Prepared by: W. McCracken

The purpose of this project is to demonstrate that I am capable of:

- Assessing the quality of a data set for validity, accuracy, completeness, consistency and uniformity
- Parsing and gathering data from a popular file formats such as .csv, .json, .xml, and .html
- Processing data from multiple files or very large files that can be cleaned programmatically.
- Learning how to store, query, and aggregate data using SQL.

I chose the area of Alameda County shown in the shaded box above as Udacity is headquartered and I live there. Reference: Alameda County Map via mapzen - https://mapzen.com/data/metro-extracts/your-extracts/11184ceaf7a2 Reference to download of the OSM file: https://s3.amazonaws.com/mapzen.odes/ex_mj5W3UGssTC68RBcyHym7dM1SuvcF.osm.bz2 The boundaries of the map are: minimum latitude ="37.3900729" , minimum longitude ="-122.4546969" , maximum latitude ="37.9700843", maximum longitude="-121.3876639". The process overview below shows the high level flow of the activity to complete the project:

Process Overview:

Sampling: I ran a quick count tags on the full data full data set and took a few minutes to process even though I have 8GB of RAM and processing the 1.5GB file should not be a problem. I used the sample extract code (sampleOSMfilecreation.py) and extracted three samples, every 10, every 20 and every 30 nodes. Lower, lower colon, other and problem character columns are a count of the format fields found in each data set. Skipping 30 nodes still created a large file so I opted to skip every 95 nodes to develop test code for testing. If the table below; nodes, node tags, ways, ways nodes and ways tags are a count of the number of records in each csv file. A comparison of the files follow:

Data set	Lower	Lower colon	Other	Problem Characters	Nodes	Node Tags	Ways	Ways Nodes	Ways Tags
Full	1,66,0897	781,175	32,877	91	7,247,571	301,600	902.723	8,530,599	2,145,461
Sample every 10th	165,860	78,142	3,280	16	724,758	30,108	90,272	854,237	214,409
Sample every 20th	83,174	39,113	1,650	6	362,378	15,481	45,136	426,568	107,163
Sample every 30th	55,236	25,813	1,058	8	241,587	10,032	30,091	285,633	71,132
Sample every 95th	17,470	8,351	336	7	76,441	3,152	9,502	88,434	22,708

Since sampling every 95th element creates files with a good variety, I used this to developed cleaning and analysis algorithms before applying this to the full file. To get a sense for the types of elements in each of the files I ran the code to prepare the data for analysis and created csv files for each sample map(preparefordatabases10.py). I then opened them to explore them in Microsoft Excel by selecting a data range and applying a filter to the fields. This is a very fast way to explore variation and range in data before applying more

sophisticated/comprehensive Python techniques. As a result, the 10th sample is too large to open in excel so I studied the fields in each of the 20th files to get a sense for the values in each file.

Exploring: After exploring the data in excel, I created to exploratory python scripts count of the values of child tags to dig deeper into the data and find fields that might need cleaning. Exploring the values of most common "child" (v) tags (exploretypes.py) such as street, postcode, house number, and cuisine indicated very few issues worth cleaning. The phone numbers were in several different format warranting cleaning. It appeared the data was fairy clean so I raised the review to look at the value "parent" (k) tags. There are 452 parent "tags" in the sample. (explore_k_tagtypes.py) A review of the parent tags helped identify a few fields that could be cleaned such as the state, country and variations of the name field. In addition, this review demonstrated that the value of one of the most widely used tags in the map file "name" had variations in the tag that warrant cleaning.

The basic function in both the explore python scripts is a function "audit_element_type" shown below:

```
""
def audit_element_type(element_types, element_name):
    m = element_type_re.search(element_name)
    if m:
        element_type = element_name
        element_types[element_type] += 1


def is_element_name(elem):
    return (elem.tag == "tag") and (elem.attrib['k'] == "phone")
""
```

This loops though the osm file and adds the elements being inspected to a dictionary with a counter. A separate function prints the dictionary so you can see the results immediately.

Example 1: Changing the element name in the "is_element_name" function in the exploretypes.py script to "phone" generated the following:

```
"" Sample Output
(650) 961-8600: 1
(800) 275-8777: 1
+1 (415) 641-9885: 1
+1 (510) 524-2532: 1
+1 415 621 0874: 1
+1 415 800 7416: 1
+1 510 4440919: 1
+1 510 452-1499: 1
+1 510 545 4356: 1
+1 510-655-6336: 1
+1-415-932-6531: 1
1-415-241-6380: 1
1-650-965-3189: 1
415-648-4157: 1
""
```

This shows the variety of ways phone numbers are included in the data which led to adding a cleaning function for this field to the code to prepare the data for the database.

Example 2 : Changing the element name in the "is_element_name" function to "addr:state" generated the following :

‘’’ Sample Output
CA: 127
Ca: 2
ca: 1
California: 1
’’’

This shows the four ways state is included in the data leading to adding a cleaning function for this field to the code to prepare the data for the database.

I applied the same exploratory code to elements such as "addr:postcode", "addr:street", "amenity" and several others. While there was some variation in some of the elements I explored, I chose to focus on cleaning three "child" value fields: "phone", "addr:state" and "addr:country". Cleaning algorithms for phone number, state and country will be elaborated on in the cleaning section below.

Explore "k" types: The second exploratory script, explore_k_tagtypes.py is almost identical to the exploretypes.py with the exception of the is_element_name function. In the "k-type" script we are just checking for the count of all the various tags. This lets you know which tags might need to be cleaned.

```
’’’
def is_element_name(elem):
   return (elem.tag == "tag")
’’’
```

Example 3 : Executing the , explore_k_tagtypes.py generated the following view of the "name" tag:

‘’’ Sample Output
name: 1391
name:en: 4
name:prefix: 5
name:vi: 1
name:zh: 4
name_1: 26
name_2: 1
’’’

Note: Exploration is an iterative process. Exploring the values of name:vi and name:zh with the exploretypes.py script uncovered chines characters in the child elements values for these tags.

Given that the name tag is the most prominent tag in the file, I developed a cleaning algorithm to standardize the name tag. ". The cleaning algorithm for the tag "name" will be elaborated on in the cleaning section below. Note, I developed a cleaning algorithm for a "tag" value so that it could be easily used to clean any "tag" value in this or any other xml file. It will be useful in the future.

Other auditing / exploratory notes:

- Auditing Validity – Initial attempts to scrape the data failed as a userid was expected for some records and it was not present. An additional step was added to the parsing function to skip records with no userid. The numeric range for postal codes in the sample file is in a tight range all beginning with 94.
- Auditing Accuracy – A review of the accuracy of a few fields indicated relatively accurate data. For example, four counties appeared in the data which are all valid counties in the area. While the country identified is all United States, four abbreviations were found which led to the cleaning step discussed above.
- Auditing Uniformity – I validated several fields for uniformity using excel filtering techniques discussed above. For example, all of the postcode values in the sample file began with "94" and all the latitude and longitude values were numeric and within the minimum and maximum longitudes and latitudes of the map.

Scraping/Cleaning and Export to CSV: The main script for the project reads the OSM file (XML format) and parses though it to create five CSV files that can be loaded into SQL for analysis. (prepfordatabase.py) I added a function (def update_name) to standardize the state and country. This function could be modified to clean any "child" element. I added a function (fixphone) to standardize the phone number. To simplify the code, I modified added an additional parameter to the def update_name function which allows for standardization of either a parent "k" level label or a child "v" value. After developing the cleaning algorithms described above, I reran the preparefordatabases10.py script against the full alamedamap.osm file to export "clean" csv files for querying. I quickly ran into conditions in the update_name function that I did not plan for. For example, "name:en" and "name:ckb". After updating the map function for three or four new conditions, I decided to go back and explore the "k" types of the full map. There are 1,732 tags in the full map including 83 variations of the name "key" value. Further investigation of the OSM file revealed the following valid uses for variations of the name field:

<tag k="name" v="Palo Alto"/>
<tag k="name:ar" v="ألتو بالو"/>
<tag k="name:bg" v="Пало Алто"/>
<tag k="name:el" v="Πάλο Άλτο"/>
<tag k="name:en" v="Palo Alto"/>
<tag k="name:eo" v="Palo Alto"/>
<tag k="name:es" v="Palo Alto"/>
<tag k="name:fa" v="آلتو پالو"/>
<tag k="name:he" v="אלטו פאלו"/>
<tag k="name:hi" v="पालो आल्टो"/>
<tag k="name:ja" v="パロアルト"/>
<tag k="name:ko" v="팰로앨토"/>
<tag k="name:lv" v="Paloalto"/>
<tag k="name:nv" v="Tsin Łichíʼínéézʼ"/>
<tag k="name:ru" v="Пало-Альто"/>
<tag k="name:sr" v="Пало Алто"/>
<tag k="name:uk" v="Пало-Альто"/>
<tag k="name:ur" v="مے ئی فی لٹو"/>
<tag k="name:uz" v="Palo-Alto"/>
<tag k="name:zh" v="帕罗奥图"/>
<tag k="name:zh-hans" v="帕罗奥图"/>
<tag k="name:zh-hant" v="帕罗奥图"/>

As a result modified the update_name function to only correct one instance of the name field "nam" as that appears to be inaccurate whereas the remaining variations relate to an international indicator for translation purposes. I also encountered a unique phone number that was not in ASCII format which caused an error in the fixphone function as I was initially using a "str" function to remove all non-numeric characters from the number. I changed the string stripping function to utilize a regular expression function (numberout =re.sub('[^0-9]','', numberin) to strip the non-numeric characters from the phone string that was input to the function. The final challenge I ran into was encountering a state and country in the full map that were not in the sample map and therefore not in the mapping list used to validate and correct these fields. By this time I had gotten a little wiser and ran the exploretypes script on the state and country fields of the full map before I made final updates to the scraping script. The final scraping script rant for approximately 6 hours.

Import to SQL: After developing the cleaning algorithms described above, I reran the preparefordatabases.py script to export "clean" csv files for querying. I incorporated use of the SQLalchemy package due to the large size of the files and performance objectives. This

simplifies the code to import data to a virtual SQL database and increased the speed of queries of the data. The SQL queries are identical to traditional SQL queries with the exception that they do not have to be wrapped into a "cursor" function. After reading the project csv files into pandas dataframes using the pd.read_csv command, the following was used to import the dataframes into SQL tables and a SQL virtual database:

```
eng = create_engine("sqlite://", encoding='utf8')
conn = eng.connect()
NODES.to_sql('nodes', conn)
NODE_TAGS.to_sql('node_tags',conn)
WAYS.to_sql('ways',conn)
WAYS_NODES.to_sql('ways_nodes',conn)
WAYS_TAGS.to_sql('ways_tags',conn)
```

Note: I initially had difficulty importing data into SQL with the basic SQLalchemy format command. After hours of investigation and some guidance from my mentor, I determined that the cause of the issue was Chinese characters in the values of some "value" tags. I contemplated excluding Chinese characters but since these are legitimate names and descriptions of businesses in this area, I added an "encoding='utf-8" parameter to the read_csv and creation of the sqlite engine connection to accept Chinese characters. This solved the problem.

SQL Queries: I created a variety of SQL as follows: (All output edited for readability)

```
###Record Count Metric database queries
numberofnodes = pd.read_sql("SELECT COUNT(*) FROM nodes",eng)
numberofnode_tags = pd.read_sql("SELECT COUNT(*) FROM node_tags",eng)
numberofways = pd.read_sql("SELECT COUNT(*) FROM ways",eng)
numberofways_tags = pd.read_sql("SELECT COUNT(*) FROM ways_tags",eng)
numberofways_nodes = pd.read_sql("SELECT COUNT(*) FROM ways_nodes",eng)
```

Results – edited for ease of reading

Table_name	Records Sample 95	Records Full Map
Nodes	76,291	7,247,571
Node Tags	3,152	301,600
Ways	9,502	902,723
Ways_Tags	22,708	2,145,461
Ways_Nodes	88,434	8,530,599

```
#Distinct Users – (JOIN)
distinct_users = pd.read_sql("SELECT COUNT(DISTINCT du.uid) \
    FROM (SELECT DISTINCT uid FROM nodes) as du \
    JOIN (SELECT DISTINCT uid FROM ways) as wu \
    ON du.uid = wu.uid",eng);
```

Results: Sample Distinct Users: 360

Results: Full Map Distinct Users: 1925

#Timestamps

```
most_recentnode = pd.read_sql("SELECT DISTINCT timestamp \
    FROM nodes Order by timestamp DESC Limit 2",eng)
print "Most Recent node updates:", most_recentnode

oldestnode = pd.read_sql("SELECT DISTINCT timestamp \
    FROM nodes Order by timestamp Limit 2",eng)
print "Oldest node updates:", oldestnode
```

Results: Time stamps

Most Recent node updates:	*Most Recent ways updates:*
2018-01-14T15:02:13Z	*2018-01-14T15:02:05Z*
2018-01-14T15:02:12Z	*2018-01-14T15:02:04Z*
Oldest node updates:	***Oldest ways updates:***
2007-03-08T02:02:45Z	*2007-03-08T02:41:07Z*
2007-03-08T02:02:46Z	*2007-08-22T22:28:47Z*

#**Most Active Contributors -** Users andygol and ediyes must be pro's!

```
“”
most_active_users = pd.read_sql("SELECT e.user, COUNT(*) as num \
    FROM (SELECT user from NODES UNION ALL \
        SELECT user from ways) e \
        GROUP BY e.user ORDER BY num DESC LIMIT 5",eng);
“”
```

Number	*user*	*num*
1	andygol	1,545,190
2	ediyes	634,728
3	dannykath	604,980
4	RichRico	458,471
5	Luis	36,995

#**Top Amenities -** Lots of places to eat and park. Good thing there a lot of restrooms too!

```
“”
top_amenities = pd.read_sql("SELECT e.value, COUNT(*) as num \
    FROM (SELECT value from node_tags WHERE key = 'amenity' UNION ALL \
        SELECT value from ways_tags WHERE key = 'amenity') e \
        GROUP BY e.value ORDER BY num \
        DESC LIMIT 10",eng);
“”
```

Top Amenities:	*value*	*num*
1	*parking*	*6,594*
2	*restaurant*	*3,763*
3	*school*	*1,561*
4	*bench*	*1,506*
5	*place_of_worship*	*1,233*
6	*cafe*	*1,169*
7	*fast_food*	*998*

8	*bicycle_parking*	*928*
9	*post_box*	*780*
10	*toilets*	*642*

Conclusions and recommendations: At the onset of this project I was skeptical of the overall value and practical application of the XML scraping and cleaning process and the use of SQL to the area of work I am in. Upon completion, I very pleased that I have built a framework for obtaining, exploring, cleaning and loading data into a SQL format for summary and analysis. I will be able to replicate for many simple applications I am involved in. In hindsight, I probably should have not picked a "Silicon Valley" map with so much active use. I probably would not have encountered as many challenges had a chosen a different map for the case study. Finally, in hindsight, I should have run the exploretypes script on the entire map for each field I planned to clean. It is an extremely useful exploratory tool

Recommendation: While OpenstreetMap is "open source" data, there should published standards for common fields like name tags and phone formats. This will improve the overall quality and usefulness of the data in applications.

Fun Observation from the map: "Imagine your future! What do you want to learn today? What's your dream job?" .....This appears to be a statement made by a school, university or learning institution. A quick comparison of the high level information about Stanford, Carnegie Mellon University and Udacity reveal a few differences. (see table below) Like musical artist that release music in obscure genres, Udacity might consider listing itself as a "university" or school as opposed to "company" to improve search results for people looking for an education. I also noted that Standord included translations of it's name in many languages, also something Udacity might consider since the student body is global. It's fun to create a comparison. This framework could be used in many ways.

	Key	value	key	value	key	value
0	name	Udacity	amenity	university	name	Carnegie Mellon University Silicon Valley
1	Level	0	ele	22	amenity	university
2	Office	company	county_id	85	website	https://www.cmu.edu/silicon-valley/
3	website	https://www.udacity.com/	created	1/19/1981	alt_name	Carnegie Mellon Silicon Valley
4	City	Mountain View	edited	1/4/2008	operator	Carnegie Mellon University
5	Full	2465 Latham Street, 1st Floor	feature_id	235365	wikidata	Q5043958
6	state	CA	state_id	6	wikipedia	en:Carnegie Mellon Silicon Valley
7	street	Latham Street	name	Stanford University		
8	postcode	94040	am	ስታንፎርድ ዩኒቨርስቲ		
9	housenumber	2465	ar	جامعة سـ تاذ فورد		
10			be	Стэнфардскі ўніверсітэт		
11			bg	Станфордски университет		