For my project, I cleaned and examined the Open Street Map data of Santiago, Chile, the city in which I live.

The file can be found here: https://mapzen.com/data/metro-extracts/metro/santiago_chile/

# Problems encountered

## Abbreviations and strange characters

In many countries, streets have a prefix or suffix attached to their name denoting what type of street it is (e.g., Baker Street, Independence Avenue). In Santiago, most streets don't have prefixes. Streets just have their name (e.g., Santo Domingo, Bellavista) without prefixes or suffixes.

However, some types of streets do have prefixes. Avenues ("Avenida") denote longer, larger roadways and passages ("Pasaje") denote short side streets. These prefixes were often abbreviated, so I needed to change these abbreviations to their full forms.

In the OSM dataset, some accented characters were subsituted with others. For example, an "í" could be subsituted in the dataset as "Ã", an "ó" as "Ã³", and "é" as "Ã©". A word such as "educación" would be presented as "educaciÃ³n".

I used regular expressions to change the abbreviations and special characters.

I updated the update_name function used in the Preparing for Database - SQL quiz in the Case Study of P3's SQL For Data Analysis section to modify the street names by using regular expressions.

In [ ]:

```
# update_name function code:
av = re.compile(r'^(Av\.|Avda\.?|Avenida\.)', re.IGNORECASE)
pasaje = re.compile(r'^(Pje|Psje)\.?\s+', re.IGNORECASE)
accented_i = re.compile(r'Ã', re.IGNORECASE)
accented_o = re.compile(r'(Ã³|í³)', re.IGNORECASE)
accented_e = re.compile(r'(Ã©|í©)', re.IGNORECASE)
accented_u = re.compile(r'(í°|Ã°)', re.IGNORECASE)
accented_a = re.compile(r'í¡', re.IGNORECASE)
santa = re.compile(r'Sta(\.)?\s+', re.IGNORECASE)
nuestra = re.compile(r'ntra(\s)+', re.IGNORECASE)
senora = re.compile(r'sra', re.IGNORECASE)
n = re.compile(r'í±', re.IGNORECASE)
pob = re.compile(r'pob\.', re.IGNORECASE)

expected = ["Avenida", "Pasaje"]

mapping = { "Av.": "Avenida",
            "Pje." : "Pasaje ",
            "Ã" : "í",
            "Ã³": "ó",
            "Ã©": "é",
            "í°": "ú",
            "í³": "ó",
            "í©": "é",
            "í±": "ñ",
            "í¡": "á",
            "Sta": "Santa ",
            "Ntra": "Nuestra ",
            "Sra": "Señora",
            "Pob": "Población",
            }


def update_name(street_types):
    x = street_types
    match1 = re.findall(av, x)
    if match1:
        x = re.sub(av, mapping['Av.'], x)
    match2 = re.findall(pasaje, x)
    if match2:
        x = re.sub(pasaje, mapping['Pje.'], x)
    match3 = re.findall(accented_u, x)
    if match3:
        x = re.sub(accented_u, mapping['í°'], x)
    match4 = re.findall(accented_i, x)
    if match4:
        x = re.sub(accented_i, mapping["Ã"], x)
    match5 = re.findall(accented_o, x)
    if match5:
        x = re.sub(accented_o, mapping["í³"], x)
    match6 = re.findall(accented_e, x)
    if match6:
        x = re.sub(accented_e, mapping["í©"], x)
    match7 = re.findall(n, x)
    if match7:
        x = re.sub(n, mapping['í±'], x)
    match8 = re.findall(accented_a, x)
    if match8:
        x = re.sub(accented_a, mapping["í¡"], x)
    match9 = re.findall(santa, x)
    if match9:
        x = re.sub(santa, mapping['Sta'], x)
    match10 = re.findall(nuestra, x)
    if match10:
        x = re.sub(nuestra, mapping['Ntra'], x)
    match11 = re.findall(senora, x)
    if match11:
        x = re.sub(senora, mapping['Sra'], x)
    match12 = re.findall(pob, x)
    if match12:
        x = re.sub(pob, mapping['Pob'], x)
    return(x)
```

## Missing data

There are also some values in the dataset which are null. I modified the validate_element function, used in the Preparing for Database - SQL quiz in the Case Study of P3's SQL For Data Analysis section, to reveal the errors:

In [ ]:

```
# Code to see missing values where the data doesn't match the schema:
```

```
def validate_element(element, validator, schema=SCHEMA):
    """Raise ValidationError if element does not match schema"""
    if validator.validate(element, schema) is not True:
        try:
            field, errors = next(validator.errors.items())
            message_string = "\nElement of type '{0}' has the following 
errors:\n{1}"
            error_string = pprint.pformat(errors)
        except TypeError:
            print(element)
            print(validator.errors.items())
        #raise Exception(message_string.format(field, error_string))
```

The above code produces the following output on a sample of the full database:

{'node': {'changeset': '44066329', 'id': '4440880924', 'lat': '-33.3850177', 'lon': '-70.546608', 'timestamp': '2016-11-30T18:00:13Z', 'uid': '3544143', 'user': 'garylancelot', 'version': '3'}, 'node_tags': [{'id': 4440880924, 'key': 'source', 'value': 'Reconocimiento cartográfico 2016 por KG.', 'type': 'regular'}, {'id': 4440880924, 'key': 'leisure', 'value': 'playground', 'type': 'regular'}, {'id': 4440880924, 'key': None, 'value': 'Kg Ground Survey 2016', 'type': 'regular'}]}

dict_items([('node_tags', [{2: [{'key': ['null value not allowed']}]}])])

The value for key in one of the node_tags is "None", which violates the schema. In the full dataset, there are 25 of these errors in the nodes_tags and ways_tags lists.

I use try-excepts to bypass the errors and leave them as is in the full csvs. Guessing to fill in the values would not add value to the dataset, and even with 25 missing values, the fact that the datasets have millions of data points means that almost all of the values are present, and in this type of analysis the missing values do not significantly impact the findings.

In [ ]:

```
# validate_element with try-excepts:
def validate_element(element, validator, schema=SCHEMA):
    """Raise ValidationError if element does not match schema"""
    if validator.validate(element, schema) is not True:
        try:
            field, errors = next(validator.errors.items())
            message_string = "\nElement of type '{0}' has the following 
errors:\n{1}"
            error_string = pprint.pformat(errors)
        except TypeError:
            pass

        try:
            raise Exception(message_string.format(field, error_string))
        except UnboundLocalError:
            pass
```

## Querying the data

The section below shows the queries I ran in SQLite to obtain the results, as well as the output generated by SQL.

## Number of nodes:

select count(*) from nodes;

968870

## Number of ways:

select count(*) from ways;

238052

## Top five types of nodes:

select key, count(*) from nodesTags group by key order by count(*) desc limit 5;*

street;198858 housenumber;198315 name;23847 highway;13877 source;13420

## Largest ways:

select id, count(*) from waysNodes group by id order by count(*) desc limit 10;*

256661235|1516 247511847|1474 250710337|1270 250060986|1217 25753758|1058 461316568|898 461298131|854 225866302|832 239370554|830 107095865|786

#### What is the largest way?

select * from waysTags where id = 256661235;

256661235|natural|scrub|regular

## Top five contributing users:

#### Nodes:

select user, count(*) from nodes group by user order by count(*) desc limit 5;*

Julio_Costa_Zambelli;186240 Fede Borgnia;128459 felipeedwards;83103 dintrans_g;53271 garylancelot;41749

#### Ways:

select user, count(*) from ways group by user order by count(*) desc limit 5;*

Fede Borgnia;67453 Julio_Costa_Zambelli;20517 chesergio;19060 garylancelot;13772 felipeedwards;11675

## Contributions of top users:

Below, I look at the contributions of the top two users, Fede Borgia and Julio_Costa_Zambelli.

select key, count(key) from nodesTags join nodes on nodesTags.id = nodes.id where nodes.user = 'Julio_Costa_Zambelli' group by key order by count(key) desc limit 10;

housenumber|2523 street|2522 name|2277 city|1773 source|1709 amenity|1486 highway|1331 ref|1098 operator|957 type|931

select key, count(key) from nodesTags join nodes on nodesTags.id = nodes.id where nodes.user = 'Fede Borgnia' group by key order by count(key) desc limit 10;

street|122776 housenumber|122745 city|17 amenity|2 email|2 name|2 operator|2 source|2 type|2

select key, count(key) from waysTags join ways on waysTags.id = ways.id where ways.user = 'Julio_Costa_Zambelli' group by key order by count(key) desc limit 10;

barrier|5925 building|5681 highway|5057 name|2513 street|1900 housenumber|1799 city|1696 golf|1308 country|1192 oneway|1184

select key, count(key) from waysTags join ways on waysTags.id = ways.id where ways.user = 'Fede Borgnia' group by key order by count(key) desc limit 10;

interpolation|67453 id_origin|8860 end_number|2

I queried for the top ten most contributed tags for Fede Borgnia but the query only returned three. So this user has only made three types on contributions, one of them being interpolations. Speaking of which, what on earth is an interpolation?! I had never even seen the word before. From the Merriam-Webster dictionary, the definition of "interpolate" is:

1 a : to alter or corrupt (as a text) by inserting new or foreign matter b : to insert (words) into a text or into a conversation

2 : to insert between other things or parts : intercalate

3 : to estimate values of (data or a function) between two known values intransitive verb : to make insertions (as of estimated values)

A query to see more interpolations:

select * from waysTags where key = 'interpolation' limit 10; 38226598|interpolation|odd|addr 38226599|interpolation|odd|addr 38226600|interpolation|odd|addr 38226601|interpolation|odd|addr 38226603|interpolation|odd|addr 38226604|interpolation|odd|addr 38269089|interpolation|odd|addr 38327626|interpolation|odd|addr 38329811|interpolation|even|addr 38329812|interpolation|even|addr

My guess of what these is that the user is inserting address values into the data. The data are all of 'address' type and have values of 'even' or 'odd', which must correspond to numbers.

To finish, we see two different types of contribution styles, one where the user makes more broader edits, and one where the user focuses on a few specific areas.

## Number of users who contributed to both nodes and ways:

create table distinctNodesUid as select distinct uid from nodes; create table distinctWaysUid as select distinct uid from ways; select count(*) from distinctNodesUid join distinctWaysUid on distinctNodesUid.uid = distinctWaysUid.uid;

731

## Top five sporting ways tags:

select value, count(*) from waysTags where key = 'sport' group by value order by count(*) desc limit 5;

soccer;587 golf;236 tennis;208 basketball;142 swimming;39

...but you already knew that Chileans love soccer!

## File sizes

santiago_chile.osm: 269 MB stgo.db: 148 MB nodes: 84 MB nodes tags: 20 MB ways: 15 MB ways nodes: 28 MB ways tags: 15 MB

## Suggested improvements

This dataset is very detailed. I'm not sure what I was expecting, but I am impressed by the level of detail that the dataset has and how well organized it all was, which made cleaning the data much easier than it could have been. However there are some improvements that could be made:

### Cleaning up tags

There are multiple tags that explain the same thing in the dataset. For example, in the waysTags table there are tags for soccer, futbol, and football. They all reference the same sport. There are also two values for yoga, multi, and table tennis. Merging these tags into more appropriate groups would make the data more concise.

### Standardizing street names

Similarly, removing the abbreviations and the strange/foreign characters from the dataset would make the data easier to read and process. Someone with no knowledge of Spanish names and words may be thrown off by them; they wouldn't have the background knowledge to figure out what something should say in order to clean and process it.

It would be great to develop code that could find these characters in the code programatically. I solved the problems as I found them when manually looking at samples of the dataset. Besides being inefficient, this could leave errors unfound. The challenge would be knowing what characters and subsitutions were present in the raw dataset beforehand. It would have to be done before the data is put into SQL because after the data has been cleaned, the problems largely disappear. I don't have a solution to this problem, but working on it would improve the quality of the dataset.

### Better defining email types

Right now tags with 'email' keys have two types: 'contact' and 'regular'.

select * from waysTags where key = 'email' limit 5;

23395403|email|ovaldene@sanignacio.cl|regular 28410709|email|m.toro@cmpuentealto.cl|contact 28410927|email|liceoindustrial@hotmail.com|contact 28411094|email|administracion@celbosque.cl|contact 28664228|email|adelabarra@cmpuentealto.cl|contact

It's not immediately clear what the distinction is between the two. The 'regular' type comes from the code, as certain node_tags were assigned this value, but this method of processing the data puts a distinction into the emails that might not exist in reality.

From the shape_element function:

for node_tag in elem: node_tag_attribs = {} match = re.findall(PROBLEMCHARS, node_tag.attrib['k']) match2 = re.findall(more_than_one_colon, node_tag.attrib['k']) match3 = re.findall(LOWER_COLON, node_tag.attrib['k']) if match: node_tag_attribs['id'] = elem.attrib['id'] node_tag_attribs['key'] = None node_tag_attribs['value'] = node_tag.attrib['v'] node_tag_attribs['type'] = 'regular' tag_list.append(node_tag_attribs) elif match2: split = node_tag.attrib['k'].split(':') node_tag_attribs['id'] = elem.attrib['id'] node_tag_attribs['key'] = split[1] + split[2] node_tag_attribs['value'] = node_tag.attrib['v'] node_tag_attribs['type'] = split[0] tag_list.append(node_tag_attribs) elif match3: k_split = node_tag.attrib['k'].split(':') node_tag_attribs['id'] = elem.attrib['id'] node_tag_attribs['key'] = k_split[1] node_tag_attribs['value'] = node_tag.attrib['v'] node_tag_attribs['type'] = k_split[0] tag_list.append(node_tag_attribs) else: node_tag_attribs['id'] = elem.attrib['id'] node_tag_attribs['key'] = node_tag.attrib['k'] node_tag_attribs['value'] = node_tag.attrib['v'] node_tag_attribs['type'] = 'regular' tag_list.append(node_tag_attribs)

## Fixing phone formatting

Phone numbers are presented in a variety of formats

select * from waysTags where key = 'phone' limit 10;

28410709|phone|+56 2 4854096|contact 36977066|phone|+56222104000|regular 39611047|phone|+56 2 8535715|contact 40554204|phone|2520 4100|regular 47997799|phone|+5622321807|regular 56177875|phone|+5627700000|regular 93165179|phone|+5623715723|regular 106056892|phone|+56 2 8211707|contact 106056908|phone|+56 2 8215407|contact 106056917|phone|+56 2 8216165|contact

There are multiple problems here. There is inconsistent spacing between the digits. Some values don't include the area code (+56) as well as a digit to specify whether the number is a mobile or landline (9 for mobile, 2 for landlines). There is also the 'regular' - 'contact' issue mentioned earlier. To clean them, we would have to settle on a consistent format for phone numbers, then write code to detect numbers that don't fit the format and fix them. But even this wouldn't help us find out where the eight digit numbers belong to cell phones or landlines, as there is no way in the data to find out the prefix which makes that distinction. It may or may not matter for an analysis, but this is a shortcoming of this part of the OSM data. To fix this, better collection/input procedures would need to be established so that all phone numbers have all of the digits.