README.md

Wrangle OpenStreetMap Data

Wrangling operations on an OpenStreetMap dataset for the city of La Rochelle, France.

Part of the Data Analyst Nanodegree by Udacity.

I choosed to wrangle the data for the city of La Rochelle, France. The dataset is available here.

I'm first going to provide a detailed walkthrough of my cleaning strategy and code. We'll then put the pieces together and process the dataset. Finally we'll explore the MongoDB database and run some queries.

1. Auditing & cleaning keys

To get an overview of the structure of the data and determine a cleaning strategy I wrote a script to extract the key ('k') value from nodes and tags' subelements and order them by order of occurrence.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import xml.etree.cElementTree as ET
from collections import defaultdict
from operator import itemgetter

def audit_keys(osmfile):
    osm_file = open(osmfile, 'r')
    tag_elems = defaultdict(int)
    for event, elem in ET.iterparse(osm_file, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            for tag in elem.iter("tag"):
                k = tag.get('k')
                if k:
                    tag_elems[k] += 1
    return sorted(tag_elems.items(), key=itemgetter(1), reverse=True)

tags = audit_street_type("la-rochelle_france.osm")
for i, j in tags:
    print(i)
```

The results are saved to a text file node_and_tags_keys.txt. There are 439 different tags. The 10 most popular are:

```
source
building
wall
highway
name
addr:housenumber
oneway
amenity
service
leisure
```

One surprising thing I found is the number of keys with a structure such as:

```
name:la
name:el
name:en
name:eu
```

In this case, the name value is referenced in different languages.

```
<tag k="name:ca" v="Poitou-Charentes"/>
<tag k="name:eo" v="Puatuo-Ĉarentoj"/>
<tag k="name:eu" v="Poitou-Charentes"/>
<tag k="name:oc" v="Peitau-Charantas"/>
<tag k="name:ru" v="Пуату-Шаранта"/>
```

It would not make sense for this data to be saved as different keys:

```
{
    "name:ca":"Poitou-Charentes",
    "name:eo":"Puatuo-Ĉarentoj",
    "name:eu":"Poitou-Charentes",
    "name:oc":"Peitau-Charantas",
    "name:ru":"Пуату-Шаранта"
}
```

But rather as:

```
{
    "name": {
        "ca":"Poitou-Charentes",
        "eo":"Puatuo-Ĉarentoj",
        "eu":"Poitou-Charentes",
        "oc":"Peitau-Charantas",
        "ru":"Пуату-Шаранта"
    }
}
```

One problem though, is that this pattern is repeated in many other instances, and often with more than two degrees. For example with seamarks:

```
<tag k="seamark:topmark:shape" v="2 cones base together"/>
<tag k="seamark:topmark:colour" v="black"/>
<tag k="seamark:buoy_cardinal:shape" v="spar"/>
<tag k="seamark:buoy_cardinal:colour" v="black;yellow;black"/>
<tag k="seamark:buoy_cardinal:category" v="east"/>
<tag k="seamark:buoy_cardinal:colour_pattern" v="horizontal"/>
```

The data should be processed to look like:

```
{
    "seamark": {
        "topmark": {
            "shape": "2 cones base together",
            "colour": "black",
        },
        "buoy_cardinal": {
            "shape": "spar",
            "colour": ["black", "yellow", "black"],
            "category": "east",
            "colour_pattern": "horizontal",
        }
```

```
    }
}
```

Processing the keys to create nested data structures.

We need to detect ":" caracters within keys, split the keys at each ":" occurrence, and creates nested dictionaries of varying depth that reflect the data structure.

```
def create_nested_dict(data):
    """ Returns a list of nested dictionaries mapping only one value.
          Output example: [{'seamark': {'buoy_cardinal': {'colour_pattern': 'horizontal'}}},
          {'seamark': {'topmark': {'shape': '2 cones base together'}}} ... ]
    """
    nested_dicts = []
    for k, v in data.items():
        if ":" in str(k):
            k_elems = k.split(':')
            node = v
            for k in reversed(k_elems):
                node = {k: node}
        else:
            node = {k: v}
    return node

def update_nested_dict(d, nested_dict):
    """ We use recursion to update a dictionary 'd' with a nested dict
        while avoiding duplicates. By looping this function over the list
        of values extracted by create_nested_dicts we build our final valid nested dictionary.
        Output example:
        {'seamark': {
            'buoy_cardinal': {'colour_pattern': 'horizontal'},
            'topmark': {
                'shape': '2 cones base together'
                }
            }
        }
    """
    for k, v in nested_dict.items():
        if isinstance(d, collections.Mapping):
            if isinstance(v, collections.Mapping):
                r = update_nested_dict(d.get(k, {}), v)
                d[k] = r
            else:
                d[k] = nested_dict[k]
        else:
            d = {k: nested_dict[k]}
    return d
```

2. Processing the values

When auditing keys, I noticed that some of the values given in the example were formatted like this: `"black;yellow;black"` .

Those values should be processed as a list of values. (e.g `"black;yellow;black" -> ["black", "yellow", "black"]`).

We'll split the value at the ";" character. But we need to avoid splitting valid strings also using this character. One way to find those is to check for of empty space characters, which a valid string most likely have, and that our soon-to-be lists don't.

All the data from the XML file is text data, but we'll also want to convert number values to their corresponding valid python data format, namely integers and floats.

We'll process values using this function:

```
def process_value(value):
    """
        Process a value to return it in the correct format:
            - numbers are converted to int, or float.
            - text with ";" charaters but no blank space are split and a list of processed values is returned.
            - unicode values are converted to string
    """
    if ';' in value and ' ' not in value:
        process_value(value.split(';'))
    else:
        if isinstance(value, list):
            return [process_value(v) for v in value]
        else:
            try:
                value = int(value)
            except ValueError:
                try:
                    value = float(value)
                except ValueError:
                    try:
                        value = str(value.encode('utf8'))
                    except:
                        pass
    return value
```

3. Auditing & cleaning street types

One of the potentially problematic values are street names. In the French road system, street types are written at the beginning of the street names. I'll extract road types with a regex matching the first word from street names strings.

The French language use few abbreviations for road types, but there are much more different road types, so more room for mistakes. I found a list of the most common road types in France on Wikipedia https://fr.wikipedia.org/wiki/Odonymie_en_France, and saved them to a file for later use.

```
import json
common_street_types = ["Allée", "Anse", "Avenue", "Boulevard", "Carrefour", "Chaussée", "Chemin", "Cité", "Clos",

with open('common_street_types.json', 'w') as outfile:
        json.dump(common_street_types, outfile)
```

To clean the street types, I'll need to check for unexpected street types, but also for typos. One of the common typo are lowercase letters. Fortunately this is easily fixed so I'll make sure in the cleaning script to check for lowercase but valid street types and replace them with the correct type. (i.e 'rue' --> 'Rue').

For now, let's check how dirty our data is by running a script against our common_street_types to return street types not listed in the json file, by order of occurence.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import xml.etree.ElementTree as ET  # Use cElementTree or lxml if too slow
from collections import defaultdict
import pprint, re
from tqdm import tqdm
from operator import itemgetter
import json

# Regex matches first word from street string
# as per the french system
```

```
street_type_re = re.compile(r'([^\s]+)', re.IGNORECASE)

# Load expected_street_types from our common_street_types.json file
expected_street_types = []
with open('common_street_types.json', 'rw') as f:
    expected_street_types = [e.encode('utf-8') for e in json.load(f)]

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

def audit_street_type(osmfile):
    osm_file = open(osmfile, 'r')
    streets = defaultdict(int)
    for event, elem in tqdm(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):
                    street_name = str(tag.attrib.get('v').encode('utf-8'))
                    m = street_type_re.search(street_name)
                    if m and m.group() not in expected_street_types:
                        streets[m.group()] += 1
    return sorted(streets.items(), key=itemgetter(1), reverse=True)

print(audit_street_type("la-rochelle_france.osm"))
```

We get the following output:

```
>>> [('rue', 5), ('Grande', 3), ('Qu\xc3\xa9reux', 1), ('Zone', 1), ('D111', 1), ('La', 1), ('fran\xc3\xa7ois', 1
```

- rue
- Grande
- Quéreux
- Zone
- D111
- La
- françois
- quai
- Saint

'rue' and 'quai' are correct street types with a lowercase first letter. I need to see more of the others to understand what is going on.

I ran the same script, adding another if statement to see the full value for those problematic streets:

```
# Added if-statement to the audit_street_type function
if m.group().lower() not in [e.lower() for e in expected_street_types]:
    print(tag.attrib['v'])
```

Output:

```
Saint Nicolas
Zone Commerciale Beaulieu Est
D111
Grande Rue de Périgny
Grande Rue de Périgny
Grande Rue de Périgny
françois de vaux de foletier
Quéreux de la Plousière
```

```
La Plousière
```

After manually checking each one, here are the results:

- **Saint Nicolas:** should have been written "Rue Saint Nicolas", the street type is missing. It's not valid.
- **Zone Commerciale Beaulieu Est:** A commercial area, it's valid.
- **D111:** It's a county road, the D stands for "Département" It's valid.
- **Grande Rue de Périgny:** The valid street type was the second word! It's quite rare but "Grande Rue" is a valid type.
- **françois de vaux de foletier:** should have been written "Rue François de Vaux de Foletier", the street type is missing. It's not valid.
- **Quéreux de la Plousière:** "Quéreux" is a traditionnal form of habitat which only exists in a few regions of France, it's valid.
- **La Plousière:** In la Rochelle there is a "Rue de la Plousière", "Quéreux de la Plousière" and a "La Plousière", and they are just next to each other. So it's confusing, but valid.

Cleaning the problematic street names

```
mapping = {
    "rue": "Rue",
    "quai": "Quai",
    "Saint": "Rue Saint",
    "françois": "Rue François",
}

def clean_street_type(tag):
    street_name = str(tag.attrib.get('v').encode('utf-8'))
    m = street_type_re.search(street_name)
    val = m.group()
    if val not in expected_street_types:
        if val.lower() in lowercase_list(expected_street_types):
            bad_to_good = dict(zip(lowercase_list(expected_street_types), expected_street_types)) # {'boulevard':
            street_name = street_name.replace(val, bad_to_good[val])
        else:
            if mapping.get(val):
                street_name = street_name.replace(val, mapping[val])
    print(street_name)
    return street_name
```

4. Putting it all together

1. prepare_data_for_database.py process the OSM file and export a cleaned JSON file.
2. insert_data_in_mongodb.py insert the data from the JSON file to a Mongodb database.

5. Exploring our database

Overview statistics:

File sizes:

- `la-rochelle_france.osm` : **131.3 Mo** (Uncompressed)
- `processed-la-rochelle_france.json` : **183 Mo**

Number of documents:

```
>>> db.larochelle.find().count()
642777
```

Number of nodes

```
>>> db.larochelle.find({"elem_type":"node"}).count()
550656
```

Number of ways

```
>>> db.larochelle.find({"elem_type":"way"}).count()
92121
```

Top 5 contributing users

```
>>> c = db.larochelle.aggregate([
    {"$group":{"_id":"$user", "count":{"$sum":1}}},
    {"$sort":{"count":-1}}, {"$limit":5}
])
>>> for i in c:
        print(i)

{u'count': 314974, u'_id': u'\xcbdz\xebronK'}
{u'count': 88537, u'_id': u'aerx11'}
{u'count': 52541, u'_id': u'Eric V'}
{u'count': 52179, u'_id': u'PierenBot'}
{u'count': 21934, u'_id': u'Jessy Bertrand'}
```

Number of restaurants:

```
>>> db.larochelle.find({"amenity":"restaurant"}).count()
102
```

Additional data exploration:

La Rochelle is quite a touristic city, let's see what kind of leisure activities it can offer:

```
>>> query = [
  {"$match":{"leisure":{"$exists":1}}},
  {"$group":{"_id":"$leisure", "count":{"$sum":1}}},
  {"$sort":{"count":-1}}, {"$limit":20}
]
>>> for el in db.larochelle.aggregate(query):
...     print(el)
...
{u'_id': u'swimming_pool', u'count': 948}
{u'_id': u'pitch', u'count': 176}
{u'_id': u'park', u'count': 55}
{u'_id': u'sports_centre', u'count': 21}
{u'_id': u'playground', u'count': 14}
{u'_id': u'track', u'count': 13}
{u'_id': u'marina', u'count': 8}
{u'_id': u'garden', u'count': 5}
{u'_id': u'slipway', u'count': 5}
{u'_id': u'common', u'count': 2}
{u'_id': u'golf_course', u'count': 1}
{u'_id': u'stadium', u'count': 1}
{u'_id': u'merry_go_round', u'count': 1}
{u'_id': u'picnic_table', u'count': 1}
{u'_id': u'miniature_golf', u'count': 1}
{u'_id': u'hackerspace', u'count': 1}
{u'_id': u'dancing', u'count': 1}
```

948 swimming pools is quite an impressive number! But most of them actually seem to be private.

```
>>> c = db.larochelle.find({"leisure":"swimming_pool", "access":"private"})
>>> c.count()
910
```

I also wonder what kind of pitches there is.

```
>>> query = [
    {"$match": {"leisure": "pitch", "sport": {"$exists":1 }}},
    {"$group": {"_id": "$sport", "count":{"$sum":1}}},
    {"$sort":{"count":-1}},
    {"$limit":10}
]
>>> for el in db.larochelle.aggregate(query):
        print(el)

{u'count': 61, u'_id': u'tennis'}
{u'count': 48, u'_id': u'soccer'}
{u'count': 11, u'_id': u'basketball'}
{u'count': 8, u'_id': u'multi'}
{u'count': 4, u'_id': u'team_handball'}
{u'count': 2, u'_id': u'boules'}
{u'count': 1, u'_id': u'land_sailing'}
{u'count': 1, u'_id': u'baseball'}
{u'count': 1, u'_id': u'athletics'}
{u'count': 1, u'_id': u'table_tennis'}
```

Ideas for Additional improvements

One useful improvement to this dataset could be to create a field called **popular tourism points**. The use case for those points would be for instance to separate public amenities destined for local use (an example would be a public swimming pool mostly used by school classes) versus public amenities destined to tourists.

Implementing this improvements would be quite complex for a few reasons:

- First, what is "popular" is quite subjective. We would need some sort popularity threshold, or let an official and unique entity determine what to put in the category.
- Second measuring popularity is complex, some companies like Foursquare are good at it (could we use their API to scrape data? How good is the data is beyond restaurants and bars?). It would also probably require a local expertise (for instance for sightseeing points), so would not easily be automated at scale.
- Third, popularity can change, what is popular this year may not be next year. Such data would need to be updated regularly.

Ressources