OSM Data Wrangling For Madison, WI

Trevor McGlynn

June 2020

1 Introduction

OpenStreetMap (OSM) is a collaborative project to create a free an editable map of the world, much like what Wikipedia did for the encyclopeida. OSM offers a community-supported version of the dominant mapping application, Google Maps.

For this project, we will be wrangling OSM data for Madison, WI. The OSM file was pulled from BBBike Exports of OpenStreetMap, which makes available copies of OSM data for more than 200 cities and regions world wide. I lived in Madison for a little while, and just recently moved from there in January, so I'm a bit familiar with this region.

To start, we will import some modules we will be using, the files that will be used to form the tables, and some regular expressions we will be using to validate the data as it comes to us. Due to the size of the OSM file we will be using the `xml.etree` module to iteratively parse through each line of XML data.

Next, we will write some functions that will help us audit the XML data and categorize it for us, based on what we want to be editing/validating. the `audit()` function will perform checks for `is_x` where x is one of the areas we will be validating. Once determined, an `audit_x` function will be performed to check the validity of the data entry against our regular expressions.

Map source: https://download.bbbike.org/osm/bbbike/

2 Auditing the Data

The first task is to begin auditing the data for any inconsistencies which might show up in our database. Specifically, we will be looking at three fields where inconsistencies would be common: zip codes, phone numbers, and street addresses.

```
# ********************************************************************** #
      Auditing: ZIP CODES
# ********************************************************************** #
{'5': {'5'}}

# ********************************************************************** #
      Auditing: STREET NAMES
# ********************************************************************** #
{'106': {'East Cheryl Parkway, Ste 106'},
 '2611': {'2611'},
. . . .
  'Ave': {'Atlas Ave',
          'Commercial Ave',
          . . . .
          'Thornton Ave',
          'West Washington Ave'},
 'Ave.': {'E. Verona Ave.'},
. . . .
```

Not all of the data points that fail validation are considered invalid. In the case above, an address ending in "US-51" would be completely valid. This step requires a touch of additional human validation.

The values we select from this stage will be use to make "map" dictionaries for the three elements we'll be investigating: zip codes, street names, and phone numbers. We'll be using these later to correct them to standardized formats.

Below is the result of this scrubbing! Our updates on phone numbers are especially slick through use of a regular expression substitution (`re.sub()`) which eliminates for the various inconsistencies of a phone number:

```
^[+]?(1)?[- \.]?[( ]?(\d{3})[ )]?[- \.]?(\d{3})[- \.?]?(\d{4})$
```

This regex consists of four groups and a substitution pattern like so: `\2\3\4`.

Group 1 Optional country code (for the US, +1)

Group 2 Area code

Group 3 Exchange code, 3 digits

Group 4 Subscriber number, 3 digits

3 Parsing the XML Tree

Next, we will parse through the actual XML tree, shaping each of the elements to conform with what will then become a multiple csv files which will then be transformed into the tables which will make up our SQLite database. We'll start by defining a number of functions that will help us as we encounter each of the XML elements.

The main function takes the OSM data and processes it into csv files which will then be manipulated into SQL databases which will exist in the memory of the browser–this is the easiest way to present the database through Jupyter and also makes it nicely portable!

4 Managing & Querying the DB

Now that we have parsed the XML tree, we are ready to create an populate the database. For this, we will be using the library SQLAlchemy, which allows us to create a SQLite database in the memory of the browser. We will be taking the csv files that were created during the `process_map()` function and transform them in SQL databases using pandas.

Database Statistics

Our fileset includes the following files and their sizes. Note the original `Madison.osm` file at 152.8 MB.

File Name	File Size (MB)
`Madison.osm`	152.8
`node.csv`	47.7
`node_tag.csv`	1.3
`way.csv`	2.0
`way_node.csv`	33.7
`way_tag.csv`	9.2

Querying the DB

Now that we have the datasets properly loaded into the database, we can begin exploring some interesting features of Madison. These next couple of queries look at unique keys from the `node_tag` table that have at least 30 instances across Madison. We'll look at values in the midspread of that, to find a couple that would be worth looking into.

To start, let's look at some simple statistics about the database: the total number of nodes, and the total number of ways.

```
SELECT COUNT(id) AS "Number of Nodes" FROM node;
```

	Number of Nodes
0	1305256

```
SELECT COUNT(*) AS "Number of Ways" FROM way;
```

	Number of Ways
0	157906

There is plenty of data points we can look into using a dataset like this one. The choices, in fact, can be overwhelming! Sometimes the best thing to do is to set some kind of bottom limit and look from there. After looking at a statistical summary of the counts of all `node_tag` keys that have counts over 30 (see table below), another query was run based on intuition, further narrowing down the choices.

	count_keys
count	83.000
mean	469.048
std	1203.210
min	31.000
25%	55.000
50%	148.000
75%	440.500
max	10060.000

```
SELECT DISTINCT(key) AS unique_keys,
COUNT(*) AS count_keys
FROM node_tag
GROUP BY key
HAVING count_keys
BETWEEN 55 AND 440;
```

	unique_keys	count_keys
3	bench	56
22	parking	67
30	state_id	67
...		
35	tourism	337
40	website	349
10	direction	351
2	barrier	379
8	cuisine	407
16	leisure	410
25	postcode	421
29	state	425

Cuisine Types in Madison

This next query looks at the different types of cuisines available in Madison.

```
SELECT value AS "Type of Cuisine",
COUNT(*) AS "Count of Type"
FROM node_tag
JOIN (SELECT DISTINCT id FROM node_tag
WHERE value='restaurant') node_id
ON node_tag.id=node_id.id
WHERE key='cuisine'
GROUP BY value
ORDER BY "Count of Type" DESC
LIMIT 10;
```

	Type of Cuisine	Count of Type
0	pizza	29
1	mexican	16
2	chinese	15
3	asian	15
4	italian	10
5	indian	9

Leisure Activities

Similar to the types of cuisines, we can look at different kinds of leisure activities available in Madison.

```
SELECT value AS "Type of Leisure",
COUNT(*) AS "Count of Type"
FROM node_tag
JOIN (SELECT DISTINCT id FROM node_tag) node_id
ON node_tag.id=node_id.id
WHERE key='leisure'
GROUP BY value
HAVING "Count of Type" > 5
ORDER BY "Count of Type" DESC;
```

	Type of Leisure	Count of Type
0	playground	115
1	picnic_table	100
2	slipway	36
3	fitness_centre	32
4	pitch	27
5	firepit	18
6	park	17
7	sports_centre	13
8	outdoor_seating	10
9	garden	10

5 Ideas for Improvement

For there to be a need to "wrangle" data that comes from a single, primary source is a bit unnecessary. Ideally, validation of the data should occur on entry. In this analysis, input validity was checked on three different fields: zip codes, phone numbers, and street names. There we be an immediate and significant benefit on the quality of the dataset were these handled on entry. For instance, when a new node is tagged a business by a user, the entry of the telephone number for that business should follow a predicable, standardized pattern and, likewise, its record in the database should be in the exact same format as every other record with that attribute.

Benefits

- Validation upfront saves time and money later
- No additional effort is needed to scrub the data after it is stored in the database

Drawbacks

- User frustration, such as not understanding the format being requested
- Such an implementation would add complications for any bulk imports