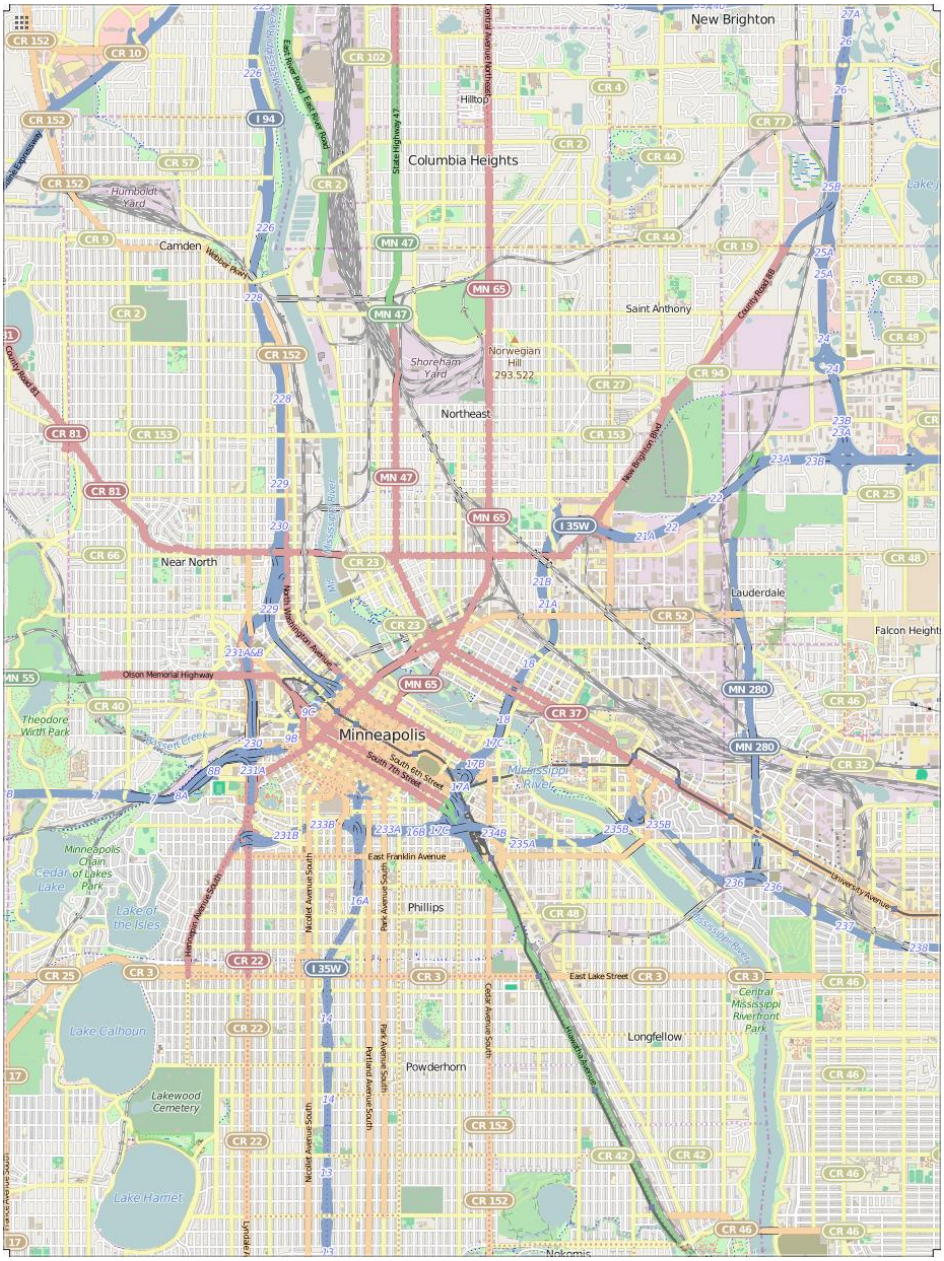
Table of Contents

Introduction

For this project I have chosen my home area Minneapolis, MN. The reason is to be better able to validate the data by cross-referencing the data with personal knowledge of the area. The area is mainly Minneapolis, MN with some of the suburbs including New Brighton, Hopkins, and Minneapolis' Twin City, St. Paul. Here is a link to the Overpass API for downloading the OSM for the region. http://overpass-api.de/api/map?bbox=-93.3292,44.9150,-93.1713,45.0644.

All the relevant code will be located in the end of this document with all the links that helped me complete the project. [back to top]

Below is the Map screenshotted from OSM.



1. Problems encountered in your Map

[back to top]

- File-size > 87MB
  i. Even with iterative parsing, auditing and identifying meaningful patterns took longer than expected
  ii. Took a smaller sample (<12MB) for creating functions for auditing, parsing, cleaning, etc.
  iii. Once the functions were complete, they were applied to the 87MB OSM file
- Values for cities were redundant
  i. For example—there were 4 different representations of Saint Paul. Here are the mappings

```
city_mapping = {"Minneapolis, MN" : "Minneapolis",
                "St. Anthony" : "Saint Anthony",
                "St. Paul" : "Saint Paul",
                "St Anthony" : "Saint Anthony",
                "St Paul" : "Saint Paul"}
```

- Values for postal codes ranged from being 3-digits long to 9-digits, alphabetical, or simply repeated values from different fields (*postcode = street*). One thing to note is out of the 108 unique *postcode* values, once extracting only the 5-digit prefixes, we were left with only 33 unique results which is consistent with the area I've chosen.
  i. First I chose to limit the postal codes to 5-digit lengths. This is to standardize the data by the 'least common denominator' so the data was more intuitively queryable.
  ii. The problem postal codes had letters, were too short, or misplaced from a different field. The only way to find the correct postal codes was to search the internet using the other fields for the search criteria and then extrapolate the postal codes. It could be done case-by-case manually but it would be faster if we could do it programmatically.
  iii. Luckily I found pygeocoder
    - Pygeocoder is a Python Library that utilizes Google's Geocoding Functionality to extrapolate geographic information such as coordinates, postal codes, etc.
    - I decided to use the *reverse_geocode()* function which will take the latitudinal and longitudinal coordinates and return an object from which the postal codes could be called from.

```
> from pygeocoder import Geocoder
> results = Geocoder.reverse_geocode(45.0079126, -93.2473816)
> int(results.postal_code)
> 55418
```

    - There were some *postalcode* fields which did not have coordinate systems to use. For this, Pygeocoder also has the *geocode()* function which takes in a string argument, preferably a street address and state, and returns an object which also contains the postal code information.

```
> results = Geocoder.geocode('The Carlyle MN')
> int(results.postal_code)
> 55401
```

- Values for street suffixes were redundant

i. The original mapping from Lesson 6 only covered a few keys to be normalized. After sifting through the data, I have discovered many more keys. Here is my mapping for street abbreviations.

```
street_mapping = { "St": "Street",
                   "St.": "Street",
                   "Ave" : "Avenue",
                   "Rd." : "Road",
                   "Rd" : "Road",
                   "SE" : "Southeast",
                   "S.E." : "Southeast",
                   "NE" : "Northeast",
                   "S" : "South",
                   "Dr" : "Drive",
                   "Rd/Pkwy" : "Road/Parkway",
                   "Ln" : "Lane",
                   "Dr." : "Drive",
                   "E" : "East",
                   "Pl" : "Plain",
                   "ne" : "Northeast",
                   "NW" : "Northwest",
                   "Ave." : "Avenue",
                   "N." : "North",
                   "W" : "West",
                   "Pkwy" : "Parkway",
                   "Ter" : "Terrace",
                   "Pky" : "Parkway",
                   "SW" : "Southwest",
                   "N" : "North",
                   "Blvd" : "Boulevard"}
```

- The original *shape_element* function created in Lesson 6, when applied to this particular OSM file resulted in 447 unique top-level descriptors. A large number of them had similar prefixes meaning they could be grouped under a single dictionary to simplify querying.

  My goal is to aggregate related descriptors into a single top-level key which can be accessed by calling that key. This will result in a more organized representation of the data. The following describes the grouped descriptors and their sources.

i. 'address:' fields
   - These were fields dealt with in Lesson 6

ii. 'metcouncil:' fields
   - These fields with information from the Metropolitan Council of Minnesota

iii. 'tiger:' fields
   - **T**opologically **I**ntegrated **G**eographic **E**ncoding and **R**eferencing system (TIGER) data was produced by the US Census Bureau and later merged onto OSM database

iv. 'metrogis:' fields
   - Regional geographic information system of the seven-county metropolitan area of Minneapolis-St. Paul, Minnesota

v. 'umn:' fields
   - Fields related to the University of Minnesota, example.
   - This field presented a unique programmatic problem—If it existed, the positional data (Latitude,Longitude) was located in its parent element (in this case *way*), it would be located in a *umn:* k sub-element.

vi. 'gnis:' fields
   - USGS **G**eographic **N**ames **I**nformation **S**ystem (GNIS) contains millions of names for geographic features in the US.

After grouping I reduced the number of descriptors from 447 to 364. The fields mentioned above are important because they determine the sources of information merged onto the OSM user-based database. Cross-referencing these fields could determine accuracy, constituency, and completeness of the data.

2. Overview of the Data

[back to top]

- Statistics
  - i. File Sizes
    - map.osm – 87MB
    - map.json – 95MB
  - ii. Documents

```
> db.minneapolis.find().count()
> 436802
```

  - iii. Nodes

```
> db.minneapolis.find({'type' : 'node'}).count()
> 373017
```

  - iv. Way

```
> db.minneapolis.find({'type' : 'way'}).count()
> 59143
```

  - v. Unique users

```
> len(db.minneapolis.distinct('created.user'))
> 401
```

  - vi. Top 5 cuisines in the category of restaurant

```
> pipeline = [{'$match' : {'amenity' : 'restaurant',
                           'cuisine' : {'$exists' : 1}}},
            {'$group' : {'_id' : '$cuisine',
                         'count' : {'$sum' : 1}}},
            {'$sort' : {'count' : -1}},
            {'$limit' : 5}]
> top5 = db.minneapolis.aggregate(pipeline)
> for i in top5['result']:
         print i['_id'], i['count']
> pizza 16
  chinese 15
  american 10
  italian 10
  regional 7
```

3. Other ideas about the Datasets

[back to top]

I've decided to take the positional data (latitude, longitude) of the top 5 contributing users and create a graphical display of the places they contributed overlaid on the map area the dataset came from. This should create a sort of 'heat-map' of each users' territory, so to speak. We may be able to see home neighborhoods or traveling patterns. The image of the top 5 contributors on to one map became too confusing to interpret so I split it into 5 individual maps/graphs. The codes for making the maps and an explanation of each step of the code will be available in the Codes section.

```
> pipeline = [{'$group' : {'_id' : '$created.user', 'count' : {'$sum' : 1}}},
                        {'$sort' : {'count' : -1}},
                        {'$limit' : 5}]
> top5 = db.minneapolis.aggregate(pipeline)
> for i in top5['result']:
                print i['_id'], i['count']
> Mulad 146082
  iandees 81267
  stucki1 63803
  DavidF 27032
  sota767 25901
```

- From the previous section we have discovered the top 5 contributors to be…
  i. Mulad – 146082

```
## Query to find percentage of total documents user contributed
db.minneapolis.find({'created.user': 'Mulad'}).count() /
float(db.minneapolis.find().count())
```

  - ~33.4 % of the total documents

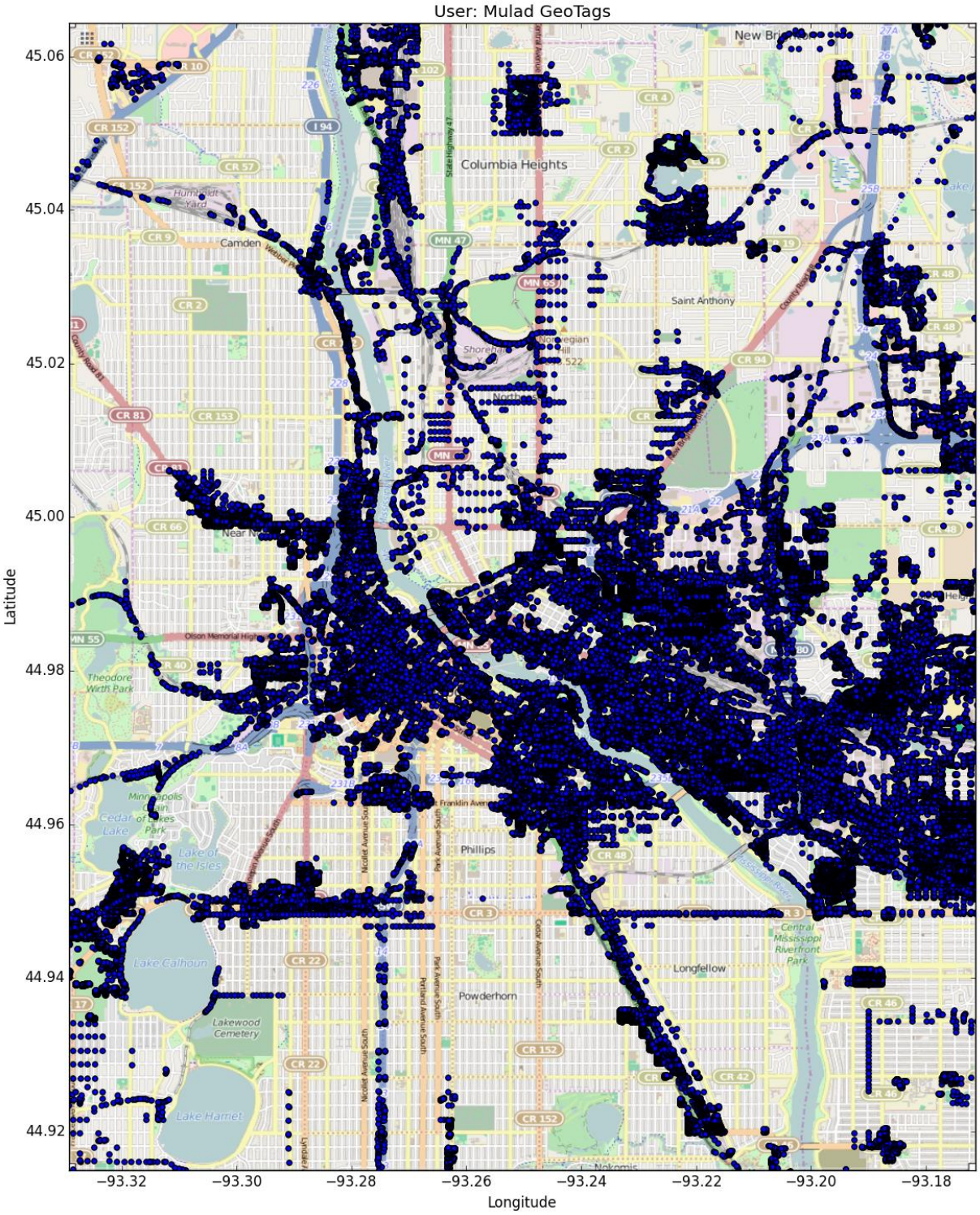
  ii. iandees – 81267
  - ~18.6 % of the total documents

  iii. stucki1 – 63803
  - ~14.6 %

  iv. DavidF – 27032
  - ~6.2 %

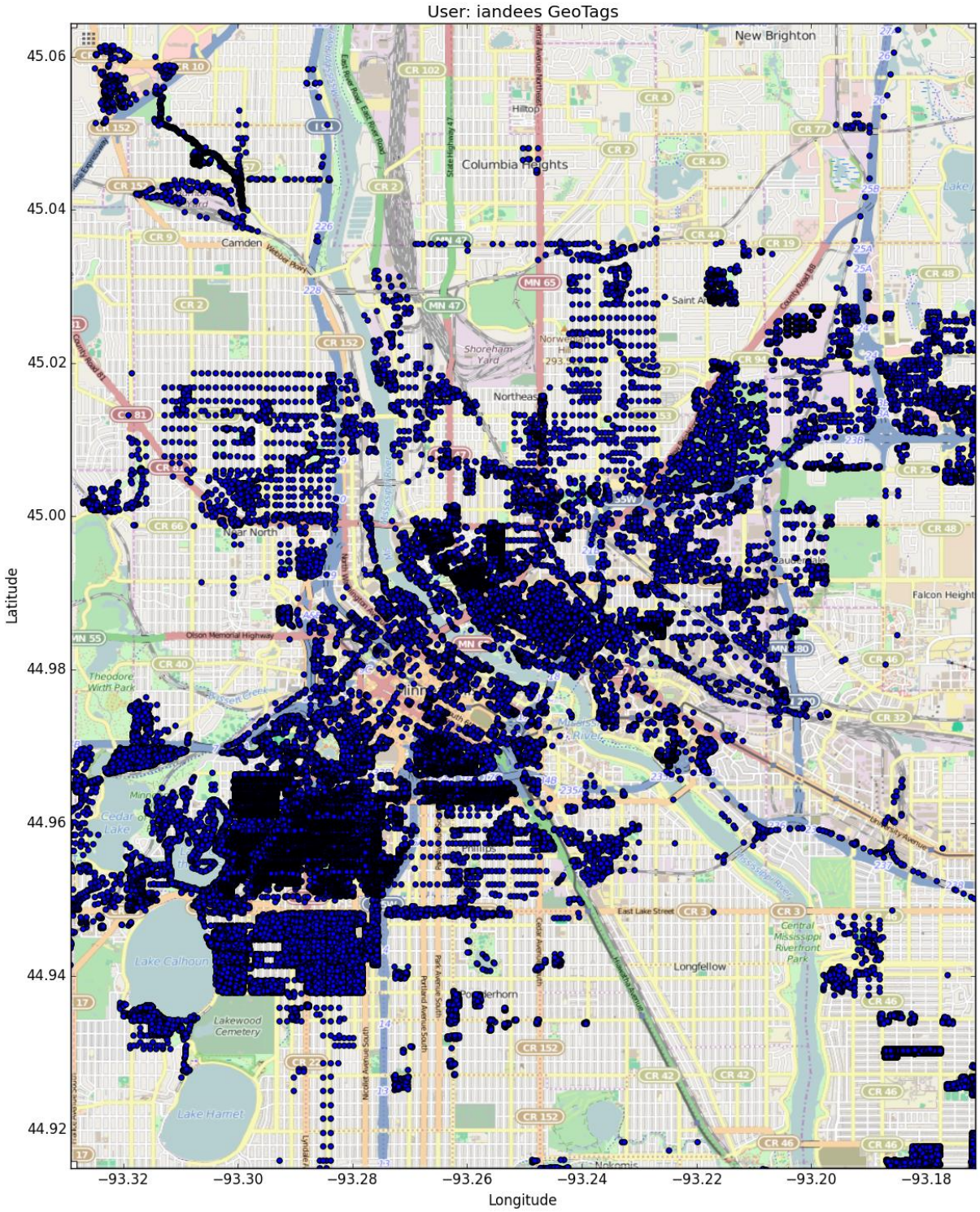
  v. sota767 – 25901
  - ~5.9

- There relative map and geotag locations will be shown on the following pages in order of magnitude of contributions.
- I will also include some notes derived from my personal knowledge of this area that could be of some interest.



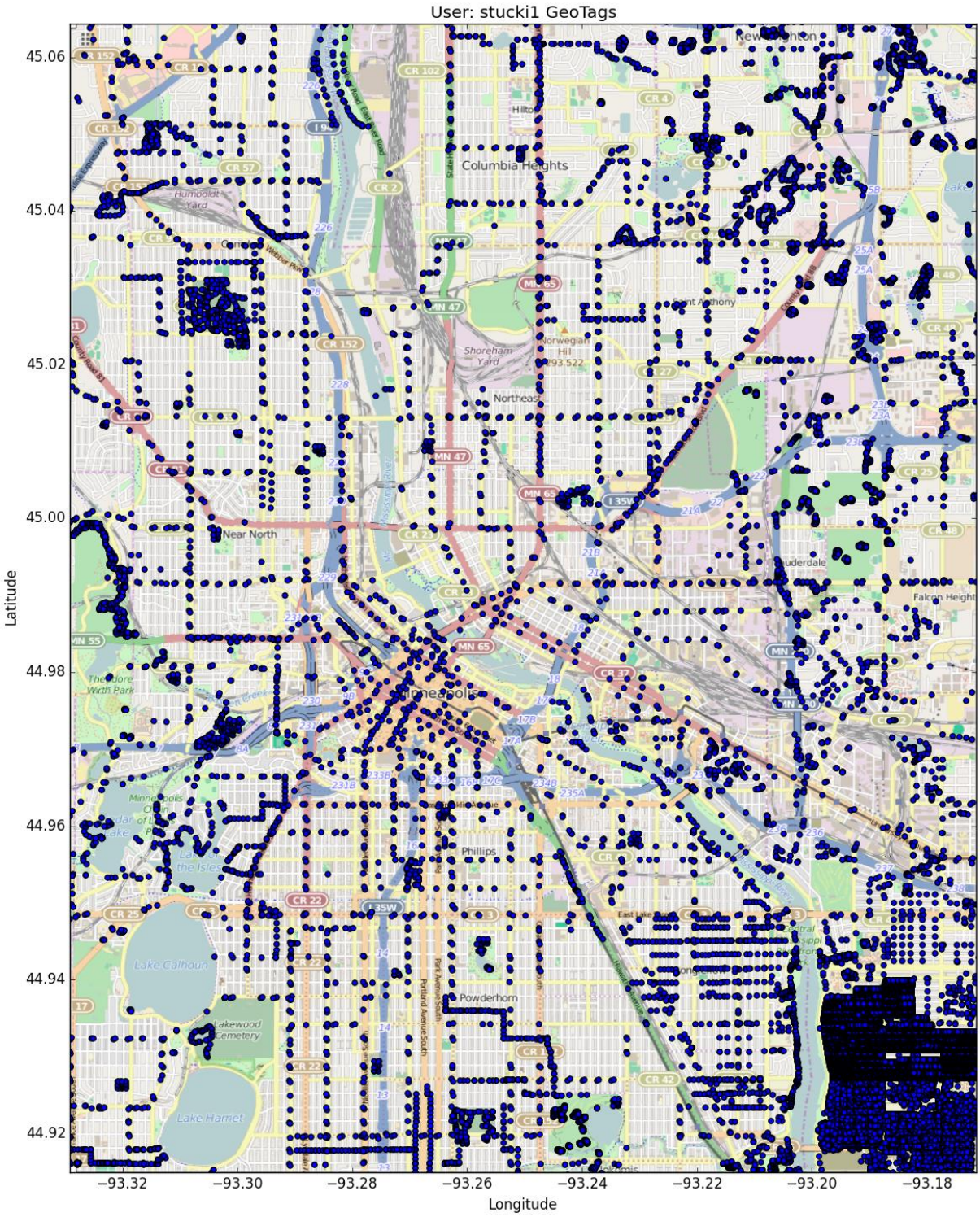
'Mulad' geotagged heavily in downtown Minneapolis, east through UofM and St. Paul, and a south stream through Hiawatha/55. These regions are high-frequency public transportation routes especially the Hiawatha Light Rail which is a primary route to connect the southern suburbs into Minneapolis. This could indicate 'Mulad' depends on public transportation and/or resides in Minneapolis proper.

[back to top]



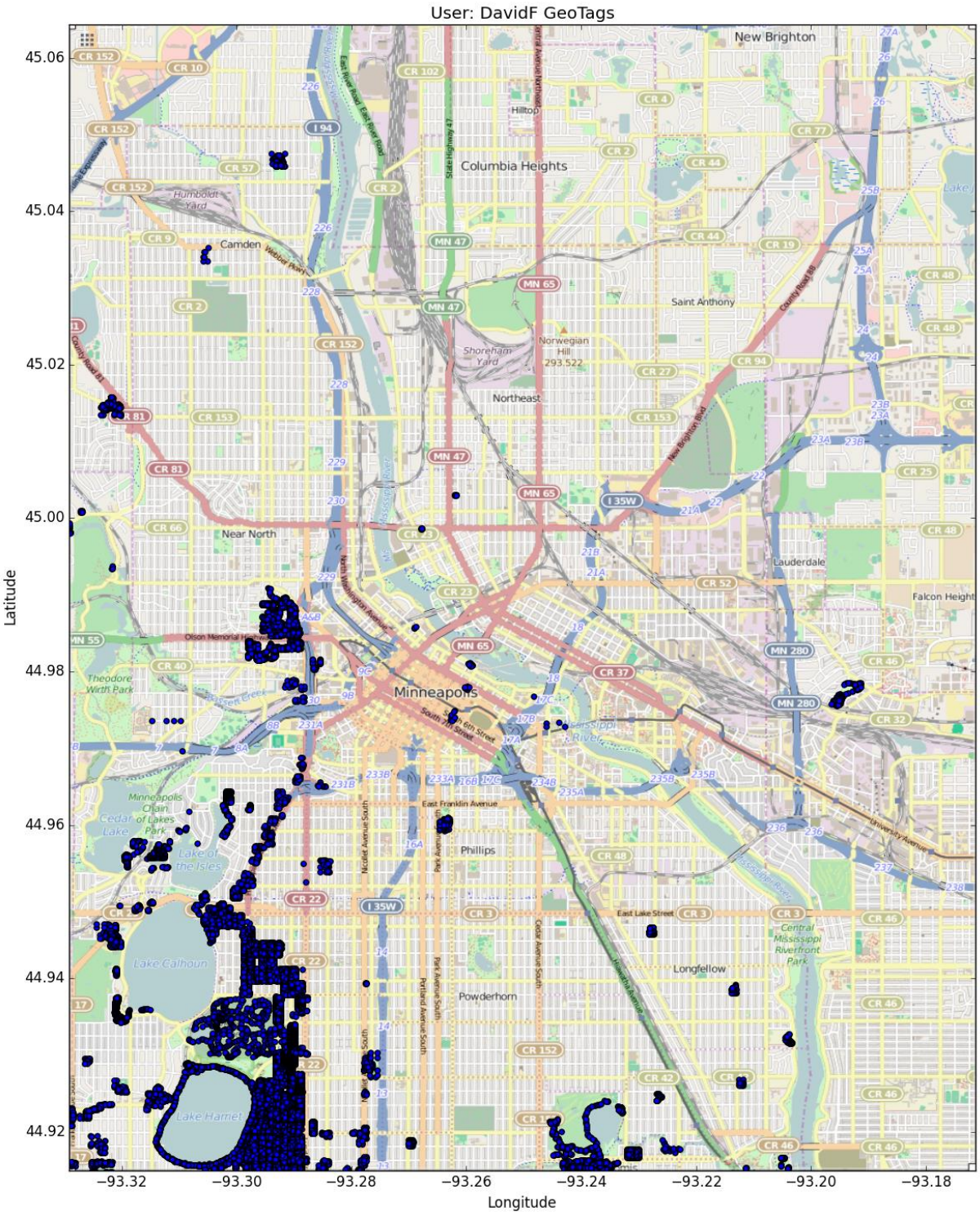
'iandees' geotags are primarily in a region of Minneapolis called 'Uptown' and runs to Northeast Minneapolis. These two regions, especially 'Uptown', are known for a vibrant 'bar scene'. Such a high concentration in the 'Uptown' region could indicate location of home.

[back to top]



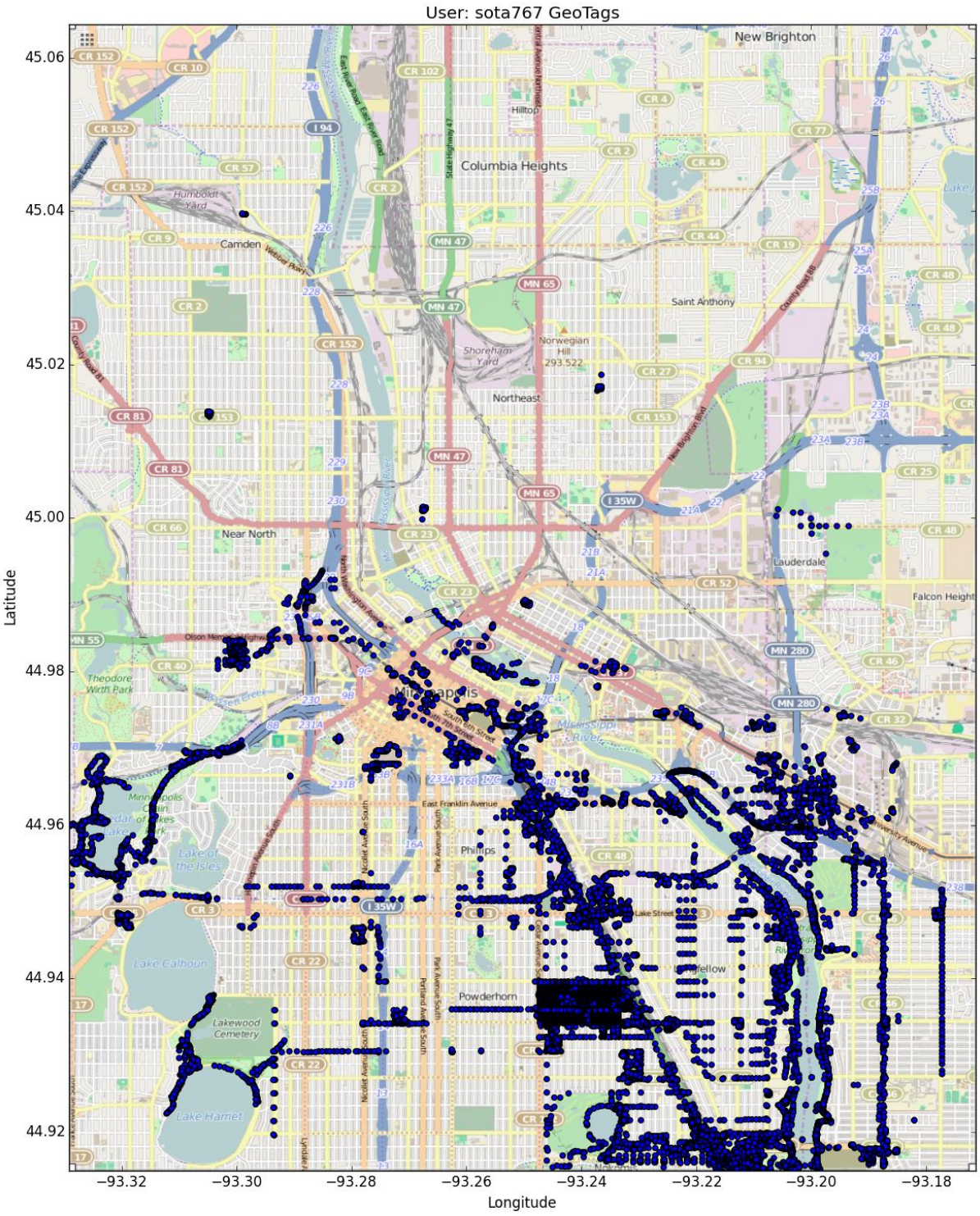
'stucki1's geotag patterns show high concentrations in the St. Paul side and a small concentration Northwest in Minneapolis with sparse tags on various intersections throughout the area. The intersection-tags could indicate 'stucki1' simply tagging while driving and the high-concentrations could indicate a home area and a work area.

[back to top]



'DavidF's geotags are highly concentrated around Lake Harriet/Lake Calhoun area. This area has higher than average housing prices and could indicate personal salary. This could also indicate a frequent area of recreation for the user.

[back to top]



'sota767's geotags are indicative of a home region around the Minnehaha falls area. User could also be a student since there are high concentrations in the UofM campus area. His tagging patterns are concentrated in the Hiawatha Light Rail system and along North-South public transportation routes which could indicate a reliance on public transport.

[back to top]

4. Conclusions

[back to top]

- I am inclined to presume that the top 5 geotaggers were hired by OSM or some other geographical tagging association. What leads me to this hypothesis is that these users tagged in specialized areas of the region that, for the most part, did not overlap and, between them, encompass ~79% of the total tags.
- I would also like to recommend a 'fresh' start with the geotags. Scanning through the documents I found a lot of key value pairs that did not categorize well and most seemed to be in a random format. These outliers are sparsely dispersed throughout the data and do not occur with a great enough frequency to warrant any sort of pattern. They are simply clutter. These values should be removed and a stricter form of geotagging should be implemented so future tags adhere to a certain standard.
- The positional data, once overlaid on top of the data region yielded some intriguing opportunities. If I were to utilize a larger area, I am sure I would be able to extract some meaningful patterns in users' geotagging tendencies relative to known geographical features. Many of the top users tagged in highly populated areas of the region leaving some 'holes'

Codes

[back to top]

```
# The xml2dict() is the main function which takes in an OSM file and turns it into a dictionary that can
# be accessed by normal dicitonary queries.  Only specified elements of the file will be used to populate
# The dictionary.
def xml2dict(filename):
    import time
    import re
    import xml.etree.ElementTree as ET
    from pygeocoder import Geocoder
    expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane", "Road",
            "Trail", "Parkway", "Commons"]
    street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)
    street_mapping = {"St": "Street",
                          "St.": "Street",
                          "Ave" : "Avenue",
                          "Rd." : "Road",
                          "Rd" : "Road",
                          "SE" : "Southeast",
                          "S.E." : "Southeast",
                          "NE" : "Northeast",
                          "S" : "South",
                          "Dr" : "Drive",
                          "Rd/Pkwy" : "Road/Parkway",
                          "Ln" : "Lane",
                          "Dr." : "Drive",
                          "E" : "East",
                          "Pl" : "Plain",
                          "ne" : "Northeast",
                          "NW" : "Northwest",
                          "Ave." : "Avenue",
                          "N." : "North",
                          "W" : "West",
                          "Pkwy" : "Parkway",
                          "Ter" : "Terrace",
                          "Pky" : "Parkway",
                          "SW" : "Southwest",
                          "N" : "North",
                          "Blvd" : "Boulevard"}
    city_mapping = {"Minneapolis, MN" : "Minneapolis",
                    "St. Anthony" : "Saint Anthony",
                    "St. Paul" : "Saint Paul",
                    "St Anthony" : "Saint Anthony",
                    "St Paul" : "Saint Paul"}
    CREATED = [ "version", "changeset", "timestamp", "user", "uid"]
    address = ['addr:unit', 'addr:full','addr:housenumber', 'addr:postcode', 'addr:street',
                'addr:city', 'addr:state', 'addr:country',
                'addr:suite', 'addr:housename']
    # The shape_element() function takes in iterations of 'ET.iterparse'.  Each iteration is a line in the OSM xml file
    # and will move forward in the function if the line has a tag that equals 'node' or 'way'.
    def shape_element(element):
        # First a dictionary is created.  This dictionary will recieve the cleaned data from the OSM file
```

```
        # and then returned at the end.
        node = {}
        # First we check if the tag is either 'node' or 'way'.
        if element.tag == "node" or element.tag == "way" :
            # Then we create a key called 'type' which equals either 'node' or 'way'.
            node['type'] = element.tag
            # Then we check every attribute in the line.
            # If an attribute matches any value in CREATED, we create a dictionary call
ed 'created'.
            # The loop breaks immedietly after the match and creation of the dictionary
.
            for i in element.attrib.keys():
                if i in CREATED:
                    node['created'] = {}
                    break
            # Once the 'created' dictionary is created.  The attributes which match the
 values in CREATED and their
            # respective values are added to the dictionary 'created'.  Attributes that
 equal 'lat' or 'lon' are skipped.
            # The rest of the attributes are made into keys in the node dictionary with
 values being their respective
            # value pair.
            for i in element.attrib.keys():
                if i in CREATED:
                    node['created'][i] = element.attrib[i]
                elif i == 'lon' or i == 'lat':
                    continue
                else:
                    node[i] = element.attrib[i]
            # If 'lat' is one of the attributes in the element, it is assumed it has a
'lon' pair as well.
            # These two values are combined in a list [Latitude, Longitude].  A key, 'p
os', is created in the
            # node dictionary with its value being the [Latitude, Longitude] list.
            if 'lat' in element.attrib.keys():
                node['pos'] = [float(element.attrib['lat']), float(element.attrib['lon'
])]
            # The following scans the 'k' values of the subelements of 'node' or 'way'
tags and creates a dictionary
            # depending on the matches.
            # For example, if scanning the 'k' values reveal a value that starts with '
gnis:'
            # the code will create a dictionary 'gnis' inside the node dictionary
            for i in element:
                if 'k' in i.attrib:
                    if i.attrib['k'] in address:
                        node['address'] = {}
                    elif i.attrib['k'].startswith('metcouncil:'):
                        node['metcouncil'] = {}
                    elif i.attrib['k'].startswith('tiger:'):
                        node['tiger'] = {}
                    elif i.attrib['k'].startswith('metrogis:'):
                        node['metrogis'] = {}
                    elif i.attrib['k'].startswith('umn:'):
                        node['umn'] = {}
                    elif i.attrib['k'].startswith('gnis:'):
                        node['gnis'] = {}
            # Some of the Latitudenal and Longitudenal data are not located in the pare
nt levels of 'node' and 'way'
            # In particular, if the subelements' 'k' values include items that start wi
th 'umn:', the corresponding
```

```
                    # locational data is located in 'umn:BuildingCenterXYLatitude' and 'umn:BuildingCenterXYLongitude'
                    # The following code extracts those values and includes them in the 'pos' list.
                    for i in element:
                        if 'k' in i.attrib:
                            if i.attrib['k'] in ['umn:BuildingCenterXYLatitude', 'umn:BuildingCenterXYLongitude']:
                                node['pos'] = []
                                break
                    for i in element:
                        if 'k' in i.attrib:
                            if i.attrib['k'] == 'umn:BuildingCenterXYLatitude':
                                node['pos'].append(float(i.attrib['v']))
                                break
                    for i in element:
                        if 'k' in i.attrib:
                            if i.attrib['k'] == 'umn:BuildingCenterXYLongitude':
                                node['pos'].append(float(i.attrib['v']))
                                break
                    # As instructed in lesson 6, some of the subelements whose tags are 'ref' need to be grouped
                    # under the list 'node_refs'.  The following creates that list.
                    for i in element:
                        if 'ref' in i.attrib:
                            node['node_refs'] = []
                    # The following code populates the previously created groups in the node dictionary
                    # by scanning the subelements and tests the existence of the necessary value-pairs.
                    # If the necessary value-pairs exist, the 'k' value prefix is stripped and added
                    # as a key under the respective group.  Then the 'v' is added as teh value/
                    # For example. 'k' = 'addr:street', 'v' = 'dorland' is added as
                    # 'street' : 'dorland'
                    for i in element:
                        if 'k' in i.attrib:
                            if i.attrib['k'] in address:
                                if i.attrib['k'] == 'addr:city':
                                    if i.attrib['v'] in city_mapping.keys():
                                        node['address'][re.sub('addr:', '', i.attrib['k'])] = city_mapping[i.attrib['v']]
                                    else:
                                        node['address'][re.sub('addr:', '', i.attrib['k'])] = i.attrib['v']
                                else:
                                    node['address'][re.sub('addr:', '', i.attrib['k'])] = i.attrib['v']
                            elif i.attrib['k'].startswith('metcouncil:'):
                                node['metcouncil'][re.sub('metcouncil:', '', i.attrib['k'])] = i.attrib['v']
                            elif i.attrib['k'].startswith('tiger:'):
                                node['tiger'][re.sub('tiger:', '', i.attrib['k'])] = i.attrib['v']
                            elif i.attrib['k'].startswith('metrogis:'):
                                node['metrogis'][re.sub('metrogis:', '', i.attrib['k'])] = i.attrib['v']
                            elif (i.attrib['k'].startswith('umn:') and
                                  i.attrib['k'] not in ['umn:BuildingCenterXYLatitude', 'umn:BuildingCenterXYLongitude']):
```

```
                                node['umn'][re.sub('umn:', '', i.attrib['k'])] = i.attri
    b['v']
                            elif i.attrib['k'].startswith('gnis:'):
                                node['gnis'][re.sub('gnis:', '', i.attrib['k'])] = i.att
    rib['v']
                            elif ('addr:street:' in i.attrib['k'] or
                                    i.attrib['k'] in ['umn:BuildingCenterXYLatitude', 'u
    mn:BuildingCenterXYLongitude']):
                                continue
                            else:
                                # All the remaining value pairs are then added with the
    'k' value being the key
                                # and the 'v' value being the value of that key
                                node[i.attrib['k']] = i.attrib['v']
                        # Earlier we created the 'node_refs' list inside the node dictio
    nary
                        # The following will add the values to this list.
                        if 'ref' in i.attrib:
                            node['node_refs'].append(i.attrib['ref'])
                    # Finally after the node dictionary is created from the particular i
    teration of ET.iterparse(),
                    # it is returned
                    return node
            # 2 empty lists are created.  We will append each dictionary returned by fee
    ding the iterations of the OSM
            # file into shape_element() into the 'temp' list.  Since some of the iterati
    ons will not contain the tags
            # 'node' and 'way', the shape_element() will return 'None' for these instanc
    es.
            # We will then run throught each iteration of temp and only append the itera
    tions that contains information
            # to the data list.
            temp = []
            data = []
            for _, element in ET.iterparse(filename):
                temp.append(shape_element(element))
            for i in temp:
                if i != None:
                    data.append(i)
            # The following corrects the street values fo the data.  We run through each
     iteration of the data list.
            # If the iteration contains a key 'address', we move forward and check if th
    e key 'address' contains a key
            # 'street'.  If these conditions are met, we check the street value's suffix
     is in the keys of 'street_mapping'.
            # If it is, the code replaces that word for the value of the key it matches.

            for i in data:
                if 'address' in i:
                    if 'street' in i['address']:
                        search = street_type_re.search(i['address']['street'])
                        if search:
                            if street_type_re.search(i['address']['street']).group() in
    street_mapping.keys():
                                data[data.index(i)]['address']['street'] = re.sub(street
    _type_re.search(i['address']['street']).group(),
                                                                                  stre
    et_mapping[street_type_re.search(i['address']['street']).group()],
                                                                                  data
    [data.index(i)]['address']['street'])
```

```
188.            # The following corrects the incorrect postal codes.  We run through each it
    eration of the data list.
189.            # The first 'if' statement checks if the iteration has the keys 'address' an
    d 'pos'
190.            # If so, we proceed to check if the postal code is incorrect.
191.            # If it is, we use the coordinates in the 'pos' list and apply it to the Geo
    coder.reverse_geocode()
192.            # The postal code from that query will replace the incorrect postal code.
193.            #
194.            # The second 'elif' statement is for iterations that have an 'address' but d
    o not have a 'pos' list
195.            # to reverse_geocode() the postal code.
196.            # It instead will use the Geocoder.geocode() function and search specific el
    ements of the 'address'
197.            # key to return the correct postal code.
198.            for i in data:
199.                if 'address' in i and 'pos' in i:
200.                    if 'postcode' in i['address']:
201.                        if len(i['address']['postcode']) < 5 or re.search('[a-zA-
    Z]', i['address']['postcode']):
202.                            results = Geocoder.reverse_geocode(i['pos'][0], i['pos'][1])

203.                            i['address']['postcode'] = str(results.postal_code)
204.                elif 'address' in i and 'pos' not in i:
205.                    if 'postcode' in i['address']:
206.                        if len(i['address']['postcode']) < 5 or re.search('[a-zA-
    Z]', i['address']['postcode']):
207.                            q = ''
208.                            if 'housename' in i['address']:
209.                                q = i['address']['housename'] + ' MN'
210.                            elif 'housenumber' in i['address'] and 'street' in i['addres
    s']:
211.                                q = i['address']['housenumber'] + ' ' + i['address']['st
    reet'] + ' MN'
212.                            results = Geocoder.geocode(q)
213.                            i['address']['postcode'] = str(results.postal_code)
214.            # The following will standardize 2 part postal codes to 1 part postal codes.

215.            # For example, 55404-1234 will be turned to 55404
216.            for i in data:
217.                if 'address' in i:
218.                    if 'postcode' in i['address']:
219.                        if len(i['address']['postcode']) > 5:
220.                            i['address']['postcode'] = i['address']['postcode'][0:5]
221.            # The cleaned and sorted data is ready to be returned.
222.            return data
223.
224.        # The dict2json() simply takes the dictionary produced by xml2dict() and saves i
    t as the specified
225.        # file in .json format
226.        def dict2json(dict, output_file):
227.            import codecs
228.            import json
229.            with codecs.open(output_file, 'w') as fo:
230.                for i in dict:
231.                    fo.write(json.dumps(i) + '\n')
232.            fo.close()
233.
234.
235.
```

```
236.        # The audit_xml() scans through the original OSM file, which is formatted in XML
237.        # and returns a specified dictionary of audits
238.        def audit_xml(filename, form = 'all', value = None):
239.            import xml.etree.ElementTree as ET
240.            # count_attrib() will count the unique values of each element in the file
241.            def count_attrib(filename):
242.                attrib_count = {}
243.                for _, element in ET.iterparse(filename):
244.                    if element.tag in ['node', 'way', 'tag', 'nd']:
245.                        for i in element.attrib.keys():
246.                            if i not in attrib_count.keys():
247.                                attrib_count[i] = {'count' : 1}
248.                            else:
249.                                attrib_count[i]['count'] += 1
250.                return attrib_count
251.            # count_val() will count a specific value within the file
252.            def count_val(x, filename):
253.                k = {}
254.                for _, element in ET.iterparse(filename):
255.                    if element.tag in ['node', 'way', 'tag', 'nd']:
256.                        if x in element.attrib:
257.                            if element.attrib[x] not in k:
258.                                k[element.attrib[x]] = 1
259.                            else:
260.                                k[element.attrib[x]] += 1
261.                return k
262.            # tag_count simply counts the unique tags located in the file
263.            def tag_count(filename):
264.                tags = {}
265.                for _, element in ET.iterparse(filename):
266.                    if element.tag in ['node', 'way', 'tag', 'nd']:
267.                        for i in element:
268.                            if i.tag not in tags:
269.                                tags[i.tag] = 1
270.                            else:
271.                                tags[i.tag] += 1
272.                return tags
273.            # depending on the 'form' and/or 'value' argument, audit_xml() will return t
    he desired
274.            # audit type
275.            if form.lower() == 'all':
276.                data = {}
277.                attrib = count_attrib(filename)
278.                for i in attrib:
279.                    data[i] = count_val(i, filename)
280.                for i in data:
281.                    data[i]['TOTAL'] = attrib[i]['count']
282.                data['TAGS'] = tag_count(filename)
283.                return data
284.            elif form.lower() == 'tags':
285.                return tag_count(filename)
286.            elif form.lower() == 'attributes':
287.                return count_attrib(filename)
288.            elif form.lower() == 'values':
289.                return count_val(v, filename)
290.            else:
291.                if not form or form not in ['all', 'tags', 'attributes', 'values']:
292.                    print "Invalid Audit Type"

1.
```

The following code created the map which represented the individual user's geotag. I basically used 'cbook' and 'imread' to load the image I screenshotted from OSM of my region as the graph background.
[back to top]

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import imread
import matplotlib.cbook as cbook
from pymongo import MongoClient
client = MongoClient()
db = client['maps']
## Top 5 users found from the data statistics are of this document
users = ['Mulad', 'iandees', 'stucki1', 'DavidF', 'sota767']
## Loading only the documents where 'created.user' match the users list
q = db.minneapolis.find({'created.user' : {'$in' : users}})
temp = []
for i in q:
    if 'pos' in i:
        temp.append(i)
## Creating a dictionary consisting of users as the key and their
## Latitude and Longitude in a list for values of that key
data = {}
for i in temp:
    if i['created']['user'] not in data:
        data[i['created']['user']] = []
    data[i['created']['user']].append(i['pos'])
## Defining which user to create the graph for
user = 'DavidF'
lon = []
lat = []
## Loading that users positional data for latitude and longitude into the
## 'lat' list and 'lon' list
for i in data[user]:
    lon.append(i[1])
    lat.append(i[0])
## Creating the boundaries of the graph from the original XML max/min lat/lon
extent = [-93.3292, -93.1713, 44.915, 45.0644]
## loading the image from the screen shot used in the beginning of this report
datafile = cbook.get_sample_data('map.png')
img = imread(datafile)
## Using 'lat' values as the y-axis and 'lon' values as the x-axis, we will
## create a scatter plot
plt.scatter(lon,lat,zorder=0.2)
## Labeling the graph
plt.ylabel('Latitude')
plt.xlabel('Longitude')
plt.title('User: {} GeoTags'.format(user))
## Producing the graph with the aspect ratio being the original screenshot's
## pixel ratio and the extent of its limits the boundaries stated in the list
## 'extent'
plt.imshow(img, zorder = 0, extent = extent, aspect = (1235/float(924)))
```

Lesson 6 Codes

6.1 Iterative Parsing [back to top]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Your task is to use the iterative parsing to process the map file and
find out not only what tags are there, but also how many, to get the
feeling on how much of which data you can expect to have in the map.
The output should be a dictionary with the tag name as the key
and number of times this tag can be encountered in the map as value.

Note that your code will be tested with a different data file than the 'example.osm'
"""
import xml.etree.ElementTree as ET
import pprint

def count_tags(filename):
        # YOUR CODE HERE
        tree = ET.parse(filename)
        data = []
        dic = {}
        tags = tree.getiterator()
        for i in range(len(tags)):
            data.append(tags[i].tag)

        utags = list(set(data))
        for i in utags:
                dic[i] = 0

        for i in range(len(tags)):
            dic[tags[i].tag] += 1
        return dic


def test():

    tags = count_tags('example.osm')
    pprint.pprint(tags)
    assert tags == {'bounds': 1,
                    'member': 3,
                    'nd': 4,
                    'node': 20,
                    'osm': 1,
                    'relation': 1,
                    'tag': 7,
                    'way': 1}


if __name__ == "__main__":
    test()
```

6.2 Data Model [back to top]

We would like to get the data into a database.
But we need to decide on a data model!
Which of the following models would you prefer
for the following data node?


```
{
"_id": "2406124091",
"visible":"true",
"version":"2",
"changeset":"17206049",
"timestamp":"2013-08-03T16:43:42Z",
"user":"linuxUser16",
"uid":"1219059",
"pos": [41.9757030, -87.6921867],
"tags":[ {"k": "addr:housenumber", "v": "5157"},
         {"k": "addr:postcode", "v": "60625"},
         {"k": "addr:street", "v": "North Lincoln Ave"},
         {"k": "amenity", "v": "restaurant"},
        ]
}
```

```
{
"_id": "2406124091",
"visible":"true",
"created": {
          "version":"2",
          "changeset":"17206049",
          "timestamp":"2013-08-03T16:43:42Z",
          "user":"linuxUser16",
          "uid":"1219059"
        },
"pos": [41.9757030, -87.6921867],
"address": {
          "housenumber": "5157",
          "postcode": "60625",
          "street": "North Lincoln Ave"
        },
"amenity": "restaurant",
"cuisine": "mexican",
"name": "La Cabana De Don Luis",
"phone": "1 (773)-271-5176"
}
```

6.3 Tag Types [back to top]

```
import xml.etree.ElementTree as ET
import pprint
import re

lower = re.compile(r'^([a-z]|_)*$')
lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')


def key_type(element, keys):
    if element.tag == 'tag':
        k = element.attrib['k']
        search = lower.search(k)
        if search:
            keys['lower'] += 1
        else:
            search = lower_colon.search(k)
            if search:
                keys['lower_colon'] += 1
            else:
                search = problemchars.search(k)
                if search:
                    keys['problemchars'] += 1
                else:
                    keys['other'] += 1
        # YOUR CODE HERE
        return keys
        pass

    return keys


def process_map(filename):
    keys = {"lower": 0, "lower_colon": 0, "problemchars": 0, "other": 0}
    for _, element in ET.iterparse(filename):
        keys = key_type(element, keys)

    return keys


def test():
    # You can use another testfile 'map.osm' to look at your solution
    # Note that the assertions will be incorrect then.
    keys = process_map('example.osm')
    pprint.pprint(keys)
    assert keys == {'lower': 5, 'lower_colon': 0, 'other': 1, 'problemchars': 1}


if __name__ == "__main__":
    test()
```

6.4 Exploring Users [back to top]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import xml.etree.ElementTree as ET
import pprint
import re
"""
Your task is to explore the data a bit more.
The first task is a fun one - find out how many unique users
have contributed to the map in this particular area!

The function process_map should return a set of unique user IDs ("uid")
"""

def get_user(element):
    return


def process_map(filename):
    users = set()
    for _, element in ET.iterparse(filename):
        att = element.attrib
        if 'uid' in att:
            users.add(att['uid'])


    return users


def test():

    users = process_map('example.osm')
    pprint.pprint(users)
    assert len(users) == 6


if __name__ == "__main__":
    test()
```

[back to top]

6.5 Improving Street Names [back to top]

```
"""
Your task in this exercise has two steps:

- audit the OSMFILE and change the variable 'mapping' to reflect the changes needed to fix
    the unexpected street types to the appropriate ones in the expected list.
    You have to add mappings only for the actual problems you find in this OSMFILE,
    not a generalized solution, since that may and will depend on the particular area you are auditing.
- write the update_name function, to actually fix the street name.
    The function takes a string with street name as an argument and should return the fixed name
    We have provided a simple test so that you see what exactly is expected
"""
import xml.etree.cElementTree as ET
from collections import defaultdict
import re
import pprint

OSMFILE = "example.osm"
street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)


expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane", "Road",
            "Trail", "Parkway", "Commons"]

# UPDATE THIS VARIABLE
mapping = { "St": "Street",
            "St.": "Street",
            "Ave" : "Avenue",
            "Rd." : "Road"}


def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)


def is_street_name(elem):
    return (elem.attrib['k'] == "addr:street")


def audit(osmfile):
    osm_file = open(osmfile, "r")
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):

        if elem.tag == "node" or elem.tag == "way":
            for tag in elem.iter("tag"):
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])

```

```
    return street_types


def update_name(name, mapping):

    # YOUR CODE HERE
    if (street_type_re.search(name).group() not in expected and
        street_type_re.search(name).group() in mapping.keys()):
        name = re.sub(street_type_re.search(name).group(),
                      mapping[street_type_re.search(name).group()],
                      name)

    return name


def test():
    st_types = audit(OSMFILE)
    assert len(st_types) == 3
    pprint.pprint(dict(st_types))

    for st_type, ways in st_types.iteritems():
        for name in ways:
            better_name = update_name(name, mapping)
            print name, "=>", better_name
            if name == "West Lexington St.":
                assert better_name == "West Lexington Street"
            if name == "Baldwin Rd.":
                assert better_name == "Baldwin Road"


if __name__ == '__main__':
    test()
```

[back to top]

6.6 Preparing for Database [back to top]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import xml.etree.ElementTree as ET
import pprint
import re
import codecs
import json

lower = re.compile(r'^([a-z]|_)*$')
lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')

CREATED = [ "version", "changeset", "timestamp", "user", "uid"]
address = ['addr:housenumber', 'addr:postcode', 'addr:street', 'addr:city']

def shape_element(element):
    node = {}
    if element.tag == "node" or element.tag == "way" :
        # YOUR CODE HERE
        node['type'] = element.tag
        for i in element.attrib.keys():
            if i in CREATED:
                node['created'] = {}
                break
        for i in element.attrib.keys():
            if i in CREATED:
                node['created'][i] = element.attrib[i]
            elif i == 'lon' or i == 'lat':
                continue
            else:
                node[i] = element.attrib[i]
        if 'lat' in element.attrib.keys():
            node['pos'] = [float(element.attrib['lat']), float(element.attrib['lon'])]

        for i in element:
            if 'k' in i.attrib:
                if i.attrib['k'] in address:
                    node['address'] = {}
                    break
        for i in element:
            if 'ref' in i.attrib:
                node['node_refs'] = []
        for i in element:
            if 'k' in i.attrib:
                if i.attrib['k'] in address:
                    node['address'][re.sub('addr:', '', i.attrib['k'])] = i.attrib['v']

                elif 'addr:street:' in i.attrib['k']:
                    continue
                else:
                    node[i.attrib['k']] = i.attrib['v']
            if 'ref' in i.attrib:
                node['node_refs'].append(i.attrib['ref'])

        return node
    else:
```

```
        return None


def process_map(file_in, pretty = False):
    # You do not need to change this file
    file_out = "{0}.json".format(file_in)
    data = []
    with codecs.open(file_out, "w") as fo:
        for _, element in ET.iterparse(file_in):
            el = shape_element(element)
            if el:
                data.append(el)
                if pretty:
                    fo.write(json.dumps(el, indent=2)+"\n")
                else:
                    fo.write(json.dumps(el) + "\n")
    return data

def test():
    # NOTE: if you are running this code on your computer, with a larger dataset,
    # call the process_map procedure with pretty=False. The pretty=True option adds
    # additional spaces to the output, making it significantly larger.
    data = process_map('example.osm', True)
    #pprint.pprint(data)

    correct_first_elem = {
        "id": "261114295",
        "visible": "true",
        "type": "node",
        "pos": [41.9730791, -87.6866303],
        "created": {
            "changeset": "11129782",
            "user": "bbmiller",
            "version": "7",
            "uid": "451048",
            "timestamp": "2012-03-28T18:31:23Z"
        }
    }
    assert data[0] == correct_first_elem
    assert data[-1]["address"] == {
                                    "street": "West Lexington St.",
                                    "housenumber": "1412"
                                      }
    assert data[-
1]["node_refs"] == [ "2199822281", "2199822390",  "2199822392", "2199822369",
                                    "2199822370", "2199822284", "2199822281"]


       if __name__ == "__main__":
           test()
```

[back to top]