Project 3. Data Wrangling of Nashville_sm

March 14, 2017

1 Data Wrangling

This project illustrates various stages of data wrangling and done as a part of the coursework for data analyst nano degree from Udacity.

1.1 Description of Data

The data chosen for this project is downloaded as a OSM XML zip file (19Mb - Unzipped: 295Mb) from https://mapzen.com/data/metro-extracts/metro/nashville_tennessee/. Nashville area in Tennessee is chosen for the limited computing power of the machine. Before inserting the cleaned bulk data into database to perform queries, an initial screening is done follwed by auditing specific fields and cleaning the data.

1.2 Inital Screening

The initial screening includes programtically screening data to understand the frequency of different tags and tag-attributes in `XML` file. Iterative parsing of `XML` is performed using the package `xml.etree.cElementTree` by looping through each tags. The function `count_tags` prints the frequency of the tags and `tag_attrib` prints the various attributes in each tag as a dictionary of sets.

```
In [12]: """
         Counting the number of various tags in the file.
         """
         # filename=open("test.xml")
         filename=open("nashville_tennessee.osm","r")

         def count_tags(filename):
             tags = {}
             # iterative parsing of tags
             for event, elem in ET.iterparse(filename, events=("start",)):
                 #increment for tags
                 if elem.tag not in tags:
                     tags[elem.tag] = 1
                 else:
                     tags[elem.tag] += 1
             return tags
```

```
count_tags(filename)
```

```
Out[12]: {'bounds': 1,
          'member': 16816,
          'nd': 1499455,
          'node': 1317086,
          'osm': 1,
          'relation': 1897,
          'tag': 919289,
          'way': 135617}
```

```
In [3]: """
        The various attributes for each tag
        """
        filename=open("nashville_tennessee.osm","r")

        def tag_attribs(filename):
            tag_attrib = defaultdict(set)
            for event, elem in ET.iterparse(filename, events=("start",)):
                for e in elem.attrib:
                    tag_attrib[elem.tag].add(e)
            return tag_attrib
        audit_tag=tag_attribs(filename)
        pprint.pprint(audit_tag)
```

```
defaultdict(<class 'set'>,
            {'bounds': {'minlat', 'maxlon', 'minlon', 'maxlat'},
             'member': {'type', 'role', 'ref'},
             'nd': {'ref'},
             'node': {'changeset',
                      'id',
                      'lat',
                      'lon',
                      'timestamp',
                      'uid',
                      'user',
                      'version'},
             'osm': {'version', 'timestamp', 'generator'},
             'relation': {'changeset',
                          'id',
                          'timestamp',
                          'uid',
                          'user',
                          'version'},
             'tag': {'v', 'k'},
             'way': {'changeset', 'version', 'uid', 'id', 'user', 'timestamp'}})
```

1.3 Auditing Data

The various tags and attributes are inspected further to find that most of the information is stored in the tag "tag" as `keys["k"]` and `values["v"]`. Function `unique_keys` is defined to aggregate various key values to decide which data has to be cleaned to make a data structure to be passed to database. 3 other major types of tags are found to be `node, way and relation` with each having attributes related to its creation. Since printing the lengthy outputs are impractial, `write_dict` function is defined to print the output to a `.txt` file in the current directory.

Auditing of data reveals that

1. Some `Postcodes` are prefixed with `"TN"` and has to be cleaened to a neat 5-digit values
2. Phone numbers are in various formats with some including extensions and some lacking area code. In the absense of area code, the phone number has to be rejected. Those with extension numbers has to be stripped from the end.
3. `State` for nashville has to be cleaned to `"TN"` for uiformity
4. `Amenities` can be passed on to database with out cleaning
5. `City` names have to be cleaned for uniformity, some city names with lower caps in the begining of the sentence has to be changed to uppercase.
6. `Country` has to be uniformly cleaned to `"USA"`
7. `Neighbourhood` fields doesn't have to be changed
8. `Street` names has to be fixed by changing the abbreviated version to the longform, eg: `"Ave."` to `"Avenue"`, `"Dr."` to `"Drive"`
9. There is not enough validated daa to fix `housenumbers` and `housenames`, hence not fixed.

```
In [44]: """
         Auditing various fields in the data set to find inconsistencies
         """

         filename=open("nashville_tennessee.osm","r")
         def audit_tags(filename):
             #Regular Expressions
             pincode=re.compile(r'^([0-9]{5})(?:-[0-9]{4})?$')
             phone=re.compile(r"^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$")
             street_name_re=re.compile(r"\b\S+\.?$", re.IGNORECASE)

             # fields in tags to be audited
             POST_FIELDS=["addr:postcode", "postal_code"]
             CITIES_FIELDS = ["addr:city", "is_in:city"]
             PHONE_FIELDS=["Phone","phone"]#,'contact:phone'],
                           #'telephone','communication:mobile_phone',
                           #'phone:local', 'disused:phone']
             STREET_FIELDS=['addr:street','destination:street']
             EXPECTED_STREET_NAMES = ["Avenue", "Court", "Lane", "Boulevard", "Drive",
                                      "Court", "Place","Road","Parkway","Circle",
                                      "South","North","Highway","Trail","Terrace",
                                      "Square", "Pike","Alley","Street","Trace",
                                      "Bypass","Way","Fork", "Plaza","Broadway",
                                      "Loop", "Cove","Flagpole", "Foxborough",
                                      "Foxland", "Center", "Hollow","East","Heights",
```

```
                          "Landing", "Springs","Hills", "Mission"]
bad_postcodes=set()
bad_states=set()
cities=set()
countries=set()
bad_phones=set()
good_phones=set()
bad_street_names=defaultdict(set)
amenities=set()
names=set()

for event, elem in ET.iterparse(filename, events=("start",)):
    if elem.tag == "tag":
        if elem.attrib["k"] in POST_FIELDS:
            code=elem.attrib["v"]
            m=pincode.match(code)
            if not m:
                bad_postcodes.add(code)
        if elem.attrib["k"]=="addr:state" and elem.attrib["v"]!="TN":
            bad_states.add(elem.attrib["v"])
        if elem.attrib["k"]=="amenity":
            amenities.add(elem.attrib["v"])
        if elem.attrib["k"] in CITIES_FIELDS:
            cities.add(elem.attrib["v"])
        if elem.attrib["k"]=="addr:country":
            countries.add(elem.attrib["v"])
        if elem.attrib["k"]=="name":
            names.add(elem.attrib["v"])
        if elem.attrib["k"] in PHONE_FIELDS:
            phone_num=elem.attrib["v"].lstrip("+1- ")
            m=re.match(phone,phone_num)
            if m:
                good_phones.add(m.groups())
            if not m:
                bad_phones.add(phone_num)
        if elem.attrib["k"] in STREET_FIELDS:
            m=street_name_re.search(elem.attrib["v"])
            if m:
                street_type=m.group()
                if street_type not in EXPECTED_STREET_NAMES:
                    bad_street_names[street_type].add(elem.attrib["v"])
```

1.4 Cleaning and Inserting to Database

Various functions are defined to clean the fields identified in data through auditing and the following functions are performed. An instance of `mongodb` has to be running before executing the code below.

1. `Phone` number is extracted as a 3-tuple and phone numbers with missing data is discarded.
2. Zip codes are cleaned to a 5-digit sequence ensure data integrity.
3. Street names are cleaned if not in the expected names, to use the longform street name. A dictionary is used for mapping the problematic fields to required clean fields.
4. `Country` and state fields are changed if address field exists.
5. `City` names and street names are cleaned based on a mapping dictionary.
6. `References` to other tags are added as a list
7. `address`, `created` and `ref` fields are added only if exists.

Cleaned data is made into a nested data structure with fields grouped as address, created etc. The structure is defined in function `shape_data`. Data is then passed to a JSON file using `data_to_json`. Finally a database `openstreetmap` is fetched in the `test` and `.json` is inserted as a collection.

```
In [2]: """
        Functions used to clean various fields.
        Take in data and returns cleaned values.
        """

        def clean_phone(number):
            # returns phone numbers after cleaning as 3-tuple: (xxx,xxx,xxxx)
            # discarded numbers are printed
            try:
                phone_re=re.compile(r"^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$")
                m=phone_re.match(number.lstrip("+1- ").strip())
                phone="1 ({0})-{1}-{2}".format(m.groups(0)[0],m.groups(0)[1], m.groups(0)[2])
                return phone
            except AttributeError:
                print("From clean_phone, Bad number\t:",number) # Uncomment to print bad phone
                pass
```

```
In [3]: """
        Cleaning data to pass to database
        =================================
        A general structure of data:
           {
            'address': {'neighbourhood': ...
                        'country': 'USA',
                        'state': 'TN',
                        'street': 'Old Rocky Fork'},
             'created': {'changeset': '4470981',
                         'timestamp': '2010-04-19T18:58:26Z',
                         'uid': '270262',
                         'user': 'Ab Ye',
                         'version': '1'},
             'e_id': '702177640',
             'name': 'Nolensville Ball Park',
             'pos': [35.9544391, -86.6665184],
```

```
    'type': 'node'
    'phone' : ["XXX","XXX","XXXX"]
    'amenity': ""
    'ref':[345534,534534,534534]
   }
    # references are added from  <"nd"> for <"way"> && <"member"> for <"relation">

"""


def shape_data(element):
    if element.tag in ["relation", "way", "node"]:
        data={}
        data["type"]=element.tag
        data["e_id"]=element.attrib["id"]

        created={}
        for field in [ "version", "changeset", "timestamp", "user", "uid"]:
            created[field]=element.attrib[field]
        if created!={}:
            data["created"]=created

        if element.tag=="node": # add position as list : pos=[lat,lon]
            pos=[0,0]
            pos[0]=float(element.attrib["lat"])
            pos[1]=float(element.attrib["lon"])
            data["pos"]=pos

        address={} # initializing address field
        for elem in element.iter("tag"):
            if elem.attrib["k"]=="addr:postcode": #add cleaned postcode
                postcode=clean_pincode(elem.attrib["v"])
                if postcode:
                    address["postcode"]=postcode
            if elem.attrib["k"]=="addr:street": # add street after mapping
                if elem.attrib["v"]:
                    address["street"]=clean_streetnames(elem.attrib["v"])
            if elem.attrib["k"]=="addr:city": # add city after mapping
                if elem.attrib["v"]:
                    address["city"]=clean_city(elem.attrib["v"])
            if elem.attrib["k"] in ["Phone","phone"]:# adding phone number
                phone = clean_phone(elem.attrib["v"])
                if phone:
                    data["phone"]=phone

        if address != {}: # Append address if not empty
            address["state"]="TN" # Adding state field
            address["country"]="USA" #Adding country field
```

```
                data["address"]=address

            ref=[]
            for member in element.iter("member"):
                ref.append(member.attrib["ref"])
            if ref!=[] and element.tag=="relation":
                data["ref"]=ref

            for nd in element.iter("nd"):
                ref.append(nd.attrib["ref"])
            if ref!=[] and element.tag=="way":
                data["ref"]=ref

            return data

    def get_db(db_name):
        """ initiating client and returns db from the Case Study scripts"""
        from pymongo import MongoClient
        client = MongoClient('localhost:27017')
        db = client[db_name]
        return db

    def test():
        file_in=open("nashville_tennessee.osm","r")
        file_out="cleaned_data_{0}.json".format(file_in.name[:10])

    #     Generating JSON data from osm file
        cleaned_data=data_to_json(file_in, file_out)
        db=get_db("openstreetmap")
```

2 Performing Queries

After inserting data into the database, queries are performed to answer various. `get_db` is defined to access the data basto perform needed queries.

2.1 Data Overview

File Sizes

- nashville_tennessee.osm.....294.8Mb
- cleaned_data_nashville_.json.....312.4Mb

This section contains basic statistics about the dataset and the MongoDB queries used to gather them. The required queries to meet the specifiation of project is followed by additional suggestions to improve the dataset and additional queries.

```
In [102]: # total number of entries in the nashville collection
          total_queries=db.nashville.find().count()
```

```
print("Total number of entries in the collection Nashville\t:", total_queries)
# total number of nodes in the nashville collection
nodes=db.nashville.find({"type":"node"}).count()
print("Total number of nodes in Nashville\t:", nodes)
# total number of nodes in the nashville collection
way=db.nashville.find({"type":"way"}).count()
print("Total number of ways in Nashville\t:", way)
# number of unique users in the nashville collection
un=len(db.nashville.distinct("created.user"))
print("Total number of unique users\t\t:", un)
# Top contributing user
pipeline1 = [
        {"$group":{"_id":"$created.user",
                   "count":{"$sum":1}}},
        {"$sort" :{"count":-1}},
        {"$limit":1}
        ]
top_user=db.nashville.aggregate(pipeline1)["result"][0]["_id"]
top_user_count=db.nashville.aggregate(pipeline1)["result"][0]["count"]
print("Top contributing user\t\t:{0}, (no. of entries:{1})".format(top_user, top_user
# Number of contributing users appearing only once (having 1 post)
pipeline2 = [
        {"$group":{"_id":"$created.user",
                   "count":{"$sum":1}}},
        {"$group":{"_id":"$count","num_users":{"$sum":1}}},
        {"$sort" :{"_id":1}},
        {"$limit":1}
        ]
one_users=db.nashville.aggregate(pipeline2)["result"][0]["num_users"]
print("Number of contributing users appearing only once\t:",one_users)
```

```
Total number of entries in the collection Nashville        : 1454600
Total number of nodes in Nashville        : 1317086
Total number of ways in Nashville        : 135617
Total number of unique users                : 1025
Top contributing user                :woodpeck_fixbot, (no. of entries:278223)
Number of contributing users appearing only once        : 187
```

```
In [72]: # Number of entries based on type
         pipeline = [
                {"$group":{"_id":"$type",
                           "count":{"$sum":1}}},
                {"$sort":{"count":-1}}
                ]
         pprint.pprint(db.nashville.aggregate(pipeline))
```

```
{'ok': 1.0,
 'result': [{'_id': 'node', 'count': 1317086},
```

```
{'_id': 'way', 'count': 135617},
{'_id': 'relation', 'count': 1897}]}
```

2.2 Contributor Statistics and Suggestions to improve data content and quality

2.2.1 Contributor Statistics

Top 3 users (`woodpeck_fixbot, Shawn Noble, st1974`) contribute bulk of the data accounting for 37.47% of the data. All the remaining users contribute lesser than 5% of the data suggesting almost equal contributions.

2.2.2 Suggestions to imporve user contribution and fill missing data

1. To further the user contributions, it is suggested that each user get paid a small amount (eg: 2 cents) for each additional data which can be verified by another user(s) (for validity of the data). The monetary cost for payment to contributers can be recovered by giving a paid premium version of the data. The customer with access to premium version can recover the cost paid by extracting value out of premium services to further his bussiness model. Since most of the data is available openly, it is imperative that the premium data services include significant advantage to the entities subscribing the service.

    - **Anticipated problems** :
        - One user can add unverified data(manually or through online-robots/scripts) and the other approve unverified data since they both benefit from the payment.
        - Since amenities can be closed down and moved to another area/building it is also problematic to re-enlist new location and updating old ones.

2. Additionally, points (benefits) to popular games can be offered as an incentive to improve dataset. For example, a tie-up with the game "PokemanGo" can offer free poke-points to gamers who can contribute data. Such a system allows "PokemanGo" improve their services by introducing poke-stops at more convincing places.

3. Grouping the nodes based on the co-ordinates of county boundaries can be used to input the `address:county` field in the data set. Also some of the fields in the dataset has to be validated with publicly available verified data for the accuracy of the given dataset.

    - **Anticipated problems** :
        - Getting verified public data for remote areas or areas which has not been surveyed before can be challenging.

4. Laws can be passed for mandatory listing of essential public services such as police stations, courts, hospitals etc in atleast 2 different maping systems.

```
In [22]: db=get_db("openstreetmap")
         # Top 5 contributing users and their percentage
         pipeline = [
                 {"$group":{"_id":"$created.user",
                            "count":{"$sum":1}}},
```

```
                {"$sort" :{"count":-1}},
                {"$limit":5}
                ]
        pprint.pprint(list(db.nashville.aggregate(pipeline)))
        print('\nwoodpeck_fixbot\t:{0}%'
              '\nShawn Noble\t:{1}%'
              '\nst1974\t\t:{2}%'
              '\nAndrewSnow\t:{3}%'
              '\nRub21\t\t:{4}%'.format(round(278223*100/1454600,2),
                                        round(100*170523/1454600,2),
                                        round(96271/1454600*100,2),
                                        round(100*55606/1454600,2),
                                        round(100*53880/1454600,2)))
```

```
[{'_id': 'woodpeck_fixbot', 'count': 278223},
 {'_id': 'Shawn Noble', 'count': 170523},
 {'_id': 'st1974', 'count': 96271},
 {'_id': 'AndrewSnow', 'count': 55606},
 {'_id': 'Rub21', 'count': 53880}]

woodpeck_fixbot         :19.13%
Shawn Noble         :11.72%
st1974                 :6.62%
AndrewSnow         :3.82%
Rub21                  :3.7%
```

2.3 Additional data exploration using MongoDB queries - Missing address for amenities

Further `db` queries using shows that there are number of entries (8167 entries) lacking an address for the amenity specified. By grouping, we found that `amenity:grave_yard` has maximum number of missing address entries.

```
In [96]: # Number of nodes with amenities but no address
         pipeline = [
                {"$match":{"type":"node",
                           "amenity":{"$exists":1},
                           "address":{"$exists":0}}},
                {"$count":"amenities_with_no_address"}
                ]
         pprint.pprint(db.nashville.aggregate(pipeline))
```

```
{'ok': 1.0, 'result': [{'amenities_with_no_address': 8167}]}
```

```
In [83]: # Number of nodes with amenities(grouped) but no address
         pipeline = [
                {"$match":{"type":"node"}},
```

```
                {"$match":{"amenity":{"$exists":1}}},
                {"$match":{"address":{"$exists":0}}},
                {"$group":{"_id":"$amenity",
                           "count":{"$sum":1}}},
                {"$sort" :{"count":-1}},
                {"$limit":5}
                ]
        pprint.pprint(db.nashville.aggregate(pipeline))
```

```
{'ok': 1.0,
 'result': [{'_id': 'grave_yard', 'count': 3469},
            {'_id': 'place_of_worship', 'count': 2351},
            {'_id': 'school', 'count': 1356},
            {'_id': 'restaurant', 'count': 179},
            {'_id': 'fast_food', 'count': 105}]}
```

2.4 Additional queries

2.4.1 Number of references to nodes in each tag type:

It is found that most of the references to the nodes are associated with `way` type (> 98%) than `relation` type (< 2%).

```
In [8]: # Number of references in each node type
        pipeline = [
                {"$unwind"  :"$ref"},
                {"$group":{"_id":"$type",
                           "count":{"$sum":1}}},
                {"$sort" :{"count":-1}}
                ]
        print(db.nashville.aggregate(pipeline))
        total=1422876+15216
        print('\nway\t\t:{0}%'
              '\nrelation\t:{1}%'.format(round(1422876*100/total,2),
                                        round(15216*100/total,2)))
```

```
<pymongo.command_cursor.CommandCursor object at 0x10646cc50>

way                :98.94%
relation        :1.06%
```

2.4.2 Top 5 amenities:

It is shown that `grave_yard` is the top amenity [largest no. of entries] mentioned in the data set.

```
In [73]: # Top 5 amenities occuring in the data
         pipeline = [
```

```
                {"$match":{"amenity":{"$exists":1}}},
                {"$group":{"_id":"$amenity",
                           "count":{"$sum":1}}},
                {"$sort" :{"count":-1}},
                {"$limit":5}
                ]
        pprint.pprint(db.nashville.aggregate(pipeline))
```

```
{'ok': 1.0,
 'result': [{'_id': 'grave_yard', 'count': 3488},
            {'_id': 'place_of_worship', 'count': 2512},
            {'_id': 'parking', 'count': 1561},
            {'_id': 'school', 'count': 1504},
            {'_id': 'restaurant', 'count': 374}]}
```

2.4.3 Commonly occuring area codes:

The most comonly occuring area codes are identified: 931 and 615.

```
In [21]: # 2 most commonly occuring area codes
         pipeline = [
                {"$unwind"  :"$phone"},
                {"$group":{"_id":"$phone",
                           "count":{"$sum":1}}},
                {"$sort" :{"count":-1}},
                {"$limit":2}
                ]
         print(list(db.nashville.aggregate(pipeline)))
```

```
[{'count': 459, '_id': '931'}, {'count': 263, '_id': '615'}]
```

3 Conclusion

A thorough review of the data wrangling process is shown along with inserting cleaned data into MongoDB database. All required queries are performed along with new additional queries. We have also included certain suggestions on improving the missing data and also to monetize the data.

4 References

1. Python Documentation https://docs.python.org/3.4/index.html
2. ElementTree Overview http://effbot.org/zone/element-index.htm
3. MongoDb Documentation https://docs.mongodb.com/