT key, COUNT(*) c
FROM Node_Tags
WHERE key IN ('city', 'country', 'full', 'housename', 'housenumber', 'postcode', 'street',
GROUP BY key
ORDER BY c DESC;
```

```
street|7433
housenumber|7315
city|6639
state|3500
postcode|3359
country|1174
housename|214
full|29
```

## Data Problems Encountered

### addr:street Attributes Contain Values for Other Fields

Several addr:street attributes included full addresses, with state, zipcode, phone, house, suite and unit numbers. The first cleaning function, 'clean_streets()' scans values in all addr:street attributes and uses regular expressions to match any values that should be in other fields.

```
'1 Brookline BlvdHavertown, PA 19083(610) 446-1234'
'200 Manor Ave. Langhorne, PA 19047'
'2245 E. Lincoln Hwy, Langhorne, PA 19047',
'2275 E Lincoln Hwy, Langhorne, PA 19047',
'2300  East Lincoln Highway, Pennsylvania 19047'},
'East Trenton Avenue Morrisville, PA 19067'
```

If these values are found, the function inserts a new tag in the parent element with the appropriate attribute and value, unless a tag with that value already exists. For example:

```
phone = re.compile(r'\(?\d{3}\)?[-\.\s]??\d{3}[-\.\s]??\d{4}|\d{3}[-\.\s]??\d{4}')
phones = soup.find_all("tag", attrs={"k": "phone"})

call_me = phone.search(tag['v'])
if call_me:
  v_val = call_me.group()
  new_tag = soup.new_tag("tag", k="phone", v='{}'.format(v_val))
    if new_tag not in phones:
      tag.insert_after(new_tag)
```

```
<node changeset="34353963" id="1483624883" lat="39.9787384" lon="-75.3038692" timestamp="201
  <tag k="name" v="Kettle"/>
  <tag k="amenity" v="restaurant"/>
  <tag k="cuisine" v="Diner"/>
  <tag k="addr:street" v="Brookline Boulevard"/>
  <tag k="addr:city" v="Havertown"/>
  <tag k="addr:state" v="Pennsylvania"/>
  <tag k="phone" v="(610) 446-1234"/>
  <tag k="addr:postcode" v="19083"/>
  <tag k="addr:housenumber" v="1"/>
 </node>
```

### Garden Variety Typos and Abbreviations

The 'filter_words()' function scans each word in the addr:street field and replaces mispellings and abbreviations with a mapping.

#### Street name typos

```
'Atreet': {'Arch Atreet'}
'PIke': {'Princeton PIke'},
'Sreet': {'Bridge Sreet'},
'Sstreet': {'South 9th Sstreet'},
'Steet': {'South 18th Steet'},
```

#### Over abbreviated addresses

```
'N Olden Ave',
'S. 41st',
'Shannondell Blvd',
'Portsmouth Ct',
'Deerpath Dr',
'Anderson Rd',
'S 19th St'
```

#### Improper Case

```
'ROAD': {'DAVISVILLE ROAD', 'TERWOOD ROAD', 'TOWNSHIP LINE ROAD'},
```

### Which Streets Are in Philly, and Which Are Not?

The XML includes elements from a square bounding box around Philadelphia, so some streets are from nearby towns in PA, like Havertown and Langhorne as well as some from New Jersey, just accross the Delaware River. With a list of

clean, unique street names in Philadelphia, we can clean and validate the data with an approximate (fuzzy) string matching algorithm.

One surprising discovery is that no clean, canonical list of the streets in Philadelphia is easily accessbile. In fact, searching for this on the Philadelphia government webpage raised an immediate red flag.

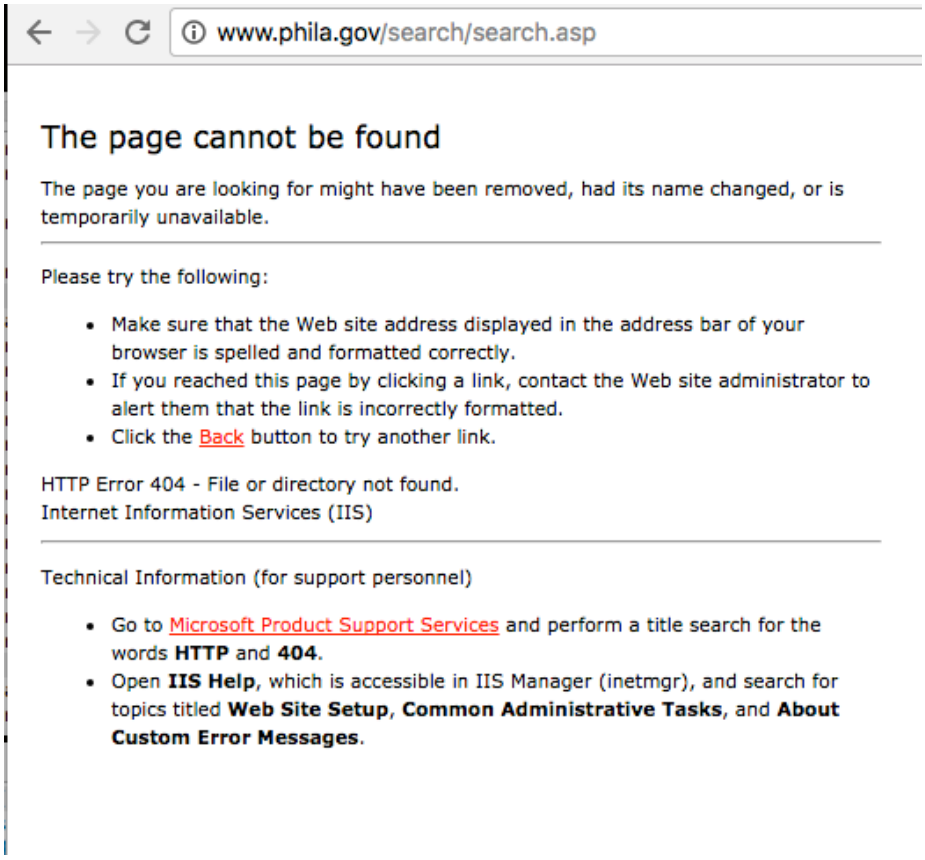


Figure 1: Philly.gov #fail

Although Philly's Open Data site did have two csv files with streetnames available, neither was cleaned or comprehensive enough to use as a canoncial list for fuzzy string matching.

- Street Name Alias
- Street Place Names

Fortunately, the US Census' Topologically Integrated Geographic Encoding and Referencing (TIGER) data imported into the 'Way' elements includes both "tiger:name_base" attributes as well as "name" attributes with full street names.

```
<way changeset="37014829" id="43117631" timestamp="2016-02-05T08:40:51Z" uid="3057995" user=
  <nd ref="110421617"/>
  <nd ref="2906080683"/>
  <tag k="name" v="West Girard Avenue"/>
  <tag k="layer" v="1"/>
  <tag k="bridge" v="yes"/>
  <tag k="highway" v="secondary"/>
  <tag k="tiger:cfcc" v="A21"/>
  <tag k="ref:penndot" v="SR2008"/>
  <tag k="tiger:zip_left" v="19131"/>
  <tag k="tiger:name_base" v="Girard"/>
  <tag k="tiger:name_type" v="Ave"/>
  <tag k="tiger:zip_right" v="19104"/>
  <tag k="tiger:name_base_1" v="United States Highway 30"/>
  <tag k="old_ref_legislative" v="67301"/>
  <tag k="tiger:name_direction_prefix" v="W"/>
 </way>
```

The downside is that sometimes a 'Way' element's 'name' attribute is a street name, and other times, it's a location name.

```
<way changeset="10677130" id="150255661" timestamp="2012-02-13T20:23:59Z" uid="594684" user=
  <nd ref="1631799891"/>
  <nd ref="1631799894"/>
  <nd ref="1631799920"/>
  <nd ref="1631799922"/>
  <nd ref="1631799962"/>
  <nd ref="1631799959"/>
  <nd ref="1631799945"/>
  <nd ref="1631799914"/>
  <nd ref="1631799891"/>
  <tag k="name" v="Red Lion Diner"/>
  <tag k="building" v="restaurant"/>
  <tag k="addr:street" v="US and US Streets"/>
  <tag k="addr:postcode" v="08088"/>
  <tag k="addr:housename" v="Red Lion Diner"/>
 </way>
```

Upon closer examination, the 'tiger:name_base' attribute contained 15,284 elements, and many contained typos and were obviously not streets.

```
u'Norfolk Southern Railway:Pennsylvania Railroad', u'Belmont;Green',
u'United States Highway 1; Lincoln', u'Mantua;Harrison', u'Reading
Railroad:Septa Railroad', u'Baltimore and Ohio Railroad:Norfolk
Southern Railway', u'Franklin:Hampton', u'State Route 68; State
Route 68; State Route 68A; State Route 68; State Route 68',
u'United States Highway 206;Old York', u'Perry; Lincoln',
u'Norfolk Southern Railway; Csx Railway; Conrail Railroad',
```

```
u'Township Line;Big Oak', u'Market:United States Highway 13
(Bus)', u'Coursey; College', u'I-295:I-76', u'Early; Davis', u'of
the Arts', u'\x7f\x7fBeech', u'Bridge; Main', Spring:Pond View',
u'Cypress:Longacre', u'Delmar;84th', u'New Jersey Transit:Penn
Central Railroad; Conrail Railroad'
```

The 'Way' element 'name' attributes, however, once filtered to exclude non-street names, was comprehensive and clean enough to use as a canonical reference list for appoximate string matching. Though not perfect, with 31,489 names, chances of finding correctly spelled matches are high, and the cases tested from this dataset worked well.

As building an approximate string matching algorithm from scratch would be time consuming, the open source FuzzyWuzzy approximate string matching module came in handy. The module is based on Levenshtein distance, and when given a string and list of strings as input, outputs the highest ranked match between the string provided and the choices in the list.

The 'fix_street_names()' function applies a fuzzy string matching algorithm to the edge case strings that the conventional word mapping doesn't easily fix.

For example:

```
from fuzzywuzzy import process

    for tag in street_tags:
        m = street_type_re.search(tag['v'])
        street_end = m.group()
        intersect = intersection.search(tag['v'])

        if street_end not in expected and not intersect and tag['v'] not in no_fix:
            st_name = process.extractOne(tag['v'], possible_streets)
            if st_name[1] >= 90:
                tag['v'] = st_name[0]
```

```
'US 70 and US 206', ('Campus Crossings E and F street', 86)
'Route 33', ('South 33rd Street', 68)
'North 37th', ('North 37th Street', 90)
'North 43rd', ('North 43rd Street', 90)
'1 Brookline Boulevard Havertown PA 19083(610) 446-1234', ('Brookline Boulevard', 90)
'West Girard Avenue, 5', ('West Girard Avenue', 95)
'DAVISVIL LE RO AD', ('Davisville Road', 94)
```

Not all matches are perfect, so strings are only replaced if they score 90 or above. This ratio is calculated from Levenshtein distance, so a score of 0 would indicate that the number of changes required is the maximum possible, while a score of 100 indicates that the number of changes required is the minimum possible.

If we wanted to be super thorough, we could use the Census Bureau's latest TIGER Roads National Geodatabase:

- TIGER Geodatabase Home
- 2016 TIGER Geodatabase Guide

### 13 Roads National Geodatabase

*The Geodatabase name is: tlgdb_2016_a_us_roads.gdb*

| *Layers in Geodatabase* |
|---|
| Roads |

This geodatabase contains all roads for the nation.

#### 13.1 Roads

| Field | Length | Type | Description |
|---|---|---|---|
| LINEARID | 22 | String | Linear feature identifier |
| FULLNAME | 100 | String | Concatenation of expanded text for prefix qualifier, prefix direction, prefix type, base name, suffix type, suffix direction, and suffix qualifier (as available) with a space between each expanded text field |
| RTTYP | 1 | String | Route type code |
| MTFCC | 5 | String | MAF/TIGER feature class code |
| PREQUAL | 3 | String | Expanded text for prefix qualifier (as available) |
| PREDIR | 2 | String | Expanded text for prefix direction (as available) |
| PRETYP | 14 | String | Expanded text for prefix type (as available) |
| NAME | 100 | String | Base name |
| SUFTYP | 14 | String | Expanded text for suffix type (as available) |
| SUFDIR | 2 | String | Expanded text for suffix direction (as available) |
| SUFQUAL | 3 | String | Expanded text for suffix qualifier (as available) |

Figure 2: Tiger National Roads

The file is 2.1 GB, though, so we would have to upload it to S3 and rewrite our code in Spark to make use of it. Perhaps it could be used for n-gram matching? Would that be faster? Databricks/Spark is probably a better way to test if this is possible. It's also not clear how to query this dataset by city or state, so more processing would be required.

## Conclusion

The data from the OSM XML are inconsistent and rife with errors. Though this project cleaned most street address strings, many of the other 853 tag keys appear to contain misspellings and values that correspond to other tag keys. To ensure cleaner, consistent data in the future, three projects may be worthwhile:

- Request lists of clean street names and any other relevant data fields from local, state and federal open data projects to use as canonical references for string and n-gram matching algorithms. It's possible that TIGER data may already meet specification, though the large file sizes would require distributed computing resources and a different programming approach, ie, Apache Spark.

- Deploy these algorithms via a user input validator within the two main OSM editors. This could be built with Django.
- Similarly constrain tag key-value pairs to those listed on the OSM Wiki to ensure that only recognized tag key-value pairs can be entered in the dataset as tag elements.

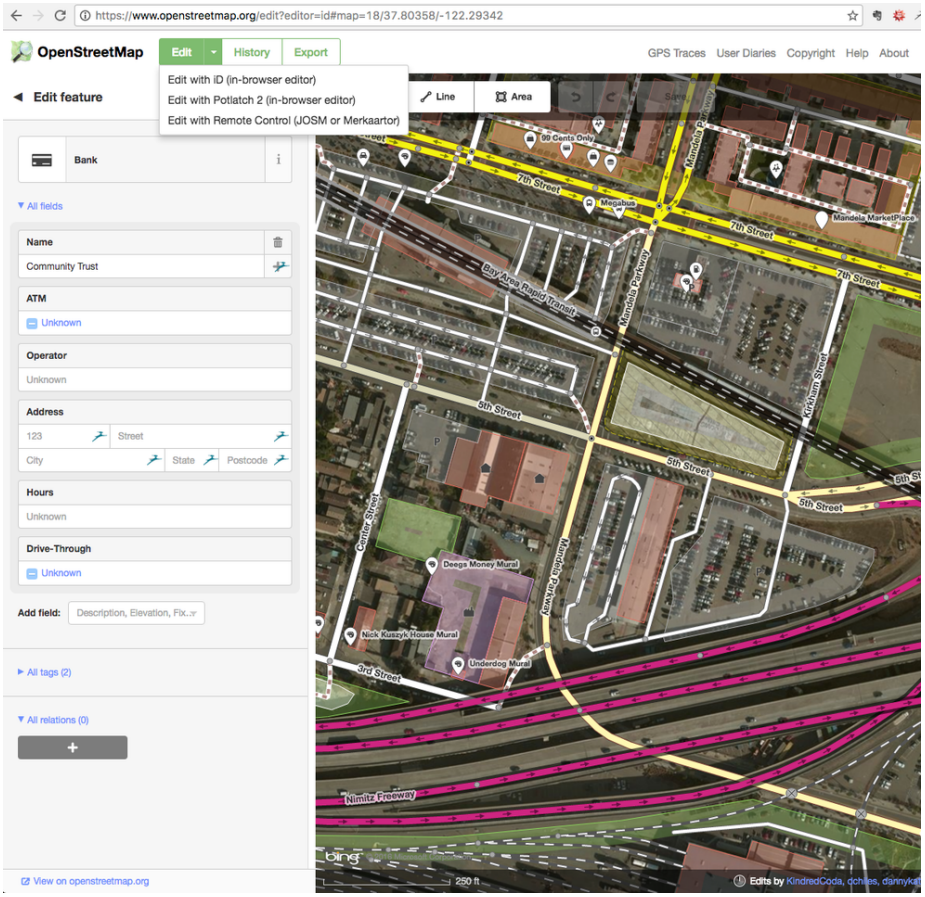


Figure 3: OsmiDeditor