OpenStreetMap - Data Wrangling

WGU | Data Wrangling
Udacity | Project: OpenStreetMap Data

Purpose

This project was created for Udacity's Data Analyst Nanodegree. An extract of xml data was downloaded for a selected city or region from OpenStreetMap (OSM). This document details the auditing, cleaning, transformation, and analysis performed on that raw dataset. After the raw data is cleaned and staged in tabular format (csv), it will be loaded into a database for additional analysis.

Selecting a Dataset

For this project I decided to work with data from Austin, TX. The selected map area is too large to export directly from OpenStreetMap[1], but I found a suitable extract hosted by Interline [2]. This particular extract was a pbf file, so I had to convert it to osm file format before auditing the data. I used a command line tool called osmosis to accomplish this.

```
# install homebrew
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)

# install osmosis
brew install osmosis

# convert pbf to osm
osmosis --read-pbf \austin_texas.osm.pbf --write-xml austin_texas.osm
```

Auditing

To begin the auditing process, I created three summaries; one each for elements, attributes, and keys. I created four functions to accomplish this.

```
import xml.etree.ElementTree as eT
```

```python
def print_sorted_dict(d, sort_by=None):
    """Prints dictionary sorted by keys or items"""

    if sort_by is None:
        sort_by = 'items'

    if sort_by == 'keys':
        sorted_dict = sorted(d.keys(), key=lambda s: s.lower())
    elif sort_by == 'items':
        sorted_dict = dict(sorted(d.items(), key=lambda s: s[1], reverse=True))
    else:
        print("Invalid sort_by: please input 'keys' or 'items'\n")
        sorted_dict = d

    for k in sorted_dict:
        v = d[k]
        print(f'{k}: {v}')


def count_elements(filename):
    """Prints element tag name and count for each XML element."""

    d = {}

    for event, elem in eT.iterparse(filename, events=('start',)):
        if elem.tag not in d:
            d[elem.tag] = 1
        else:
            d[elem.tag] += 1

    print('\n----- Count all tags -----')
    print_sorted_dict(d)
    return


def count_attributes(filename):
    """Prints attribute name and count for each XML element."""

    d = {}

    for event, elem in eT.iterparse(filename, events=('start', 'end')):
        if event == 'end':
            for attr in elem.attrib:
                if attr not in d:
                    d[attr] = 1
                else:
                    d[attr] += 1

    print('\n----- Count all attributes -----')
    print_sorted_dict(d)
```

```python
def count_keys(filename):
    """Prints key name and count for each XML element."""

    d = {}

    for event, elem in eT.iterparse(filename, events=('start', 'end')):
        if event == 'end':
            key = elem.attrib.get('k')
            if key:
                if key not in d:
                    d[key] = 1
                else:
                    d[key] += 1

    print('\n----- Count all keys -----')
    print_sorted_dict(d)
```

Using these functions, I printed a few summaries.

```python
from osmAudit import count_elements, count_attributes, count_keys

# define filename
atx_filename = 'data/austin_texas.osm'

# get count of elements
count_elements(atx_filename)

# get count of attributes
count_attributes(atx_filename)

# get count of keys
count_keys(atx_filename)
```

> ----- Count all tags -----
> nd: 8835948
> node: 7932057
> tag: 2967844
> way: 858496
> member: 58198
> relation: 4341
> osm: 1

> ----- Count all attributes -----
> ref: 8894146
> version: 8794895
> id: 8794894

> timestamp: 8794894
> uid: 8794894
> user: 8794894
> changeset: 8794894
> lat: 7932057
> lon: 7932057
> k: 2967844
> v: 2967844
> type: 58198
> role: 58198
> generator: 1

> ----- Count all keys -----
> building: 622302
> height: 441107
> addr:street: 345406
> addr:housenumber: 344583
> highway: 216664
> addr:postcode: 98282
> name: 73274
> service: 52447
> access: 41496
> tiger:county: 37785
> tiger:cfcc: 37703
> surface: 34399
> tiger:name_base: 33396
> tiger:name_type: 30904
> tiger:reviewed: 25054
> oneway: 25010
> power: 23772
> barrier: 19658
> addr:city: 19394
> addr:state: 19314
> ...

The basic components of the OpenStreetMap data model are tags, and the most important to this project are:

- node - describes points in space
- way - describes area boundaries and features
- relation - describe how other elements work together
- tag - describes the element to which they are attached, and contains two attributes: key (k) and value (v)

Exploring Key Values

Next, I'll check the top 10 keys - based on frequency of occurrence - to see where there are opportunities for data cleaning. I'll also check a few others that look interesting. I created a function to facilitate this portion of the audit.

```
import xml.etree.ElementTree as eT

def key_val_counter(filename, key_name):
    """Prints key name and count for each instance of key_name"""

    d = {}

    for event, elem in eT.iterparse(filename, events=('start', 'end')):
        if event == 'end':
            key = elem.attrib.get('k')
            if key == key_name:
                val = elem.attrib.get('v')
                if val not in d:
                    d[val] = 1
                else:
                    d[val] += 1

    print('\n----- Count of values for key: ' + key_name + ' -----')
    print_sorted_dict(d)
```

Because I'm working in a python notebook, I can't just loop through the keys I'm investigating. The printed data would get truncated well before all the keys' values were displayed. Instead, I'm going to run each key in it's own cell.

Most of the keys' values for Austin, TX OpenStreetMap data are already very clean. I suspect there are other students and hobbyists who have completed similar projects.

```
from osmAudit import key_val_counter

# define filename
atx_filename = 'data/austin_texas.osm'

# key -> height
key_val_counter(atx_filename, 'height')

# key -> addr:street
key_val_counter(atx_filename, 'addr:street')

# key -> addr:housenumber
key_val_counter(atx_filename, 'addr:housenumber')

# key -> highway
key_val_counter(atx_filename, 'highway')

# key -> name
key_val_counter(atx_filename, 'name')

# key -> service
key_val_counter(atx_filename, 'service')

# key -> tiger:county
key_val_counter(atx_filename, 'tiger:county')
```

Problem Tags

Although most of the top key-values are clean, there are a few with opportunities for cleaning or filtering. I'll outline how these tags were cleaned/filtered in the next section.

building

```
from osmAudit import key_val_counter

# define filename
atx_filename = 'data/austin_texas.osm'

# key -> building
key_val_counter(atx_filename, 'building')
```

> ----- Count of values for key: building -----
> yes: 584823
> house: 20797
> apartments: 4389

> detached: 2685
> carport: 1824
> retail: 954
> roof: 926
> commercial: 758
> school: 691
> residential: 565
> ...
> *stadium seating: 4* ←
> civic: 4
> ruins: 4
> container: 4
> temple: 3
> *sports_centre: 3* ←
> *covered area: 2* ←
> ...
> *big state electric: 1* ←
> tree_house: 1
> undefined: 1
> *Bing: 1* ←
> shelter: 1
> gas_station: 1
> transportation: 1
> *Learning_Center/_Day_Care: 1* ←
> ...

There are a few things that need to be cleaned-up in the values for the building key.

- There are spaces where there should be underscores. A simple string replace will correct those.
- A few other entries are incorrect or ambiguous; I'll correct those with a dictionary replace.

postcode

```
from osmAudit import key_val_counter

# define filename
atx_filename = 'data/austin_texas.osm'

# key -> addr:postcode
key_val_counter(atx_filename, 'addr:postcode')
```

> ----- Count of values for key: addr:postcode -----
> 78645: 10893
> 78734: 5627
> 78660: 4560

> 78653: 3553
> 78641: 3276
> 78669: 3190
> 78754: 2820
> 78704: 2559
> 78746: 2527
> 78723: 2290
> ...
> *78953: 3 ←*
> *78644: 2 ←*
> *78754;78753: 2 ←*
> *78704-5639: 1 ←*
> *78758-7008: 1 ←*
> ...

These data are mostly clean, but there are some post codes included that are not actually in Austin[3]. I'll filter those out while cleaning and staging the data.

surface

```
from osmAudit import key_val_counter

# define filename
atx_filename = 'data/austin_texas.osm'

# key -> surface
key_val_counter(atx_filename, 'surface')
```

> ----- Count of values for key: surface -----
> asphalt: 21169
> paved: 5156
> concrete: 3893
> unpaved: 1407
> *concrete:plates: 558 ←*
> ground: 518
> gravel: 452
> dirt: 391
> paving_stones: 250
> fine_gravel: 181
> ...
> cobblestone: 6
> *yes: 5 ←*
> *con: 3 ←*
> mud: 2
> ...

> *CR_127: 1 ←*
>
> *paving_stones:30: 1 ←*
>
> *creekbed_(rock): 1 ←*
>
> *concrete,_dirt: 1 ←*
>
> *Large_unattached_stones_laid_in_the_creek: 1 ←*
>
> woodchips: 1
>
> *f: 1 ←*

The values for the surface key need some cleaning. For some of them, I can figure out what the user intended - I can clean those with a dictionary. Some other values are less clear, and I'll remove those tags with a list.

city

```
from osmAudit import key_val_counter

# define filename
atx_filename = 'data/austin_texas.osm'

# key -> addr:city
key_val_counter(atx_filename, 'addr:city')
```

> ----- Count of values for key: addr:city -----
>
> Austin: 12095
>
> Cedar Park: 1985
>
> Pflugerville: 1137
>
> Round Rock: 1012
>
> Georgetown: 713
>
> Leander: 452
>
> Elgin: 437
>
> Hutto: 298
>
> Bastrop: 280
>
> Kyle: 181
>
> …
>
> Lost Pines: 2
>
> *AUSTIN: 2 ←*
>
> *Pfluggerville: 2 ←*
>
> *Wells Branch: 2 ←*
>
> *Barton Creek: 1 ←*
>
> *Ste 128, Austin: 1 ←*
>
> *San Gabriel Village Boulevard: 1 ←*
>
> Dale: 1
>
> *manor: 1 ←*
>
> *Pepe's Tacos: 1 ←*

> *N Austin: 1 ←*
>
> *Manchaca,: 1 ←*
>
> *Austin;austin: 1 ←*
>
> *kyle: 1 ←*
>
> Tampa: 1
>
> McNeil: 1
>
> Smithville: 1
>
> *wimberley: 1 ←*
>
> *Wimberly: 1 ←*
>
> Marble Falls: 1
>
> *georgetown: 1 ←*
>
> ...

Values for the addr:city tag are a little messy. To fix them, I'll capitalize just the first letter of each word in each city name. A dictionary match should clean up the remainder.

state

```
from osmAudit import key_val_counter

# define filename
atx_filename = 'data/austin_texas.osm'

# key -> addr:state
key_val_counter(atx_filename, 'addr:state')
```

> ----- Count of values for key: addr:state -----
>
> TX: 19311
>
> *FL: 1 ←*
>
> *AL: 1 ←*
>
> *tx: 1 ←*

There are a few non-Texas values in this key that need to be filtered out while cleaning and staging the data.

Other Considerations

I have a few additional cleaning steps to integrate into the data preparation function. There are also values for addr:postcode, addr:state, and surface that will be used to remove problematic elements. In addition to this, there are a set of characters that will cause problems when staging this data - any elements with these characters will be removed as well.

Cleaning & Transforming

To prepare the data for my database I need to clean and filter the raw OpenStreetMap data. Then, I need to transform the data from xml format to a tabular format (csv).

Cleaning

First, I wrote a function set for each of the problematic keys I outlined above.

building

For this key, I created a dictionary to correct a few bad values. The clean_building function compares the input value to that dictionary; if the value is contained in the dictionary keys, it's replaced with the dictionary value. Next, the value is checked for spaces, any that are located are replaced with an underscore.

```
building_dict = {
    'Bing': 'yes',
    'Learning_Center/_Day_Care': 'learning_center',
    'sports_centre': 'sports_center'
}


def clean_building(val):
    """Cleans key values for building tag"""

    # compare val to dictionary - if val in dict keys, replace with dict value
    for key in building_dict.keys():
        if val == key:
            val = building_dict.get(key)

    # replace any space with underscore
    val = val.replace(' ', '_')

    return val
```

postcode

I created two functions for the addr:postcode key:

- The clean_postcode function first takes the input value, splits on semicolon, and drops anything after the semicolon.

    - ‘12345; 98765’ → ‘12345’
- Next, it drops the last four digits from any values that have the full 9 digit zip code
    - 12345-6789 → 12345
- Then, the filter_postcode function checks a list of valid Austin, TX zip codes[3]. It returns *False* if that zip code is present on the list (meaning it should not be removed), and *True* if that zip code is not present on the list (meaning it should be removed).

```
atx_postcodes = [
    '78610', '78613', '78617', '78641', '78652', '78653', '78660', '78664',
    '78681', '78701', '78702', '78703', '78704', '78705', '78712', '78717',
    '78719', '78721', '78722', '78723', '78724', '78725', '78726', '78727',
    '78728', '78729', '78730', '78731', '78732', '78733', '78734', '78735',
    '78736', '78737', '78738', '78739', '78741', '78742', '78744', '78745',
    '78746', '78747', '78748', '78749', '78750', '78751', '78752', '78753',
    '78754', '78756', '78757', '78758', '78759'
]


def filter_postcode(val):
    """Filters key values for addr:postcode tag"""

    # run val through postcode cleaning function
    val = clean_postcode(val)

    # set to true if val is not an austin, tx zip code
    if val not in atx_postcodes:
        return True
    else:
        return False


def clean_postcode(val):
    """Cleans key values for addr:postcode tag"""

    # remove multiple zip code entries (e.g. '12345; 98765')
    split_val = val.split(';', maxsplit=1)
    val = split_val[0]

    # drop last four from full zip codes (e.g. 12345-6789 -> 12345)
    if len(val) == 10:
        val = val[0:5]

    return val
```

surface

I created two functions for the surface key:

- For the clean_surface function, I created a dictionary to correct a few bad values. Then, the input value is compared to that dictionary; if the value is contained in the dictionary keys, it is replaced with the dictionary value.
- The filter_surface function checks a list of values to remove. It returns True if the input value is on that list (meaning it should be removed), and False if the value is not on that list (meaning it should not be removed).

```
surface_dict = {
    'con': 'concrete',
    'large,_unattached_stones_through_water': 'stones',
    'Large_unattached_stones_laid_in_the_creek': 'stones',
    'paving_stones:30': 'paving_stones',
    'creekbed_(rock)': 'rock',
    'concrete,_dirt': 'concrete;dirt',
    'dirt/sand': 'dirt;sand',
    'concrete:lanes': 'concrete',
    'concrete:plates': 'concrete'
}

remove_list = [
    'yes', 'CR_127', 'f'
]


def filter_surface(val):
    """Cleans key values for surface tag"""

    # set to true if val is on the remove list
    if val in remove_list:
        return True
    else:
        return False


def clean_surface(val):
    """Cleans key values for surface tag"""

    # compare val to dictionary - if val in dict keys, replace with dict value
    for key in surface_dict.keys():
        if val == key:
            val = surface_dict.get(key)

    return val
```

city

The city key required the most cleaning among those I selected, and the function has multiple steps:

1. First, the function splits any city names that have multiple words.
    - round rock → [round, rock]
2. Then, it capitalizes each of those words by looping through each item in the list created when splitting the value.
    - [round, rock] → [Round, Rock]
3. Next, it puts the city names back together, and retains a space between each word.
    - [Round, Rock] → Round Rock
4. Finally, the value is compared to a dictionary to clean up any lingering incorrect city names.

```
city_dict = {
    'Wells Branch': 'Austin',
    'Barton Creek': 'Austin',
    'Ste 128, Austin': 'Austin',
    'Pepe’s Tacos': 'Austin',
    'N Austin': 'Austin',
    'Austin;austin': 'Austin',
    'San Gabriel Village Boulevard': 'Georgetown',
    'Manchaca,': 'Manchaca',
    'Pfluggerville': 'Pflugerville'
}


def clean_city(val):
    """Cleans key values for addr:city tag"""

    # split multi-word city names
    split = val.split(' ')
    i = 0
    cap = ''

    # capitalize first letter of each word - put names back together
    while i < len(split):
        x = split[i].capitalize()
        if i == 0:
            cap = x
        else:
            cap = cap + ' ' + x
        i += 1

    val = cap

    # compare val to dictionary - if val in dict keys, replace with dict value
    for key in city_dict.keys():
        if val == key:
            val = city_dict.get(key)

    return val
```

state

Creating a function just to filter for addr:state == TX would have been a textbook example of over-engineering a problem. Instead of creating a function, I just added that filter to the shape function outlined below.

Transforming

After the cleaning and filtering functions were developed, I wrote a function that shapes node and way elements of the raw xml file, and returns them as a Python dictionary.

Employing that function, the cleaning functions outlined above, and several helper functions provided by Udacity for this project; I cleaned, filtered, transformed, and staged the data into csv format to prepare it to be loaded into a sqlite database.

```
import cerberus
import codecs
import csv
import pprint
import re
import xml.etree.ElementTree as eT
import os

import schema
from osmKeySurface import filter_surface, clean_surface
from osmKeyPostcode import filter_postcode, clean_postcode
from osmKeyCity import clean_city
from osmKeyBuilding import clean_building
from TicToc import TicToc
t = TicToc()

# ========================================================================= #
#                            Define Variables                               #
# ========================================================================= #

# define file paths
OSM_PATH = 'data/austin_texas.osm'
NODES_PATH = 'data/csv/nodes.csv'
NODE_TAGS_PATH = 'data/csv/nodes_tags.csv'
WAYS_PATH = 'data/csv/ways.csv'
WAY_NODES_PATH = 'data/csv/ways_nodes.csv'
WAY_TAGS_PATH = 'data/csv/ways_tags.csv'

# define regex strings
LOWER_COLON = re.compile(r'^([a-z]|_)+:([a-z]|_)+')
PROBLEMCHARS = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')

# define schema
```

```
# define schema
SCHEMA = schema.schema

# set field order for csv export
NODE_FIELDS = [
    'id', 'lat', 'lon', 'user', 'uid', 'version', 'changeset', 'timestamp'
]

NODE_TAGS_FIELDS = [
    'id', 'key', 'value', 'type'
]

WAY_FIELDS = [
    'id', 'user', 'uid', 'version', 'changeset', 'timestamp'
]

WAY_TAGS_FIELDS = [
    'id', 'key', 'value', 'type'
]

WAY_NODES_FIELDS = [
    'id', 'node_id', 'position'
]


# ========================================================================= #
#                            Shape Function                                 #
# ========================================================================= #

def shape_element(element):
    """Shape node and way XML elements to Python dictionary"""

    way_attr_fields = WAY_FIELDS
    node_attr_fields = NODE_FIELDS
    problem_chars = PROBLEMCHARS
    default_tag_type = 'regular'

    node_attribs = {}
    way_attribs = {}
    way_nodes = []
    tags = []

    if element.tag == 'node':
        for i in node_attr_fields:
            node_attribs[i] = element.get(i)

        for j in element.iter('tag'):
            key = j.get('k')
            val = j.get('v')

            # drop problematic tags
```

```
            if re.match(problem_chars, key):
                continue
            if key == 'addr:state' and val != 'TX':
                continue
            if key == 'addr:postcode':
                if filter_postcode(val):
                    continue
            if key == 'surface':
                if filter_surface(val):
                    continue

            # cleaning functions
            if key == 'addr:postcode':
                val = clean_postcode(val)
            elif key == 'addr:city':
                val = clean_city(val)
            elif key == 'building':
                val = clean_building(val)
            elif key == 'surface':
                val = clean_surface(val)

            mat = re.match(LOWER_COLON, key)

            if mat:
                key_split = re.split(':', key, maxsplit=1)
                tags_dict = {
                    'id': node_attribs['id'],
                    'key': key_split[1],
                    'value': val,
                    'type': key_split[0]
                }

            else:
                tags_dict = {
                    'id': node_attribs['id'],
                    'key': key,
                    'value': val,
                    'type': default_tag_type
                }

            tags.append(tags_dict)

        return {
            'node': node_attribs,
            'node_tags': tags
        }

    elif element.tag == 'way':
        for i in way_attr_fields:
            way_attribs[i] = element.get(i)
```

```
    count = 0

    for x in element.iter('nd'):
        way_nodes_dict = {
            'id': way_attribs['id'],
            'node_id': x.get('ref'),
            'position': count
        }

        count += 1
        way_nodes.append(way_nodes_dict)

    for j in element.iter('tag'):
        key = j.get('k')
        val = j.get('v')

        # drop problematic tags
        if re.match(problem_chars, key):
            continue
        if key == 'addr:state' and val != 'TX':
            continue
        if key == 'addr:postcode':
            if filter_postcode(val):
                continue
        if key == 'surface':
            if filter_surface(val):
                continue

        # cleaning functions
        if key == 'addr:postcode':
            val = clean_postcode(val)
        elif key == 'addr:city':
            val = clean_city(val)
        elif key == 'building':
            val = clean_building(val)
        elif key == 'surface':
            val = clean_surface(val)

        mat = re.match(LOWER_COLON, key)

        if mat:
            key_split = re.split(':', key, maxsplit=1)

            way_tags_dict = {
                'id': way_attribs['id'],
                'key': key,
                'value': val,
                'type': key_split[0]
            }

        else:
```

```
                way_tags_dict = {
                    'id': way_attribs['id'],
                    'key': key,
                    'value': val,
                    'type': default_tag_type
                }

            tags.append(way_tags_dict)

        return {
            'way': way_attribs,
            'way_nodes': way_nodes,
            'way_tags': tags
        }


# ================================================================== #
#                  Assistant to the Regional Functions               #
# ================================================================== #

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()


def validate_element(element, validator, csv_schema=SCHEMA):
    """Raise ValidationError if element does not match schema"""

    if validator.validate(element, csv_schema) is not True:
        field, errors = next(iter(validator.errors.items()))
        message_string = '''\nElement of type '{0}' has the following
                            errors:\n{1}'''
        error_string = pprint.pformat(errors)

        raise Exception(message_string.format(field, error_string))


class UnicodeDictWriter(csv.DictWriter, object):
    """Extend csv.DictWriter to handle Unicode input"""

    def writerow(self, row):
        super(UnicodeDictWriter, self).writerow(row)

    def writerows(self, rows):
```

```
        for row in rows:
            self.writerow(row)


# ================================================================== #
#                         Main Function                              #
# ================================================================== #

def process_map(file_in, validate):
    """Iteratively process each XML element and write to csv(s)"""

    with codecs.open(NODES_PATH, 'w', encoding='utf8') as nodes_file, \
            codecs.open(NODE_TAGS_PATH, 'w', encoding='utf8') as nodes_tags_file, \
            codecs.open(WAYS_PATH, 'w', encoding='utf8') as ways_file, \
            codecs.open(WAY_NODES_PATH, 'w', encoding='utf8') as way_nodes_file, \
            codecs.open(WAY_TAGS_PATH, 'w', encoding='utf8') as way_tags_file:

        nodes_writer = UnicodeDictWriter(nodes_file, NODE_FIELDS)
        node_tags_writer = UnicodeDictWriter(nodes_tags_file, NODE_TAGS_FIELDS)
        ways_writer = UnicodeDictWriter(ways_file, WAY_FIELDS)
        way_nodes_writer = UnicodeDictWriter(way_nodes_file, WAY_NODES_FIELDS)
        way_tags_writer = UnicodeDictWriter(way_tags_file, WAY_TAGS_FIELDS)

        nodes_writer.writeheader()
        node_tags_writer.writeheader()
        ways_writer.writeheader()
        way_nodes_writer.writeheader()
        way_tags_writer.writeheader()

        validator = cerberus.Validator()

        for element in get_element(file_in, tags=('node', 'way')):
            elem = shape_element(element)

            if elem:
                if validate is True:
                    validate_element(elem, validator)

                if element.tag == 'node':
                    nodes_writer.writerow(elem['node'])
                    node_tags_writer.writerows(elem['node_tags'])

                elif element.tag == 'way':
                    ways_writer.writerow(elem['way'])
                    way_nodes_writer.writerows(elem['way_nodes'])
                    way_tags_writer.writerows(elem['way_tags'])


# ================================================================== #
#                              Execute                               #
# ================================================================== #
```

```
# ========================================================================== #

if __name__ == '__main__':
    py = os.path.basename(__file__)
    print('\nExecuting ' + py + '....')
    t.tic()

    process_map(OSM_PATH, validate=False)

    t.toc()
```

Problems Encountered

I encountered several problems while working with the Austin OpenStreetMap data. Chief among them was file size. I did not anticipate how resource intensive working with a dataset of this size would be. If I were to do this project again I would select a smaller map to work with. As you can see, some of the files used are quite large.

```
import os
import pandas as pd
from TicToc import TicToc
t = TicToc()

# ========================================================================== #
#                            Define Variables                                #
# ========================================================================== #

root_data = '/Users/ajp/dsProjects/workspace/osmAustin/data/'
root_csv = root_data + 'csv/'

osm = 'austin_texas.osm'
sample = 'sample_atx.osm'
nodes_tags = 'nodes_tags.csv'
nodes = 'nodes.csv'
ways_nodes = 'ways_nodes.csv'
ways_tags = 'ways_tags.csv'
ways = 'ways.csv'

path_osm = root_data + osm
path_sample = root_data + sample
path_nodes_tags = root_csv + nodes_tags
path_nodes = root_csv + nodes
path_ways_nodes = root_csv + ways_nodes
path_ways_tags = root_csv + ways_tags
path_ways = root_csv + ways

# ========================================================================== #
#                              Size Function                                 #
```

```
#                              Size Function                                 #
# ========================================================================== #


def get_size(filepath):
    """Get file size and return string with appropriate unit"""

    size_bytes = os.path.getsize(filepath)

    if size_bytes < 1024:
        size_bytes = round(size_bytes, 2)
        size = f'{size_bytes} B'
    else:
        size_kilobytes = size_bytes / 1024
        if size_kilobytes < 1024:
            size_kilobytes = round(size_kilobytes, 2)
            size = f'{size_kilobytes} KB'
        else:
            size_megabytes = size_kilobytes / 1024
            if size_megabytes < 1024:
                size_megabytes = round(size_megabytes, 2)
                size = f'{size_megabytes} MB'
            else:
                size_gigabytes = size_megabytes / 1024
                if size_gigabytes < 1024:
                    size_gigabytes = round(size_gigabytes, 2)
                    size = f'{size_gigabytes} GB'
                else:
                    size = "Wow, that's huge."

    return size


# ========================================================================== #
#                                Execute                                     #
# ========================================================================== #

if __name__ == '__main__':
    py = os.path.basename(__file__)
    print('\nExecuting ' + py + '....')
    t.tic()

    osm_size = get_size(path_osm)
    sample_size = get_size(path_sample)
    nodes_tags_size = get_size(path_nodes_tags)
    nodes_size = get_size(path_nodes)
    ways_nodes_size = get_size(path_ways_nodes)
    ways_tags_size = get_size(path_ways_tags)
    ways_size = get_size(path_ways)

    names = [
```

```
        osm,
        sample,
        nodes_tags,
        nodes,
        ways_nodes,
        ways_tags,
        ways
    ]

    sizes = [
        osm_size,
        sample_size,
        nodes_tags_size,
        nodes_size,
        ways_nodes_size,
        ways_tags_size,
        ways_size
    ]

    sizes_dict = {
        'name': names,
        'size': sizes
    }

    sizes_df = pd.DataFrame(data=sizes_dict)
    print(sizes_df)

    t.toc()
```

	name	size
0	austin_texas.osm	1.66 GB
1	sample_atx.osm	40.06 MB
2	nodes_tags.csv	13.8 MB
3	nodes.csv	696.06 MB
4	ways_nodes.csv	203.81 MB
5	ways_tags.csv	85.79 MB
6	ways.csv	56.83 MB

To work through this problem, I created a sample file that only contains elements with ways tags for addr:state=TX. This step significantly sped up testing and development, reducing the working file size from 1.79 GB to 42 MB. I did this with the java-based osmosis tool used earlier in this document.

```
# get sample dataset for testing/development
osmosis --rx file=austin_texas.osm --tf accept-ways addr:state=TX --un --wx sample_atx.osm
```

I also had a little trouble finding data to clean. Many of the keys' values were already very clean. I suspect that since Austin is a tech city other students and hobbyists like myself have done similar projects and cleaned the OpenStreetMap data for this metropolitan area.

Sqlite Database

After the data was cleaned, I created a sqlite database[4]. Then, I created a schema in that database to match the schema of my csv files. After that, I loaded the data into their respective tables.

```
create table nodes
(
    id        integer not null
        constraint nodes_pk
            primary key,
    lat       real,
    lon       real,
    user      text,
    uid       integer,
    version   integer,
    changeset integer,
    timestamp text
);


create table nodes_tags
(
    id    integer
        references nodes,
    key   text,
    value text,
    type  text
);


create table ways
(
    id        integer not null
        constraint ways_pk
            primary key,
    user      text,
    uid       integer,
    version   text
```

```
    version   text,
    changeset integer,
    timestamp text
);


create table ways_nodes
(
    id        integer not null
        references ways,
    node_id  integer not null
        references nodes,
    position integer not null
);


create table ways_tags
(
    id    integer not null
        references ways,
    key   text    not null,
    value text    not null,
    type  text
);
```

Overview of the Data

SQL Stats

Before digging into the dataset, I took a look at some database stats to get an idea of how much data I'd be working with. To generate those stats, I used another command-line utility program called sqlite3_analyzer[5].

```
sqlite3_analyzeer --stats osmDB.sqlite
```

Then, I loaded the stats into a table in my database.

The insert statements for each set of statistics are extremely long, so I'll leave them out of this document. However, they can be found in this repo for reference.

```
BEGIN;
CREATE TABLE stats(
    name       STRING,           /* Name of table or index */
    path       INTEGER,          /* Path to page from root */
    pageno     INTEGER,          /* Page number */
    pagetype   STRING,           /* 'internal', 'leaf' or 'overflow' */
    ncell      INTEGER,          /* Cells on page (0 for overflow) */
    payload    INTEGER,          /* Bytes of payload on this page */
    unused     INTEGER,          /* Bytes of unused space on this page */
    mx_payload INTEGER,          /* Largest payload size of all cells */
    pgoffset   INTEGER,          /* Offset of page in file */
    pgsize     INTEGER           /* Size of the page */
);
COMMIT;
```

After the stats table was created, I wrote a simple query to check table sizes.

```
SELECT name AS table_name
     , CASE
           WHEN bytes < 1024 THEN (bytes || ' B')
           WHEN kilobytes < 1024 THEN ROUND(kilobytes, 2) || ' KB'
           ELSE ROUND(megabytes, 2) || ' MB' END AS table_size
FROM (
         SELECT name
              , SUM(payload) as bytes
              , CAST(SUM(payload) AS FLOAT) / 1024 AS kilobytes
              , (CAST(SUM(payload) AS FLOAT) / 1024) / 1024 AS megabytes
         FROM stats
         GROUP BY name
     )
ORDER BY bytes DESC;
```

	table_name	table_size
1	nodes	528.53 MB
2	ways_nodes	123.58 MB
3	ways_tags	73.9 MB
4	ways	42.98 MB
5	nodes_tags	12.23 MB
6	sqlite_schema	2.33 KB

Analysis

Now that the stats were collected, I could start analyzing the clean data. As I was investigating this dataset, I had several questions:

- How many people have contributed to the Austin OpenStreetMap project?

```
SELECT COUNT(DISTINCT uid) AS contributors
FROM (
       SELECT uid FROM nodes
      UNION ALL
       SELECT uid FROM ways
     );
```

	contributors
1	2973

- Who are the top 10 contributors?

```
SELECT uid
     , user
     , COUNT(*) AS contributions
FROM (
       SELECT uid, user FROM nodes
      UNION ALL
       SELECT uid, user FROM ways
     )
GROUP BY uid, user
ORDER BY contributions DESC
LIMIT 10;
```

	uid	user	contributions
1	3369502	patisilva_atxbuildings	2638615
2	3370181	ccjjmartin_atxbuildings	1257245
3	3405475	ccjjmartin__atxbuildings	920175
4	3341346	wilsaj_atxbuildings	349180
5	3341321	jseppi_atxbuildings	284518
6	147510	woodpeck_fixbot	179918
7	3367383	kkt_atxbuildings	155199

	uid	user	contributions
8	3409435	lyzidiamond_atxbuildings	150603
9	5446055	torapa	131329
10	12056179	Milroy1812	124175

- How many total contributions are made each year?

```
SELECT year
     , COUNT(*) AS contributions
FROM (
       SELECT SUBSTR(timestamp, 1, 4) AS year FROM nodes
      UNION ALL
       SELECT SUBSTR(timestamp, 1, 4) AS year FROM ways
     )
GROUP BY year
ORDER BY year;
```

	year	contributions
1	2007	2111
2	2008	18020
3	2009	223961
4	2010	20007
5	2011	44811
6	2012	112824
7	2013	62621
8	2014	97015
9	2015	5958603
10	2016	73128
11	2017	79173
12	2018	227833
13	2019	581022
14	2020	534813
15	2021	754611

- During which months are the contributors most active?

```
SELECT CASE
        WHEN mon = '01' THEN 'January'
        WHEN mon = '02' THEN 'February'
        WHEN mon = '03' THEN 'March'
        WHEN mon = '04' THEN 'April'
        WHEN mon = '05' THEN 'May'
        WHEN mon = '06' THEN 'June'
        WHEN mon = '07' THEN 'July'
        WHEN mon = '08' THEN 'August'
        WHEN mon = '09' THEN 'September'
        WHEN mon = '10' THEN 'October'
        WHEN mon = '11' THEN 'November'
        WHEN mon = '12' THEN 'December'
        END AS month
      , COUNT(*) AS contributions
FROM (
       SELECT SUBSTR(timestamp, 6, 2) AS mon FROM nodes
      UNION ALL
       SELECT SUBSTR(timestamp, 6, 2) AS mon FROM ways
     )
GROUP BY month
ORDER BY mon;
```

	month	contributions
1	January	246686
2	February	288007
3	March	289744
4	April	215569
5	May	153204
6	June	294502
7	July	246379
8	August	294520
9	September	211860
10	October	274377
11	November	4806845
12	December	1468860

- How many total ways tags are in this dataset, and what are the most common tags?

```
-- count the ways
SELECT COUNT(*) AS n
FROM ways;

-- common way tags
SELECT key
     , COUNT(*) AS n
FROM ways_tags
GROUP BY key
HAVING n > 25000
ORDER BY n DESC;
```

	n
1	858496

	key	n
1	building	620762
2	height	438569
3	addr:street	261635
4	addr:housenumber	260847
5	highway	189824
6	name	63984
7	service	52432
8	access	41040
9	tiger:county	37785
10	tiger:cfcc	37703
11	surface	34319
12	tiger:name_base	33396
13	tiger:name_type	30904
14	tiger:reviewed	25023
15	oneway	25009

- How many total nodes tags are in this dataset, and what are the most common tags?

```
-- count the nodes
SELECT COUNT(*) AS n
FROM nodes;

-- common node tags
SELECT key
     , COUNT(*) AS n
FROM nodes_tags
GROUP BY key
HAVING n > 3000
ORDER BY n DESC;
```

	n
1	7932057

	key	n
1	street	83171
2	housenumber	83135
3	postcode	62267
4	highway	26828
5	barrier	16292
6	power	13811
7	name	7900
8	amenity	6776
9	natural	5955
10	crossing	5676
11	created_by	4476
12	kerb	4300
13	city	3907
14	state	3843
15	place_id	3433

- What are the most common amenities in Austin, TX?

```
SELECT value
     , COUNT(*) AS n
FROM nodes_tags
WHERE key='amenity'
GROUP BY value
HAVING n >= 100
ORDER BY n DESC;
```

	value	n
1	restaurant	916
2	bench	887
3	waste_basket	791
4	fast_food	429
5	loading_dock	316
6	parking_entrance	257
7	place_of_worship	231
8	bicycle_parking	219
9	cafe	194
10	bar	183
11	fuel	167
12	pharmacy	131
13	waste_disposal	126
14	fountain	126
15	school	124
16	dentist	120
17	letter_box	109

- What kind of restaurants are popular in Austin?

```
SELECT value
     , COUNT(*) AS n
FROM nodes_tags
WHERE key = 'cuisine'
GROUP BY value
ORDER BY n DESC
LIMIT 10;
```

	value	n
1	sandwich	116
2	pizza	115
3	mexican	110
4	coffee_shop	77
5	burger	66
6	chinese	40
7	american	38
8	indian	29
9	thai	25
10	italian	24

Other Ideas About the Dataset

I think one of the best things OpenStreetMap could do to improve their data would be to develop a robust, automated cleaning process. These scripts or bots could automatically make corrections of specific errors.

It looks like there is a bot (woodpeck_fixbot) modifying elements in the Austin dataset, but the scope of that project must be relatively narrow because I still found simple corrections to make in the data.

The kind of bot I'm imagining is more extensive, one example would be Wall-E[6], but it is currently only operational in Germany and Austria. Perhaps OSM is worried that an automated correction bot for a map as large as the United States could cause problems if it was making inaccurate 'corrections'.

Those hurdles could be overcome through proper testing. Specifically, by testing features on a very small area, and slowly widening the bot's scope before unleashing it on the entire map.

Conclusion

These data went on a long journey before landing in a clean sqlite database.

- I converted pbf file to osm format using osmosis, a java-based command line tool.
- Then I investigated the raw data in a python notebook.
- Next, I cleaned, filtered, and transformed the data using a handful of python scripts and functions.
- Finally, I loaded the data into a sqlite database and analyzed it with SQL.

There is additional work that could be done on the OpenStreetMap data for Austin, TX. Also, data issues are likely to be a constant problem for OSM until someone implements a more widespread automated cleaning process.

```
    ____                           ______                __
   / __ )_________  ____  ____ ___/_  __/___  ____ _____/ /
  / __  / ___/ __ \/ __ \/_  // _ \/ / / __ \/ __ `/ __  /
 / /_/ / /  / /_/ / / / / / //  __/ / / /_/ / /_/ / /_/ /
/_____/_/   \____/_/ /_/ /___|___/_/  \____/\__,_/\__,_/
```