# Data Wrangling Project - UDACITY

*AIMOD*

*September 5, 2017*

## Contents

## Parsing Naples

Naples, Italy is the area of my choice. Situated in the region of Campania, this municipality prevails as one of the largest Mediterranean cities with around 4.4 million people living in the metropolitan area. The port of Naples is ranked as the world's second highest level of passenger flow just after Hong-Kong. In the last decades, it has significantly expanded the transport infrastructure by developing the subway network and a high-speed rail linking the cities of Salerno and Roma. With such an influx of people, and the ongoing enlargement, it could became extremely handy to have access to a database which can provide directions on where to find a pizza, a planetarium, or the nightclub of your taste (who dares to say that Data Science is boring?). My efforts head in that direction, and in order to do so I have overpassed the surface elements: tag, node, way, and relation from the OSM file, and struggled to dig deeper in search of the attributes of interest.

### The OSM File

The map has been downloaded from The Open Street Map as an OSM XML file of a decompressed size of 62.7 MB fulfilling the requirements of the assignment prompt.[1]

#### Parsing

The function **etree.iterparse()** from the **lxml** model has been used to access the events of interests. Attention has been focused on *child attributes*, and not in *parents elements* which have been eventually deleted and records duly cleaned to improve computer performance. Given the excessive amount of rows with missing data in these levels, the output has been filtered.

#### Capturing Events of Interest

The following function aims to capture the element of our choice. In the present assignment I have focused on *cuisine*, but the script is equally useful for any other given attribute: *shop*, *amenity*, *clothing*, etc. I favor lists comprehension to iterate in search of a given string; however, being a Phyton novice, sometimes the syntax proved to be difficult to grasp; I came back to the traditional for loop, then. The output has been narrowed based on the most common keys obtained via a Counter from the collections module.[2]

[1] See attached text file for map's further information.

[2] See attached standalone.py script for a complete depiction of functions used.

```
def select_entry(data, k):
    return[dict((k,v) for (k,v) in d.items()) for d in data]

from collections import Counter
def most_common_keys(data, int):
    keys = [k for d in data for k in d.keys()]
    print(Counter(keys).most_common(int))
```

## Data Cleanup

### Foreign Characters

Being a file about Naples, It does not come as a surprise the existence of Italian street names aka multiple messages indicating that *ascii* could not be decode or vice versa. Although, I managed to save it to a csv file by encoding in *latin-1*; it became a few days task to try to solve the message received when trying to load data into the sql database: *You must not use 8-bit bytestrings …* Neither the use of *text_factory == str*, nor the *codecs module* helped me out; *Python2* made the matters more difficult as Unicode encoding does not occur automatically and there were no valid utf-8 representations. I was forced to migrate to *Python3* which handles this kind of problems without further ado. I got my csv file, converted to a Pandas data frame to facilitate some basic cleanup functions as replacing missing values by NaN or renaming columns, and subsequently writing the data to a csv file with a *latin-1* encoding.

### Standardizing Phone Numbers.

One feature that demanded extra attenton was the phone numbers as the original data presented different patterns: white spaces, Italy's international prefix not included, etc. To the best of my knowledge a valid phone number must contain: the country code for Italy (39), the Naples area code (089) and a seven digits phone number. A first approach through a *re.compile* pattern searching showed that out of the 25 appointed phone numbers in the 121 selected dictionaries, just two match the pattern.

The following functions were applied incrementaly to the column phone to standardize the phone numbers and remove the inappropriate ones. Although, the three functions could be coerced into one function; I segregated them for the sake of a better control check

```
def format_phone(data):
    phone = str(data)
    clean_phone  = re.sub('\+39\s{1}081\s{1}\d{7}', phone.replace(' ', ''), phone)
    return clean_phone

def format_phone_1(data):
    phone = str(data)
    clean_phone = re.sub('^081\d{7}', '+39{}'.format(phone), phone)
    return clean_phone

def format_phone_final(data):
    phone = str(data)
    m =  re.search(pattern_phone, phone)
    if m is not None:
        return m.group(0)
```

## The Database

I decided to use **sqlalchemy** to query the data in an efficient, and as much as possible, a *Pythonic* way. A table of 121 dictionaires containing the string *cuisine* was created. The columns *amenity* and *cuisine* were explicitly declared as *Index* to speed the subsequent queries. The list of keys is as follows:

[‘amenity’, ‘name’, ‘cuisine’, ‘opening_hours’, ‘street’, ‘housenumber’, ‘city’, ‘postcode’, ‘phone’, ‘website’, ‘internet_access’, ‘smoking’, ‘wheelchair’]

As the table’s objective is to provide a database of possible food providers in the city of Naples; one our first inquiries aimes to find the different non null types of amenities, as shown by the following display:

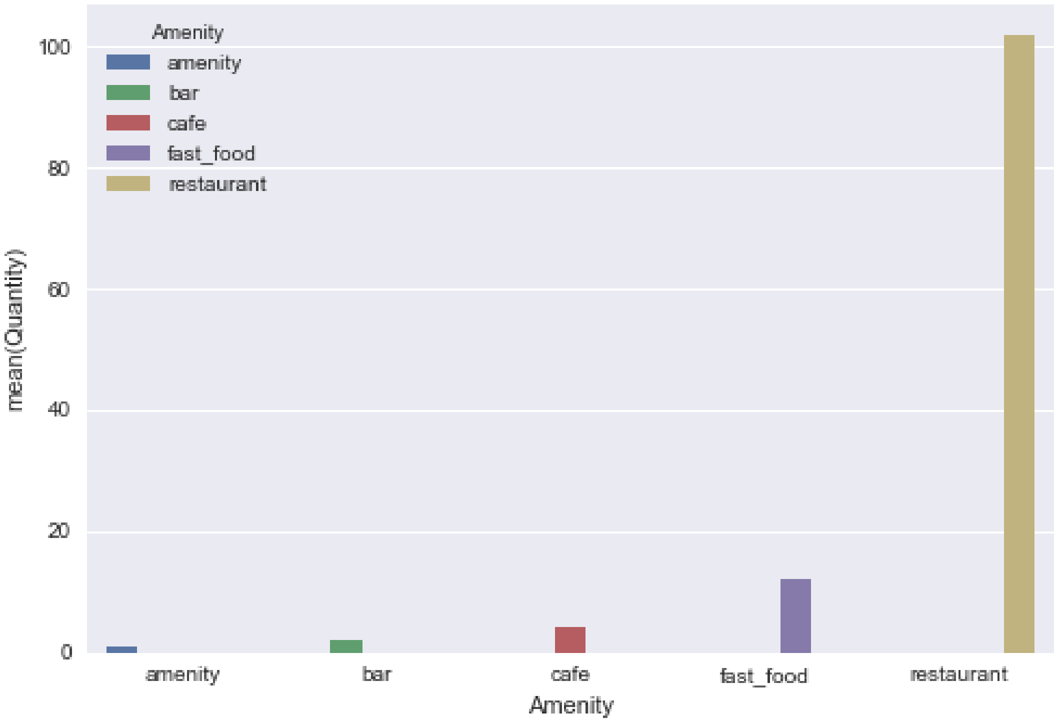

Figure 1:

Taking in account the above mentioned results, a potential user might be interested to find out what type of cuisine is being offered by the 102 restaurants described in the table. The proper query gives us 30 distinct values. For illustration purposes, a few of them are included:

| Cuisine | Quantity |
|---|---|
| italian | 30 |
| pizza | 38 |
| kebab | 2 |

It might be also the case that a tourist in Naples is craving for seafood. The following query suffices to supply the expected answers:

```
from sqlalchemy import and_
stmt = select([naples_cuisine.columns.name, naples_cuisine.columns.street,
              naples_cuisine.columns.housenumber]).where(
        and_(
                naples_cuisine.columns.amenity == 'restaurant',
                naples_cuisine.columns.cuisine.contains('seafood')
                 )
        )
```

I limit the query to the 10 first results:

[('Baccalaria', 'Piazzetta di Porto', '4'), ('Da Corrado', 'Via Michele Tenore', '1'), ('Da Patrizia', 'Borgo Marinari', '24'), ('La Lazzara', 'Piazza Francese', '9/10')]

All along, my objective has been to make tangible the skills learned on the Data Science Nanodegree's wrangling section. The following function brings me closer to my goal. A function designed to query the pertinent database to find the name, address and if provided, the phone number of a given location described in the original OSM file. As previously stated, I centered my inquire on the *cuisine* rubric, but any attribute of choice will do the deed.

```
def what_do_you_want_to_eat(type_of_food, integer):
    stmt = select([naples_cuisine.columns.name,
                   naples_cuisine.columns.amenity,
                   naples_cuisine.columns.street,
                   naples_cuisine.columns.phone])
    stmt = stmt.where(naples_cuisine.columns.cuisine == type_of_food)
    stmt = stmt.limit(integer)
    #TODO: add conditional statement in case that selection is not available
    results = connection.execute(stmt).fetchall()
    for result in results:
        #TODO: Improve printing output, space bewtween entries.
        print('Name:{}\nAmenity:{}\nAddress:{}\nPhone:{}'.format(result[0],
              result[1], result[2] ,result[3]))
```

A possible query:

what_do_you_want_to_eat('regional',2)

Et voila:

Name:Nennella

Amenity:restaurant

Address:Vico Teatro Nuovo

Phone:NaN

Name:Eccellenze Campane

Amenity:restaurant

Address:Via Benedetto Brin

Phone:NaN

## Ideas about the Dataset

Two main observations about the provided data:

- The quantity of missing data in the element's child. It hinders the proper creation of what it might be a highly advantageous tool for an interested user. As a matter of example, 0ut of the 121 observations, just 25 of them count with a phone number, some of them, not valid ones. See the statistics regarding NaN Values in the 13 columns. In some of the cases we have as far as 117 missing entries out of 121.

| Parameter | Result |
|---|---|
| count | 13.0 |
| mean | 71.0 |
| std | 45.0 |
| min | 0.0 |
| 25% | 52.0 |
| 50% | 78.0 |
| 75% | 111.0 |
| max | 117.0 |

- The quality of the data. The typology in the *cuisine* attribute seems rather loose. Apparently, it does not follow a standard, but most likely left upon the judgement of the entry's writer. The result is inaccurate and in some cases duplicated.

Since thinking out of the box is permitted; taking in account above observations, and keeping in mind the initial purpose of parsing the OSM file: to create a database capable to provide accurate and complete information to tourists and locals alike, the issues to be tackled are: to fill missing data and to assert validity

### Thinking out of the Box

I would suggest a virtual data scavenger hunt. To create a unique treasure hunt model which could be adapted for any location, and for any specific objective, as clear-cut as possible. A model able to be reused as required. For example, a rough, rough draft to respond to my needs:

- A clear and concrete objective: To find all missing telephone numbers of the subset *cuisine*
- A well-defined location: Naples, Italy
- A limited time: scavenger hunt to last 3 days
- Participants: open to data science communities
- In collaboration/sponsored by municipality or local government: Municipality of Naples
- Treasure: I could think on a wide range of possibilities, but there is nothing as enticing as the quest itself.
- Benefits:
  - B.1 - It takes advantage of already existent resources: the OSM file, the World Wide Web, social media for promotion, open sources, and most likely the already existent laptop.
  - B.2 - Results for each individual quest will allow for comparisons. The data could be validated based on similarity or dissimilarities alike. Thresholds should be established.
- Possible Caveats:
- C.1 - To assure the participation of the local entities.
- C.2 - To encourage participation of the data science community

Having fun with the sample map requested in the assignment prompt:

1- What am I looking for: **3523343315 334532121542**(Greek Square Cipher)

2- Location: **KXMLIV, FQXIV** (Cesar cipher)

3- Time: **ETHAY AMEGAY ILLWAY ARTSTAY ATYAY EVENSAY** (pig Latin, nothing to do with pig or Latin)

4 - Quest Map: The XPath syntax is perfect to formulate a quest in a classic Percy Jackson's style. One indeed, can get pretty creative with the parsed file. Below output provides as much clues as keys and values

```
<node id="5020742022" visible="true" version="2" changeset="50989612"
 timestamp="2017-08-10T05:54:01Z" user="John Bek" uid="3800987" lat="40.8406112"
 lon="14.2479445">
 <tag k="addr:housenumber" v="78"/>
 <tag k="addr:street" v="Vico Lungo del Gelso"/>
 <tag k="amenity" v="restaurant"/>
 <tag k="cuisine" v="italian;pasta"/>
 <tag k="name:en" v="Trattoria Annaré"/>
 <tag k="name:it" v="Trattoria Annaré"/>
</node>
```

5- A GOOGLE search, once the names are founded; it gives me about 107,000 results in 1.12 seconds; the first one corresponds to our hunt: *Antica Trattoria de Pepino Napoli* +39081407649. I am sure that this process could be scripted as well.

## Conclusion

I realized that the probability of creating ad-hoc data products based on the OSM files are enormously. Products that might be of great service to the communities involved. It goes beyond tags like latitude, and longitude. Data collection and process must be upscaled to provide reliable information, but the experience has been not just rewarding, but completely enticing.