IBM Applied Data Science Capstone

Week #5 – Research Paper

"Opening a New Orient Restaurant in the City of Seattle"

By: Mohamad Nael QAWAS

June 2021

https://commons.wikimedia.org/wiki/File:Seattle_Kerry_Park_Skyline.jpg

1. Introduction / Business Problem

1.1 - Preface

This research paper is my capstone project (course #9) for the IBM Data Science Professional Certificate. Part of the requirements are to publish a blog summarizing the results, hence this article. This is my fourth or fifth data science project, but first one published to the public – kudos to this IBM certification course and the way it was organized such that writing an article is part of the final.

1.2 - Introduction

Orient restaurants are an innovative and healthy Hawaiian cuisine that are seeing growing demand.[1] Orient "build-your-own" bowl flexibility allows diners to choose from multiple bases[2] (salad, rice or quinoa), multiple protein sources[2] (salmon, tuna, octopus, red snapper, etc.), and multiple toppings[2] (10+ sauces and oils, seeds, onions, cucumbers, crab-salad, seaweed, dices mangoes or oranges, and much more). Orient bowls generally offer superior nutrition and taste relative to other fast-food options[3].

1.3 - Business Problem

The objective of this research project is to analyze and select the best location in Seattle for our "hypothetical" client to open a new Orient Restaurant. S/he already owns and operates one Orient restaurant just North of Seattle and is looking to expand by opening a second somewhere in Seattle.

Searching for the optimal location is challenging because it is not as simple as finding a geographic gap where there is not yet a Orient restaurant. It is more complex in that potential customers aren't interested in driving out to an isolated neighborhood with one Orient restaurant; but rather they prefer frequenting common clusters of restaurants(see "clustering" game theory[4]…not to be confused with "clustering" algorithms[5]) However, caution must be taken to carefully select the best location, because there is risk that the Orient market is already oversaturated[6] with 68 Orient restaurants currently in Seattle as per a Google Map search (via jump to last-in-list).[7]

To address this business problem, research will be done using data science methodology and machine learning techniques such as clustering to find and rank the best locations.

2. Data Acquisition

2.1 - Data Needs

To solve the business problem of finding the most suitable location in Seattle, we will need the following data:

- List of neighborhoods in the city of Seattle
- List of zip codes for Seattle
- Latitude and longitude coordinates of those neighborhoods to plot maps and tie in venue data
- Venue data linking geographic context such as restaurant cluster locations, whether a Orient restaurant already exists, average sales, average tips, average like ratings, etc.
- Various demographic metrics for Seattle neighborhoods and/or zip codes to provide context such as population density, rental vs. ownership percentages, median age, urban community type, etc.

2.2 - Data Sources

The data sources used in this research project include:

- The **City of Seattle OpenData Urban Centers** [8] CSV download was used to split Seattle up into 42 neighborhoods. Attributes include neighborhood size in acres, population, ethnicity breakdowns, rental vs. home ownership percent, and other demographics.
- The **AgingKingCounty.org**[12] PDF download cross-walks Seattle Neighborhoods to zip codes.
- The **OpenDataSoft.com**[13] web report was filtered down to a list of all Seattle zip codes along with their latitude and longitude coordinates. These were exported out to a CSV file for use in this project.
- The Python **Geocoder** package was used just for a level set map of Seattle. Decision was made not to use it for the neighborhood zip code to geo coordinates because it would have added a little complexity and development time. Instead the simpler OpenDataSoft CSV file was used instead.
- The **Foursquare API**[9] was used to add venue data for the neighborhoods such as number of restuarants, whether or not a Orient restaurant already exists, etc. Foursquare is one of the largest venue databases having over 105 million[10] global points of interest and 125,000[11] developers building location-aware experiences with the Foursquare API.

3. Methodology

3.1 - Data Loading

3.1.1 - Load Seattle Neighborhoods, Zip Codes, and Demographics from CSV

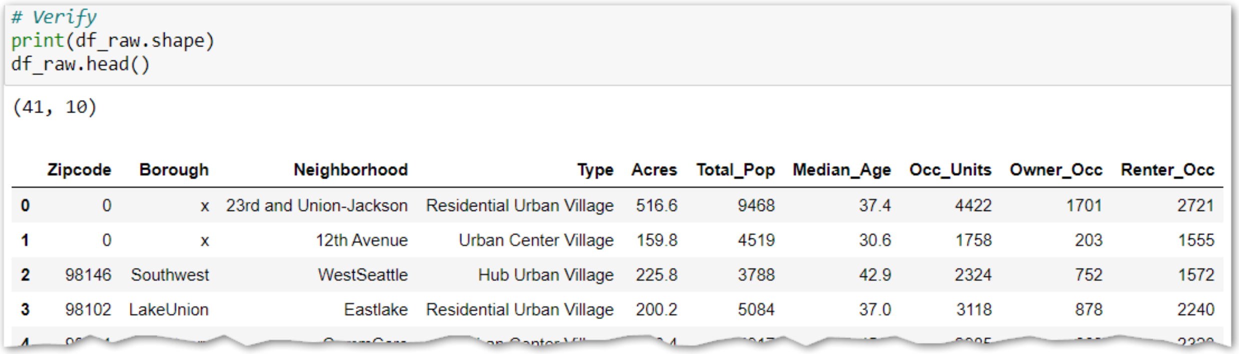
First, the City of Seattle OpenData CSV Urban Centers[8] file was downloaded containing 42 rows and columns such as neighborhood name, urban type, acres, square miles, total population, ethnicity breakdown, median age, housing units, number of renters, number of home owners, number of vacancies, etc. The initial plan was to directly download the CSV from python code (wget), but after the timebox was exceeded then a decision was made to simply manually download the CSV file.

Next, zip codes for each neighborhood were pulled from the AgingKingCounty[12] PDF file. Because this was a PDF file, it quickly became evident it would be faster to manually added the 42 zip codes to the existing CSV rather than get fancy writing python code to parse a PDF. Some Neighborhoods spanned multiple zip codes. Some zip codes spanned multiple neighborhoods. The combination resulted in 46 rows from the base set of 42. These would need to be de-duped and grouped later in code as part of data wrangling.

Next, the combined CSV file was compared to an actual online zip code map of Seattle. Two zip codes and neighborhoods were missing and were added, bringing the total to 48 neighborhood/zip code rows. Unfortunately, they were not present in the City of Seattle base set and thus were later removed.

After the data was manually consolidated into one CSV file, it was populated into Pandas DataFrames using the Pandas read_csv command. One column was dropped, and one was renamed. The data was verified using a "**print(DataFrame.shape)**" command to confirm expected row and column counts, as well as a "**dataFrame.head()**" to spot check the top 5 rows as shown below (Fig. 3.1.1).

Fig. 3.1.1 – Pandas DataFrame loaded with Seattle Neighborhoods, Zip Codes, and Demographics Data



```
# Verify
print(df_raw.shape)
df_raw.head()
```

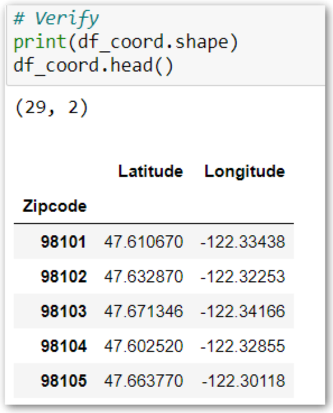
```
(41, 10)
```

	Zipcode	Borough	Neighborhood	Type	Acres	Total_Pop	Median_Age	Occ_Units	Owner_Occ	Renter_Occ
0	0	x	23rd and Union-Jackson	Residential Urban Village	516.6	9468	37.4	4422	1701	2721
1	0	x	12th Avenue	Urban Center Village	159.8	4519	30.6	1758	203	1555
2	98146	Southwest	WestSeattle	Hub Urban Village	225.8	3788	42.9	2324	752	1572
3	98102	LakeUnion	Eastlake	Residential Urban Village	200.2	5084	37.0	3118	878	2240

3.1.2 - Load Seattle Geo Coordinates by Zip Code from CSV

Next, the OpenDataSoft[13] website allowed filtering down to just zip codes in Seattle which were then exported out to a CSV file. These also were loaded into a Pandas DataFrame and verified (Fig. 3.1.2).

Fig. 3.1.2 – Seattle Geo-Coordinate data



```
# Verify
print(df_coord.shape)
df_coord.head()
```

```
(29, 2)
```

	Latitude	Longitude
Zipcode		
98101	47.610670	-122.33438
98102	47.632870	-122.32253
98103	47.671346	-122.34166
98104	47.602520	-122.32855
98105	47.663770	-122.30118

3.2 - Data Cleaning

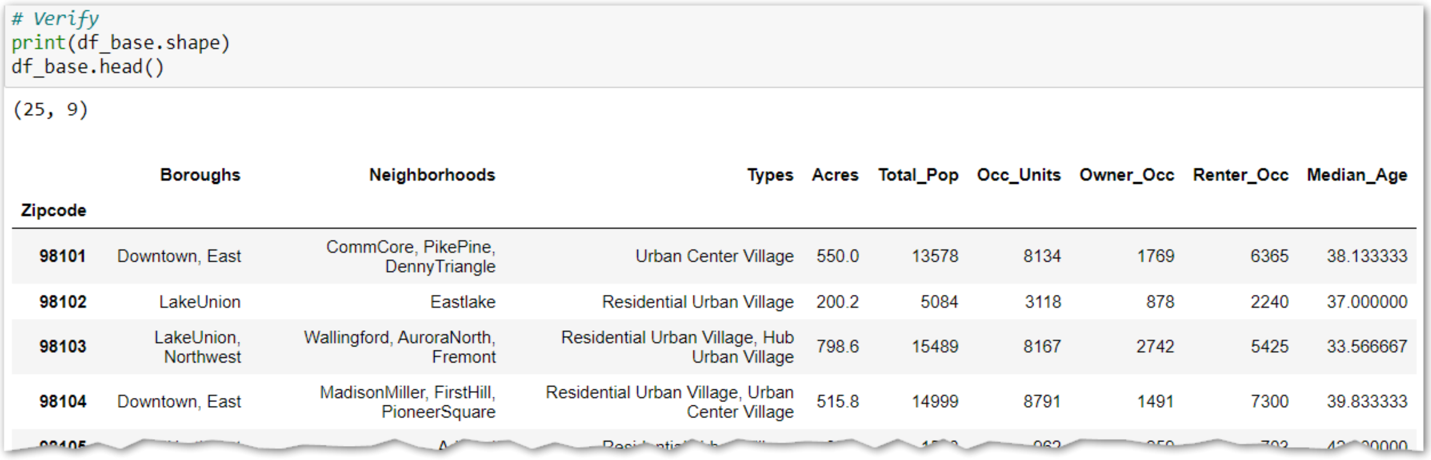
3.2.1 - Remove Missing, Null, or Blank Values Rows

Next, rows having Zipcode = 0 were dropped. Also, rows having NaN or Null rows at critical fields Borough, Neighborhood, and Type were also dropped. The data set dropped from 41 neighborhood rows down to 39 rows.

3.2.2 - Data De-Duped and Rolled Up to Zip Code

Next, data was rolled up to one zip code per row. Rollup string type fields Neighborhoods, boroughs, and Types using concatenation into comma delimited descriptions. Rollup numeric fields Acres, Total_Pop, Occ_Units, Owner_Occ, and Renter_Occ via the aggregate SUM function. Finally, rollup Median Age using the aggregate AVERAGE function. The 39 neighborhood rows dropped down to 25 rolled up zip code rows (Fig. 3.2.2) with the Boroughs, Neighborhoods, and Types columns all clearly having comma delimited list values where before there were multiple rows. The aggregated numeric fields were also tacked on to the right of the DataFrame.

Fig. 3.2.2 – Data De-Duped and Rolled Up to Zip Code



```
# Verify
print(df_base.shape)
df_base.head()
```

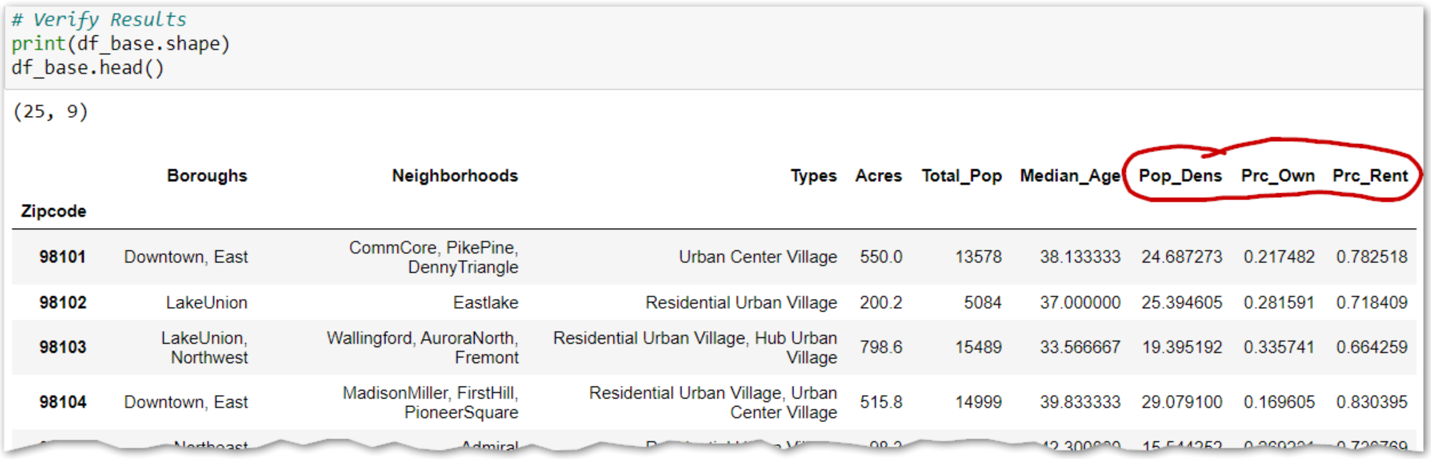
```
(25, 9)
```

	Boroughs	Neighborhoods	Types	Acres	Total_Pop	Occ_Units	Owner_Occ	Renter_Occ	Median_Age
Zipcode									
98101	Downtown, East	CommCore, PikePine, DennyTriangle	Urban Center Village	550.0	13578	8134	1769	6365	38.133333
98102	LakeUnion	Eastlake	Residential Urban Village	200.2	5084	3118	878	2240	37.000000
98103	LakeUnion, Northwest	Wallingford, AuroraNorth, Fremont	Residential Urban Village, Hub Urban Village	798.6	15489	8167	2742	5425	33.566667
98104	Downtown, East	MadisonMiller, FirstHill, PioneerSquare	Residential Urban Village, Urban Center Village	515.8	14999	8791	1491	7300	39.833333

3.2.3 - Calculate Densities and Percent Rental Units

Next, three density columns "Dens_Tot_Pop", "Dens_Home_Own", and "Dens_Apt_Dw" were calculated given the Acres per Zipcode, the total population, and the number of owner occupied and renter occupied unit counts. To cleanup this new base DataFrame, the interim fields "Owner_Occ" and "Renter_Occ" were dropped, leaving only the desirable fields. The base DataFrame remains at 25 rows, but now has 9 columns (Fig. 3.2.3).

Fig. 3.2.3 – Added Columns Pop_Dens, Prc_Own, and Prc_Rent



```
# Verify Results
print(df_base.shape)
df_base.head()
```

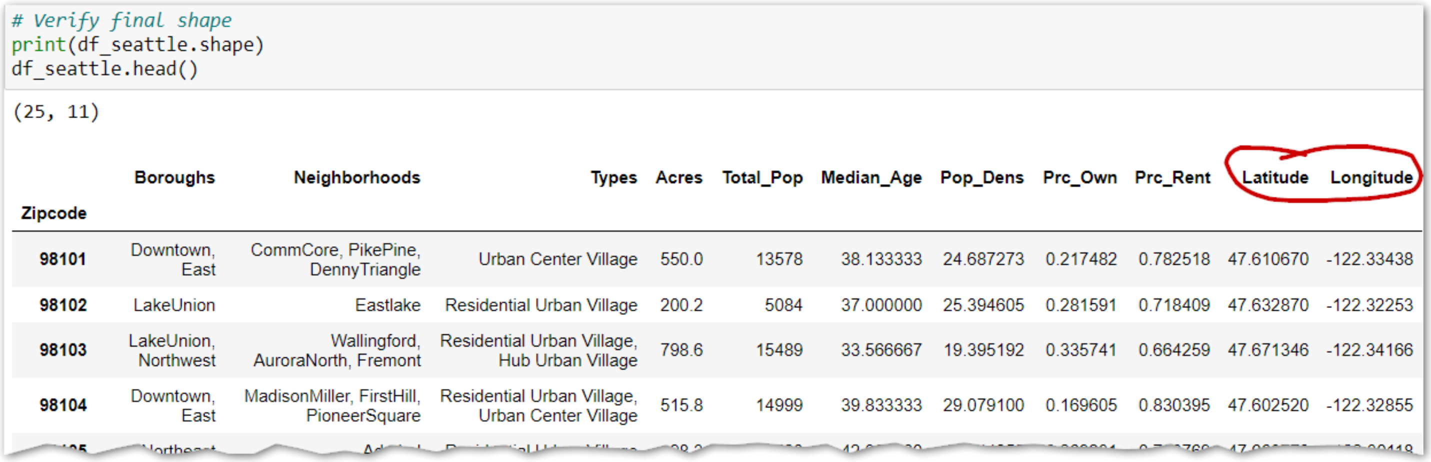
```
(25, 9)
```

	Boroughs	Neighborhoods	Types	Acres	Total_Pop	Median_Age	Pop_Dens	Prc_Own	Prc_Rent
Zipcode									
98101	Downtown, East	CommCore, PikePine, DennyTriangle	Urban Center Village	550.0	13578	38.133333	24.687273	0.217482	0.782518
98102	LakeUnion	Eastlake	Residential Urban Village	200.2	5084	37.000000	25.394605	0.281591	0.718409
98103	LakeUnion, Northwest	Wallingford, AuroraNorth, Fremont	Residential Urban Village, Hub Urban Village	798.6	15489	33.566667	19.395192	0.335741	0.664259
98104	Downtown, East	MadisonMiller, FirstHill, PioneerSquare	Residential Urban Village, Urban Center Village	515.8	14999	39.833333	29.079100	0.169605	0.830395

3.2.4 - Add Seattle Zip Code Geo Coordinates to Base DataFrame

Next, the two geo coordinates latitude and longitude are merged in to the base data frame from the DataFrame loaded in 3.1.2. The row count remains at 25, but the column count increased up to 11 (Fig. 3.2.4).

Fig. 3.2.4 – Added Columns Latitude and Longitude



```
# Verify final shape
print(df_seattle.shape)
df_seattle.head()
```

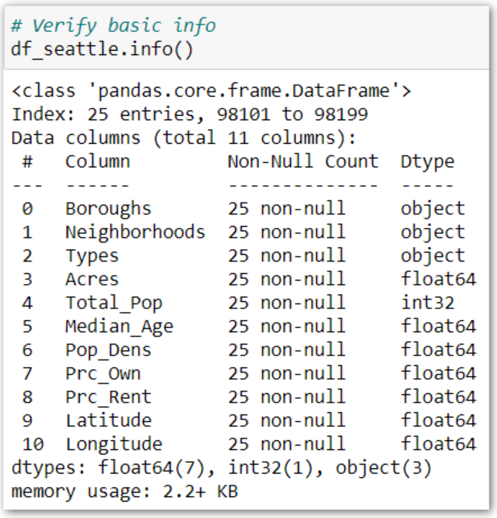
```
(25, 11)
```

	Boroughs	Neighborhoods	Types	Acres	Total_Pop	Median_Age	Pop_Dens	Prc_Own	Prc_Rent	Latitude	Longitude
Zipcode											
98101	Downtown, East	CommCore, PikePine, DennyTriangle	Urban Center Village	550.0	13578	38.133333	24.687273	0.217482	0.782518	47.610670	-122.33438
98102	LakeUnion	Eastlake	Residential Urban Village	200.2	5084	37.000000	25.394605	0.281591	0.718409	47.632870	-122.32253
98103	LakeUnion, Northwest	Wallingford, AuroraNorth, Fremont	Residential Urban Village, Hub Urban Village	798.6	15489	33.566667	19.395192	0.335741	0.664259	47.671346	-122.34166
98104	Downtown, East	MadisonMiller, FirstHill, PioneerSquare	Residential Urban Village, Urban Center Village	515.8	14999	39.833333	29.079100	0.169605	0.830395	47.602520	-122.32855

3.2.5 - Checkpoint

Verify the dataset looks good. Save it out to CSV locally. Noticed that field Total_Pop was type "float". Converted it over to type "Integer" so no trailing decimal point (cannot have a fractional person). Fig. 3.2.5.a shows the code that caught the data type error, and Fig. 3.2.5.b shows the code to change the data type.

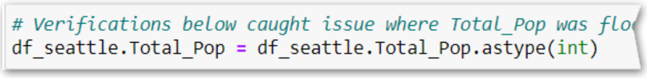
Fig. 3.2.5.a – Validate Final DataFrame



```
# Verify basic info
df_seattle.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 25 entries, 98101 to 98199
Data columns (total 11 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Boroughs       25 non-null     object
 1   Neighborhoods  25 non-null     object
 2   Types          25 non-null     object
 3   Acres          25 non-null     float64
 4   Total_Pop      25 non-null     int32
 5   Median_Age     25 non-null     float64
 6   Pop_Dens       25 non-null     float64
 7   Prc_Own        25 non-null     float64
 8   Prc_Rent       25 non-null     float64
 9   Latitude       25 non-null     float64
 10  Longitude      25 non-null     float64
dtypes: float64(7), int32(1), object(3)
memory usage: 2.2+ KB
```

Fig. 3.2.5.b – Code to Fix Data Type (from Float to Int)



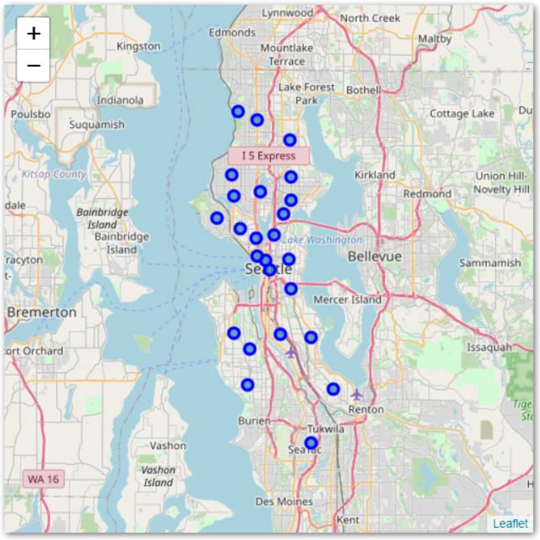
```
# Verifications below caught issue where Total_Pop was flo
df_seattle.Total_Pop = df_seattle.Total_Pop.astype(int)
```

3.3 - Data Exploration

3.3.1 - Explore Base Data Set in Map of Seattle

Next, we spun up a map of Seattle using the Folium library. Markers were established by dropping in our zip code based geo-coordinates from the DataFrame above to make sure the preliminary dataset all looks good. As it turned out, there was a gross outlier with one of the zip codes being more than 100 miles out of place. Using this technique was easy to catch and correct the typo. Afterwards, the map looks good with uniform coverage of all zip codes across the City of Seattle (Fig. 3.3.1).

Fig. 3.3.1 – Folium Map of Seattle with Zip Code Geo Coordinates Plotted



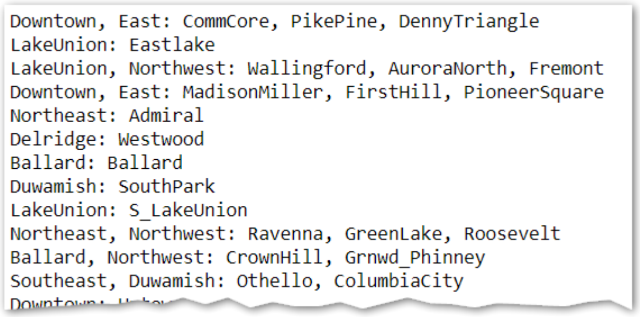
3.3.2 - Setup FourSquare for Use

For this IBM training course, I had to setup a set of one-time / throw-away credentials to the Foursquare API. Those are used as well as a generic function to call the FourSquare API that was pulled from the course lab code. Both are run, but do not yield any output.

3.3.3 - Call FourSquare API to Fetch Venues For Seattle Neighborhoods

Next, the FourSquare API was called for each Seattle neighborhood zip code. Our FourSquare request was limited to just the top 100 most common venues per request because that is more than sufficient for our analysis, and to avoid cap on daily request size. Our FourSquare request was also limited to 1,600 meters which is equivalent to 1 mile or 20 blocks at 80 meters per city block.

Fig. 3.3.3.a – FourSquare API Streaming Response

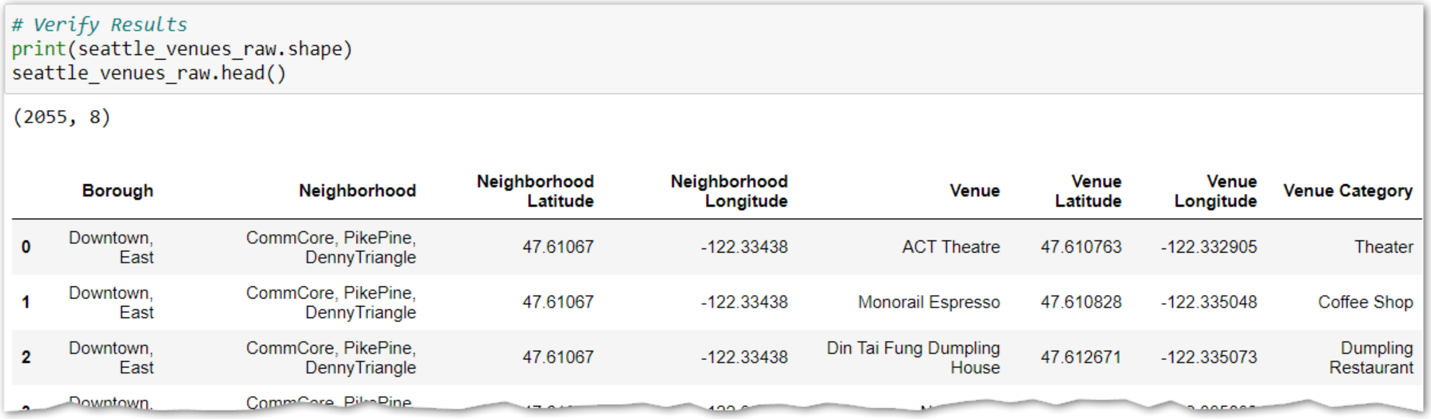


```
Downtown, East: CommCore, PikePine, DennyTriangle
LakeUnion: Eastlake
LakeUnion, Northwest: Wallingford, AuroraNorth, Fremont
Downtown, East: MadisonMiller, FirstHill, PioneerSquare
Northeast: Admiral
Delridge: Westwood
Ballard: Ballard
Duwamish: SouthPark
LakeUnion: S_LakeUnion
Northeast, Northwest: Ravenna, GreenLake, Roosevelt
Ballard, Northwest: CrownHill, Grnwd_Phinney
Southeast, Duwamish: Othello, ColumbiaCity
```

FourSquare streams its Response back line by line to Python / the Jupyter Notebook as it works thru each zip code in the API Request (Fig. 3.3.3.a). The final response is collected in a Pandas DataFrame consisting of 2,055 rows and 8 columns. The first four columns are the input parameters for Neighborhood (Borough, Name, Latitude, and Longitude). FourSquare appends four additional columns for Venue (Name, Latitude, Longitude, and Category) as shown in Fig. 3.3.3.b. Note that I originally passed in Borough and Neighborhood Name, but did not like the results downstream at the end with zip codes overlapping multiple neighborhoods in the denser areas.

Oh, and a side note…you might run into an issue where your firewall blocks the FourSquare API call out from Python or your Jupyter Notebook. If that happens and you get an error running the code, act accordingly.

Fig. 3.3.3.b - FoureSquare API Response

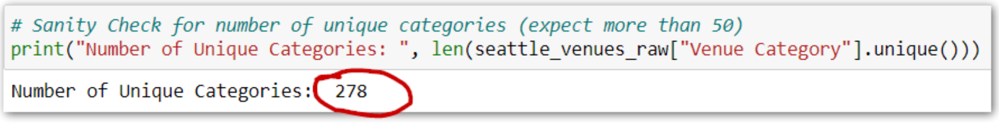


```
# Verify Results
print(seattle_venues_raw.shape)
seattle_venues_raw.head()
```

```
(2055, 8)
```

	Borough	Neighborhood	Neighborhood Latitude	Neighborhood Longitude	Venue	Venue Latitude	Venue Longitude	Venue Category
0	Downtown, East	CommCore, PikePine, DennyTriangle	47.61067	-122.33438	ACT Theatre	47.610763	-122.332905	Theater
1	Downtown, East	CommCore, PikePine, DennyTriangle	47.61067	-122.33438	Monorail Espresso	47.610828	-122.335048	Coffee Shop
2	Downtown, East	CommCore, PikePine, DennyTriangle	47.61067	-122.33438	Din Tai Fung Dumpling House	47.612671	-122.335073	Dumpling Restaurant

A quick sanity check reveals that we start with 278 unique venue categories. That seems reasonable.



```
# Sanity Check for number of unique categories (expect more than 50)
print("Number of Unique Categories: ", len(seattle_venues_raw["Venue Category"].unique()))
```

```
Number of Unique Categories:  278
```

3.3.4 - Feature Selection: Clean Venue Data

Thought we were done cleaning data back in Section 2? Not so fast. Turns out that the API Response JSON from FourSquare can use some cleanup too. Since this isn't technically source data, and it can't be requested until the source data was cleaned in Section 2, then we have to wrangle it here.

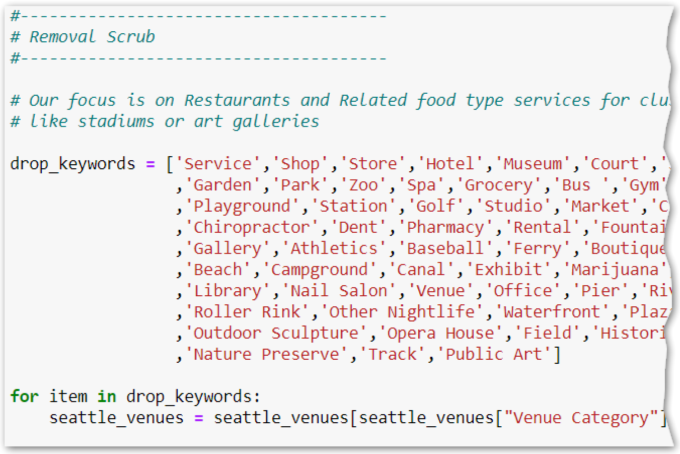
The first step was to make a copy (not a reference, but a truly separate copy) of the DataFrame housing our FourSquare venue data. This was important to avoid repeatedly making FourSquare calls as I debugged and then accidentally exceeding the daily cap in API requests.

Next, a **Venue Removal Scrub** was created (Fig. 3.3.4.a). I called the DataFrame.head(100) to walk the data and start either discarding rows that would clutter the research (don't care about museums and shops and hotels and schools). Our goal is to find clusters of other restaurants or food related businesses where there is not already a Orient restaurant existing. So, I built a large array of 77 venue category keywords to discard, and then fed them into a small two-line python loop that dropped the venues from 2,055 rows in the original response down to 1,067 rows of applicable venues. This tightened up accuracy.

Next, a **Venue Name Consolidation Scrub** was created (Fig. 3.3.4.b) to standardize 92 FourSquare Venue Group names. This ultimately collapsed the bar graph down to a more manageable set of 40+/- groups from the original 278 pulled down at the end of step 3.3.3.

Both these scrubs were highly iterative in that a first pass was made, and then 30-40 elements were added as I ran and re-ran the code over again and again below. As the output tables and graphs got refined, I'd see more and more labels that were too long or should be grouped with other labels. Creating both these as easily editable lists minimized copy-paste effort and risk.

Fig. 3.3.4.a – Venue Removal Scrub



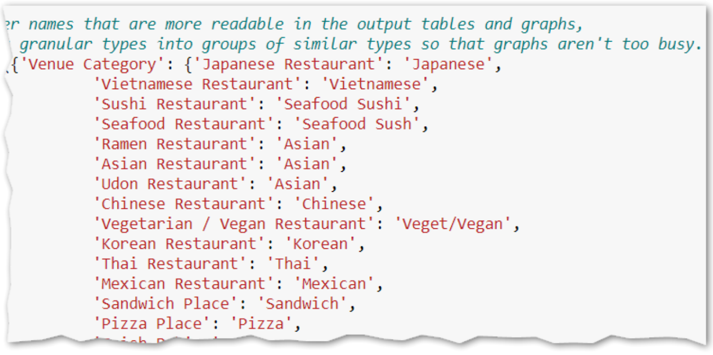
```
#--------------------------------------
# Removal Scrub
#--------------------------------------

# Our focus is on Restaurants and Related food type services for clu
# like stadiums or art galleries

drop_keywords = ['Service','Shop','Store','Hotel','Museum','Court','
                 ,'Garden','Park','Zoo','Spa','Grocery','Bus ','Gym'
                 ,'Playground','Station','Golf','Studio','Market','C
                 ,'Chiropractor','Dent','Pharmacy','Rental','Fountai
                 ,'Gallery','Athletics','Baseball','Ferry','Boutique
                 ,'Beach','Campground','Canal','Exhibit','Marijuana'
                 ,'Library','Nail Salon','Venue','Office','Pier','Ri
                 ,'Roller Rink','Other Nightlife','Waterfront','Plaz
                 ,'Outdoor Sculpture','Opera House','Field','Histori
                 ,'Nature Preserve','Track','Public Art']

for item in drop_keywords:
    seattle_venues = seattle_venues[seattle_venues["Venue Category"]
```

Fig. 3.3.4.b – Venue Name Consolidation Scrub



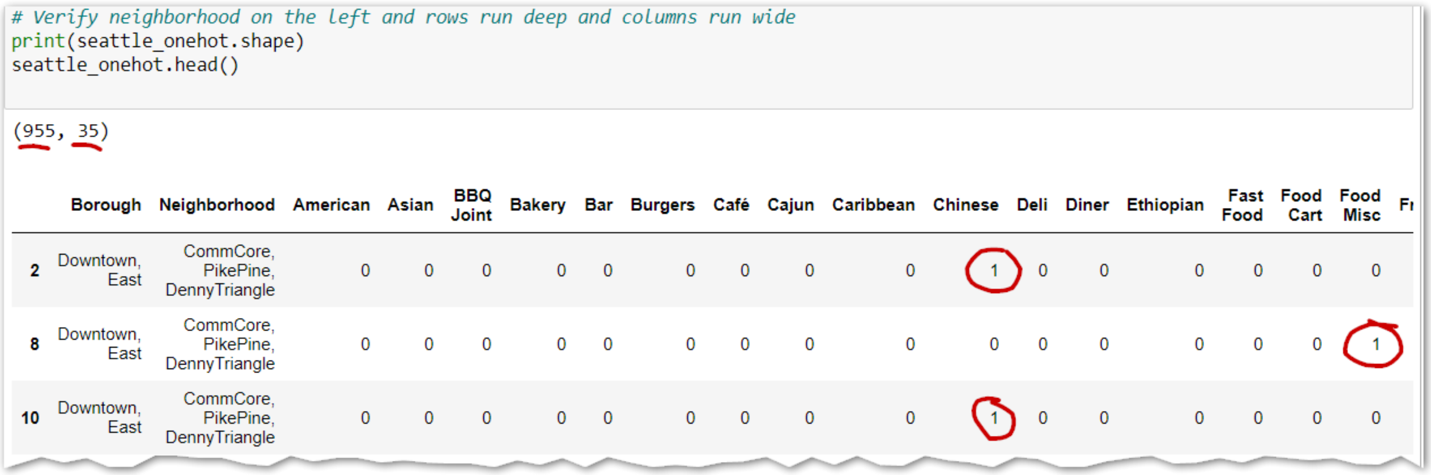
```
er names that are more readable in the output tables and graphs,
 granular types into groups of similar types so that graphs aren't too busy.
.{'Venue Category': {'Japanese Restaurant': 'Japanese',
         'Vietnamese Restaurant': 'Vietnamese',
         'Sushi Restaurant': 'Seafood Sushi',
         'Seafood Restaurant': 'Seafood Sush',
         'Ramen Restaurant': 'Asian',
         'Asian Restaurant': 'Asian',
         'Udon Restaurant': 'Asian',
         'Chinese Restaurant': 'Chinese',
         'Vegetarian / Vegan Restaurant': 'Veget/Vegan',
         'Korean Restaurant': 'Korean',
         'Thai Restaurant': 'Thai',
         'Mexican Restaurant': 'Mexican',
         'Sandwich Place': 'Sandwich',
         'Pizza Place': 'Pizza',
```

3.3.5 - Feature Selection: Calculate Venue Category Frequencies

Once the FourSquare Venue data has been cleaned, it is ready for analysis. The first step is to create a temporary analysis DataFrame with just the Zipcode along with all 33 remaining Venue Categories (dropping from 278 before the cleanup scrubs were run). Each of the 995 rows represents a Venue we are interested in (restaurant related). Within each row is the borough and neighborhood for that restaurant as well as a "1" put in the Venue Category column applicable to that restaurant. All other identifying column data is removed. The data looks like below in the Jupyter Notebook run results (Fig. 3.3.5.a).

The magic happens in the next step where the DataFrame of 995 rows is collapsed down into just 25 rows (grouped by zip code, one row each). The Venue Category columns all remain, but the individual 1's are all rolled up int means (percent frequency) per row. This means in the example run below that row 98101 has 3.45% Asian restaurants represented for all venues within a 20 block radius of the zip code centroid (Fig. 3.3.5.b).

Fig. 3.3.5.a – All 995 Scrubbed Venues from FourSquare are flattened out and assigned a "1" in the appropriate Venue Category

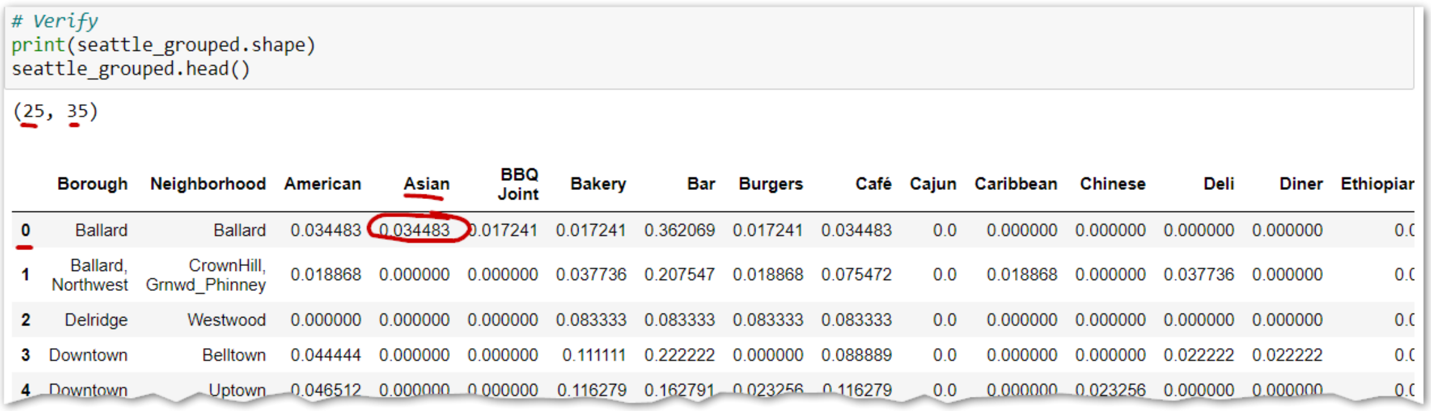


```
# Verify neighborhood on the left and rows run deep and columns run wide
print(seattle_onehot.shape)
seattle_onehot.head()
```

```
(955, 35)
```

	Borough	Neighborhood	American	Asian	BBQ Joint	Bakery	Bar	Burgers	Café	Cajun	Caribbean	Chinese	Deli	Diner	Ethiopian	Fast Food	Food Cart	Food Misc
2	Downtown, East	CommCore, PikePine, DennyTriangle	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	Downtown, East	CommCore, PikePine, DennyTriangle	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
10	Downtown, East	CommCore, PikePine, DennyTriangle	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Fig. 3.3.5.b – Group By Zip Code Rolls Up to 29 Rows, Columns no longer 0 or 1, but Frequency of Occurrence within the Group



```
# Verify
print(seattle_grouped.shape)
seattle_grouped.head()
```

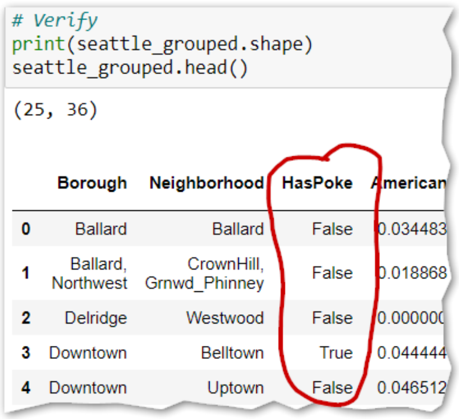
```
(25, 35)
```

	Borough	Neighborhood	American	Asian	BBQ Joint	Bakery	Bar	Burgers	Café	Cajun	Caribbean	Chinese	Deli	Diner	Ethiopiar
0	Ballard	Ballard	0.034483	0.034483	0.017241	0.017241	0.362069	0.017241	0.034483	0.0	0.000000	0.000000	0.000000	0.000000	0.(
1	Ballard, Northwest	CrownHill, Grnwd_Phinney	0.018868	0.000000	0.000000	0.037736	0.207547	0.018868	0.075472	0.0	0.018868	0.000000	0.037736	0.000000	0.(
2	Delridge	Westwood	0.000000	0.000000	0.000000	0.083333	0.083333	0.083333	0.083333	0.0	0.000000	0.000000	0.000000	0.000000	0.(
3	Downtown	Belltown	0.044444	0.000000	0.000000	0.111111	0.222222	0.000000	0.088889	0.0	0.000000	0.000000	0.022222	0.022222	0.(
4	Downtown	Uptown	0.046512	0.000000	0.000000	0.116279	0.162791	0.023256	0.116279	0.0	0.000000	0.023256	0.000000	0.000000	0.(

3.3.6 - Feature Selection: Add "HasPoke" Flag for Each Zip Code

Added a flag (Fig. 3.3.6) indicating whether the region has a Orient Place yet or not (in the top 278 venues). This is extraordinarily useful during final analysis to know whether or not a proposed region already has a Orient Place.

Fig. 3.3.6 – Code in Yellow, HasOrient Flag in Red Circle



```
# Verify
print(seattle_grouped.shape)
seattle_grouped.head()
```

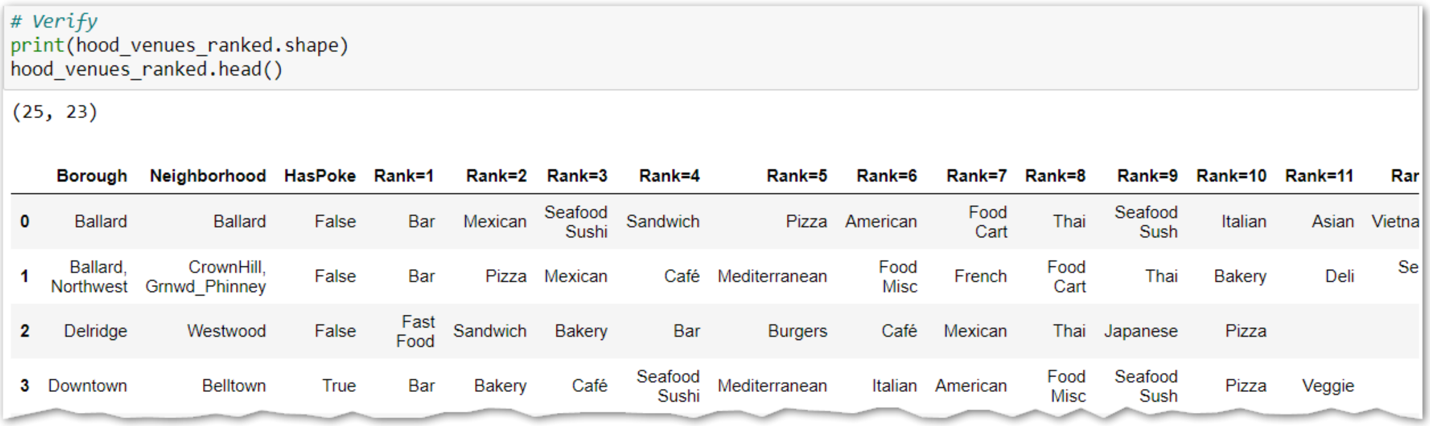
```
(25, 36)
```

	Borough	Neighborhood	HasPoke	American
0	Ballard	Ballard	False	0.034483
1	Ballard, Northwest	CrownHill, Grnwd_Phinney	False	0.018868
2	Delridge	Westwood	False	0.00000(
3	Downtown	Belltown	True	0.044444
4	Downtown	Uptown	False	0.046512

3.3.7 - Feature Selection: Rank Order Venue Categories

Next, a generic function "return_most_common_venues()" was added from the training courses. It is called from within a loop that walks each zip code to rank order each of the 23 consolidated Venue Categories columns by frequency of occurrence (sorts existing calculated mean value from highest to lowest). In the output DataFrame, the columns (Fig. 3.3.7.a) are clearly labeled "Rank=1", "Rank=2", and so on. Setting the data up this way enables accurate K-Means Clustering in subsequent steps.

Fig. 3.3.7.a – Rank Ordered Venue Categories by ZipCode



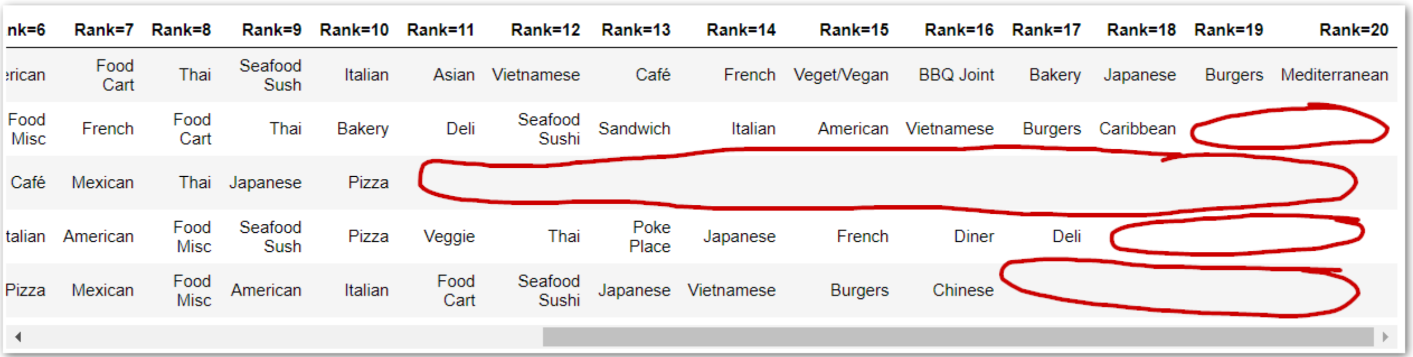
```
# Verify
print(hood_venues_ranked.shape)
hood_venues_ranked.head()
```

```
(25, 23)
```

	Borough	Neighborhood	HasPoke	Rank=1	Rank=2	Rank=3	Rank=4	Rank=5	Rank=6	Rank=7	Rank=8	Rank=9	Rank=10	Rank=11	Rar
0	Ballard	Ballard	False	Bar	Mexican	Seafood Sushi	Sandwich	Pizza	American	Food Cart	Thai	Seafood Sush	Italian	Asian	Vietna
1	Ballard, Northwest	CrownHill, Grnwd_Phinney	False	Bar	Pizza	Mexican	Café	Mediterranean	Food Misc	French	Food Cart	Thai	Bakery	Deli	Se
2	Delridge	Westwood	False	Fast Food	Sandwich	Bakery	Bar	Burgers	Café	Mexican	Thai	Japanese	Pizza		
3	Downtown	Belltown	True	Bar	Bakery	Café	Seafood Sushi	Mediterranean	Italian	American	Food Misc	Seafood Sush	Pizza	Veggie	

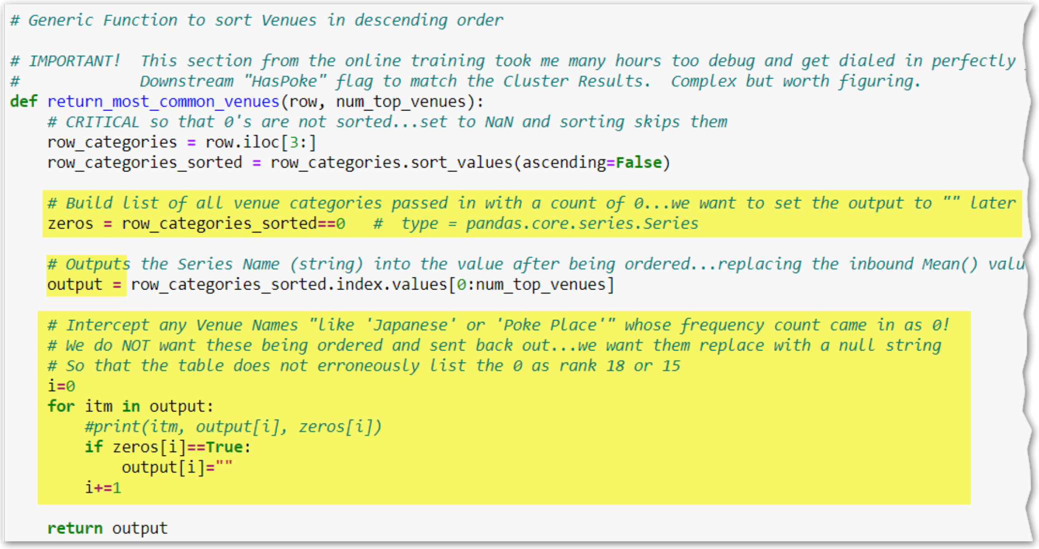
NOTE: There was a subtle bug with function "return_most_common_venues()" that took several hours to debug (see Fig. 3.3.7.b). So long as you only ever picked the top 10 ranked values, you likely never encountered the issue. However, because I wanted to pick the top 20, and I consolidated the 278 Venue Categories down to 33, then it became quite frequent that some of the top 20 ranks were empty, the count was zero. In those cases, the Venue Category was arbitrarily listed (if 6 of 20 were zeroes, they were all equal to the sort engine as zero and thus were sporadically placed). I noticed because the new flag "HasPoke" was always accurate, and yet on rows where it was equal to False, there was a ranked Orient value. After serious digging, it was the 0 count being arbitrarily passed thru. The fix (Fig. 3.3.7.c) was to set the output array value to blank string ("") where its corresponding series index value = 0.0000.

Fig. 3.3.7.b – Bug Fixed! Values in red are now properly blank, but used to have names for frequency = 0



nk=6	Rank=7	Rank=8	Rank=9	Rank=10	Rank=11	Rank=12	Rank=13	Rank=14	Rank=15	Rank=16	Rank=17	Rank=18	Rank=19	Rank=20
rican	Food Cart	Thai	Seafood Sush	Italian	Asian	Vietnamese	Café	French	Veget/Vegan	BBQ Joint	Bakery	Japanese	Burgers	Mediterranean
Food Misc	French	Food Cart	Thai	Bakery	Deli	Seafood Sushi	Sandwich	Italian	American	Vietnamese	Burgers	Caribbean		
Café	Mexican	Thai	Japanese	Pizza										
talian	American	Food Misc	Seafood Sush	Pizza	Veggie	Thai	Poke Place	Japanese	French	Diner	Deli			
Pizza	Mexican	Food Misc	American	Italian	Food Cart	Seafood Sushi	Japanese	Vietnamese	Burgers	Chinese				

Fig. 3.3.7.c – Code in Yellow was added to fix the Bug



```
# Generic Function to sort Venues in descending order

# IMPORTANT!  This section from the online training took me many hours too debug and get dialed in perfectly
#             Downstream "HasPoke" flag to match the Cluster Results.  Complex but worth figuring.
def return_most_common_venues(row, num_top_venues):
    # CRITICAL so that 0's are not sorted...set to NaN and sorting skips them
    row_categories = row.iloc[3:]
    row_categories_sorted = row_categories.sort_values(ascending=False)

    # Build list of all venue categories passed in with a count of 0...we want to set the output to "" later
    zeros = row_categories_sorted==0    #  type = pandas.core.series.Series

    # Outputs the Series Name (string) into the value after being ordered...replacing the inbound Mean() valu
    output = row_categories_sorted.index.values[0:num_top_venues]

    # Intercept any Venue Names "like 'Japanese' or 'Poke Place'" whose frequency count came in as 0!
    # We do NOT want these being ordered and sent back out...we want them replace with a null string
    # So that the table does not erroneously list the 0 as rank 18 or 15
    i=0
    for itm in output:
        #print(itm, output[i], zeros[i])
        if zeros[i]==True:
            output[i]=""
        i+=1

    return output
```

3.3.8 - Use K-Means to Generate 5 Clusters

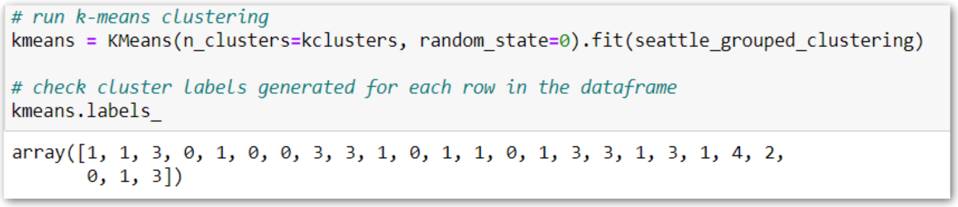
Next, the matrix of Venue Categories (columns "Rank=1" thru "Rank=20") was fed into the K-Means data model engine to try and find any patterns or relationships between neighborhoods based on the types of venues operating within.

For this project, the k-means clustering algorithm used earlier in the coursework was an appropriate algorithm. K-means partitions "n" observations (venues in our case) into "k" clusters (similar buckets) where each observation (venue) belongs to the cluster with the nearest mean. In our case, the nearest mean has nothing to do with geographic location, but rather has to do with zip codes containing the most similar traits such as having or not having a Orient restaurant (important enough that a specific Yes/No field was calculated for each zip code), having fewer or more clusters of restaurants, etc.

It was difficult to determine the hyperparameter "k". Initially, the default value of "3" seen in the course work and online examples was used. However, that didn't seem to break down into easily identifiable groups so we incremented to 4 and ultimately 5 clusters. Each of the five clusters are discussed below making it easier to understand how the value k=5 was arrived at. This was an unsupervised learning model because the dataset (venue categories) was unlabeled nominal data.

K-Means returns an array of cluster labels (numbers 0 thru 4) corresponding to the neighborhoods passed in. The output is shown below (Fig. 3.3.8.a); and the new DataFrame column in 3.3.8.b.

Fig. 3.3.8.a – Output Array of Cluster Labels from K-Means Model

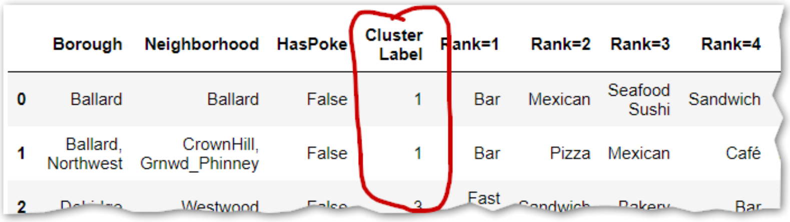


```
# run k-means clustering
kmeans = KMeans(n_clusters=kclusters, random_state=0).fit(seattle_grouped_clustering)

# check cluster labels generated for each row in the dataframe
kmeans.labels_
```

```
array([1, 1, 3, 0, 1, 0, 0, 3, 3, 1, 0, 1, 1, 0, 1, 3, 3, 1, 3, 1, 4, 2,
       0, 1, 3])
```

Fig. 3.3.8.b – Notice the New Cluster Label Column

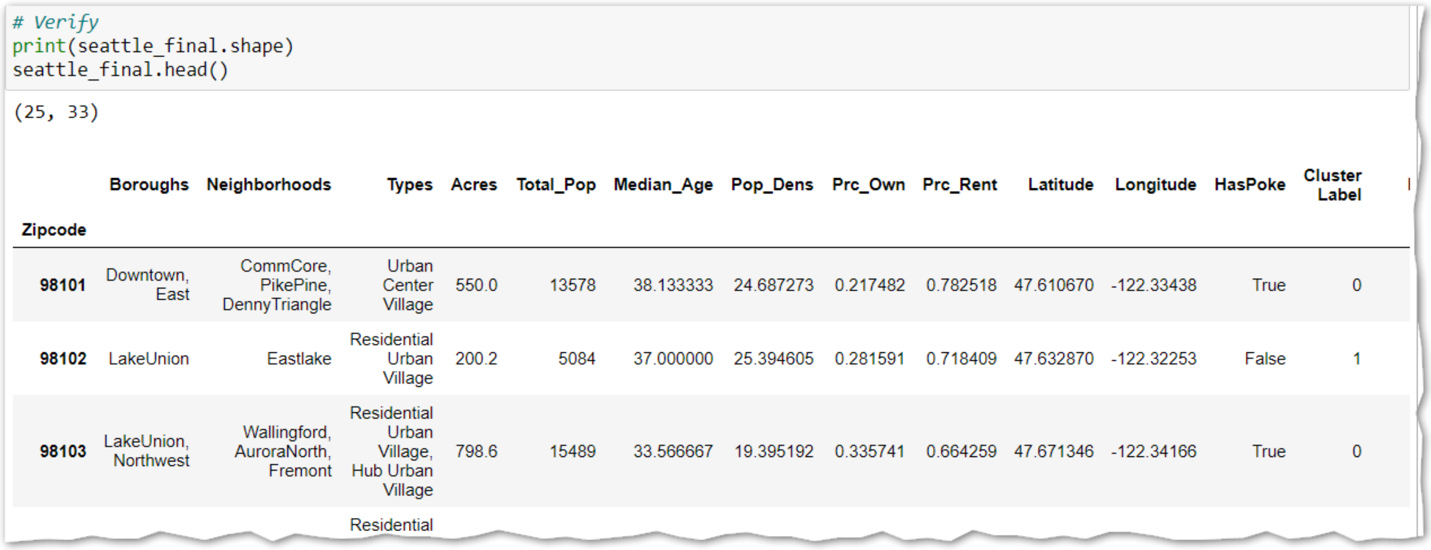


	Borough	Neighborhood	HasPoke	Cluster Label	Rank=1	Rank=2	Rank=3	Rank=4
0	Ballard	Ballard	False	1	Bar	Mexican	Seafood Sushi	Sandwich
1	Ballard, Northwest	CrownHill, Grnwd_Phinney	False	1	Bar	Pizza	Mexican	Café
2	Delridge	Westwood	False	3	Fast	Sandwich	Bakery	Bar

3.3.9 – Merge All Table Columns Back Together

Merge the aggregated neighborhood demographics data back together with the Venue Category ranked 1 - 20 columns and finally adding in the new Cluster Label column. Now all the information is in one place for analysis and decision making. Final DataFrame layout is shown below (Fig. 3.3.9). There are 25 rows (Neighborhoods rolled up to distinct zipcodes) and 33 columns (demographic attributes, ranked venue categories, etc.)

Fig. 3.3.9 – Final DataFrame After Re-Combining all the Columns



```
# Verify
print(seattle_final.shape)
seattle_final.head()
```

```
(25, 33)
```

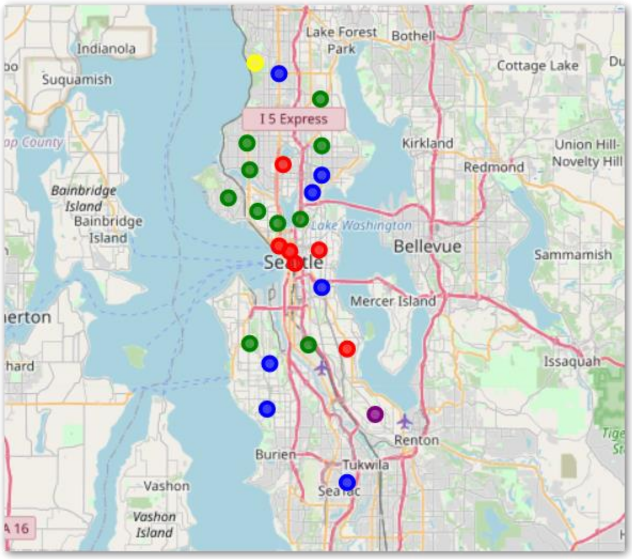
	Boroughs	Neighborhoods	Types	Acres	Total_Pop	Median_Age	Pop_Dens	Prc_Own	Prc_Rent	Latitude	Longitude	HasPoke	Cluster Label
Zipcode													
98101	Downtown, East	CommCore, PikePine, DennyTriangle	Urban Center Village	550.0	13578	38.133333	24.687273	0.217482	0.782518	47.610670	-122.33438	True	0
98102	LakeUnion	Eastlake	Residential Urban Village	200.2	5084	37.000000	25.394605	0.281591	0.718409	47.632870	-122.32253	False	1
98103	LakeUnion, Northwest	Wallingford, AuroraNorth, Fremont	Residential Urban Village, Hub Urban Village	798.6	15489	33.566667	19.395192	0.335741	0.664259	47.671346	-122.34166	True	0
			Residential										

4. Results

4.1 – Visualize the Data Clusters in a Map

Again we used Folium to map the neighborhood data. However, this time we also included the ClusterLabel (0-5) which was used to color the dots. Notice in the resulting map to the right (Fig. 4.1) that there are basically three clusters: (1) red dots Downtown, (2) green dots in the more residential areas where other restaurants are a bit more spread out, and (3) blue dots located at busier commercial districts in the outer lying areas of Seattle (denser restaurants than in the green dots, but less than the red dots). The Yellow and Purple dots are both single item outliers…need to investigate those more closely (or perhaps drop the k-clusters constant down to 4 or even 3.

Fig. 3.3.10 – Five Clusters, Mapped

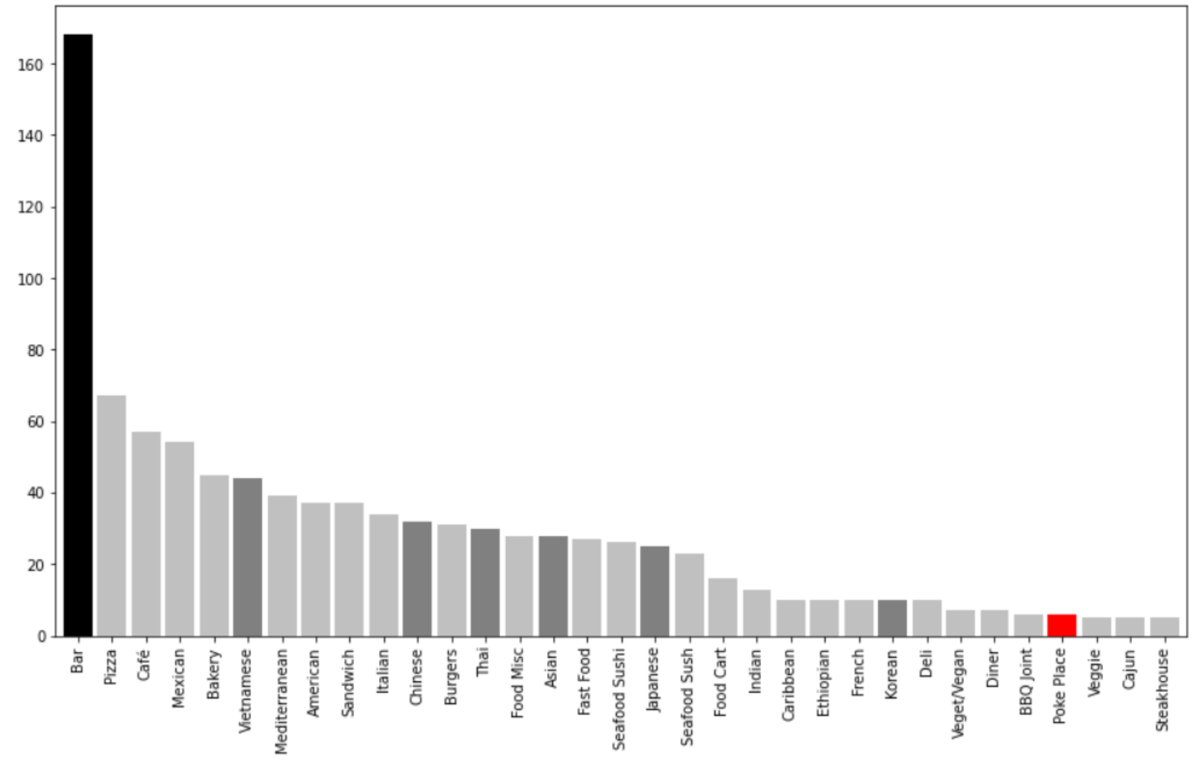


4.2 – Question "Does Seattle Have Room for Another Poke?"

To answer this question, we need to count the Venue Categories collected for all zip codes in the study, then sort them in descending order, and graph them with Python's default matplotlib. Results below in the bar chart (Fig. 4.2). Hint: Yes, there are many other restaurant types ahead of Orient Places.

Fig. 4.2 – There are approximately only 10 Orient places with 20 blocks of the zip codes we analyzed

red = Orient places, dark gray = other Asian restaurants, and black = bars…far and away the most numerous related venue



4.3 Examine the Five Clusters

4.3.1 Cluster 1: "Saturated Urban Centers, Orient Places Already Established"

In this cluster, every zip code has a Orient Restaurant; and they all rank 13th to 18th place out of the original 278 categories. These markets are probably saturated unless the "client" has something unique to offer and compete on. The Median age is 33-43. The percent renters is high, ranging from 68% up to an astronomical 83%.

4.3.2 Cluster 2: "The Target Locations"

In this cluster, there are (10) zip codes, and NOT ONE of them has a Orient Restaurant within 20 blocks! The Median age is relatively young, ranging between 31 to 39. The percent renters is generally in the 70's except for South Park and Crown Hill neighborhoods where ownership is surprisingly high -- of course we want to target the take-out crowd which tends to be renters. Many of these zip codes are good candidates for opening a Orient restaurant. These are the Green Dots back on the map above for a reason!

4.3.3 Cluster 3: "Outlier #1 – Too Sparse…Only 1 Zip Code and 3 Venue Categories"

This cluster is an outlier. There is only one zipcode in this cluster, and only 3 rankings for Venue Categories. It does not have a Orient place nearby. The median age is 35. The percent renters is almost 75%. There are Vietnamese and Mediterranean restaurants as well as Pizza places; but nothing else. This is probably not a good location for an upscale Orient Place.

4.3.4 Cluster 4: "Possible Target Zone"

In this cluster, there are (7) zip codes and NOT ONE of them has a Orient Restaurant within 20 blocks! The median age ranges from 28 (in the University District) up to 45 in Duwamish (manufacturer area). The Rental percent ranges from a low of 59% in Westwood up to a super high 94% in the University District. Pizza, Thai, Vietnamese, and Asian cuisine are the top restaurant related Venue Categories.

4.3.5 Cluster 5: "Outlier #2 - Too Sparse…Only 1 Zipcode and 4 Venue Categories”

This cluster is also an outlier. There is only one zipcode in the cluster, and only 4 rankings for Venue Categories. That means there are not very many restaurants in this area, but lots of homes and apartments. There is not yet a Orient Place here, but there are Chinese, American, Vietnamese, Asian, and Seafood/Sushi restaurants here…but little else. Although this might be a viable choice, Clusters 2 and 4 look far more promising where this might be a good special case with more analysis.

5. Observations & Recommendations

5.1 Cluster Analysis

Examination of the clusters readily indicates **Cluster #2** is best having all (10) zip codes absent of any Orient competition, yet has viable groups of restaurants near which to setup.. **Cluster #4** is similar having no Orient competition, but has fewer (7) available zip codes. **Cluster #1** should be avoided unless the client has special advantages because these dense urban areas are already saturated with Orient competition. **Clusters #3 and #5** are oddball outliers to be skipped, having just one zip code each and less than 4 venue categories (meaning few restaurant related venues).

5.2 Recommendations

Regarding the question “is there market potential for another Orient Restaurant in Seattle?”, the answer is absolutely Yes according to Fig. 4.2 above clearly showing the upside potential Orient restaurants have in the area. The analysis in this report covered 2,055 venues in the Seattle area, of which 955 were restaurant related. The analysis focused on venues within 20 blocks of the centroids of zip codes in the city of Seattle. The tiny red bar on graph 4.2 represents the actual count of Orient restaurants in Seattle as of the evening of 1/1/2021 when this paper was being written. There are just 10 Orient Places, with a few more possibly mis-classified in the 30 Sushi / Seafood restaurants and the 30-ish Japanese restaurants.
Regarding “what is the best location to open a new Orient restaurant in Seattle?”. Well it depends. The best zip codes according to cluster analysis are in clusters #2 and #4. From there, you can drill down into the demographic data to spot upwardly mobile, younger demographic folks with a higher propensity towards eating out. Since Orient would entail eating out, and Orient tends to be a little bit pricier than other dining options, then from Cluster #2, I’d recommend zip codes **98122** (Capital Hill), **98103** (Aurora North/Fremont), and **98121** (Belltown) as the best candidates.

5.3 Discussion

There are several areas of improvement that should be considered for this research project.

1. This was just a preliminary analysis that is part of an IBM Certification course. If this were a real project with a real paying client, then even more rigor would go into factoring in features such as population, median income, densities, percent rental vs. ownership, etc.
2. The k-means modeling worked properly; however, could be improved by weighting features such as bus stations heavier rather than removing them from the model.

3. Note that the 20 block radius (1,600 meters) may have left some gaps or geographic areas not covered by FourSquare venues in the analysis. As such, a follow-up study could be conducted to double or even triple the radius. Such a follow-up study would need to focus more on the outer less dense areas of Seattle so as to avoid overcounting the denser zipcodes by 3x, 4x, or even 5x in the downtown core.

4. Also, the FourSquare radius could be larger, then simply de-dup on the Venue returned name + latitude + longitude as a simple approach to increasing coverage but avoiding duplicates.

5. I find it hard to believe that zip code 98195 in the University District has no Orient places (according to results from FourSquare). Upon further digging, rank #6 is Japanese / Sushi restaurants. I suspect a better grouping logic earlier at section 3.3.4.b the Venue Name Consolidation Scrub would have been to roll Sushi restaurants into the "Orient Place" bucket as they are a close competitor. Alternatively, perhaps the limit of 100 in that dense commercia/retail area just dropped out the Orient places. Needs more research to improve accuracy

6. Conclusions

This project set out to determine whether it was feasible to open a new Orient restaurant in Seattle, and if yes then what geographic location would be best. Neighborhood data from the City of Seattle was used along with zip code data from a local company's website. That location data was augmented with venue data details from Foursquare and calculations applied in Python code. This base data set was loaded, cleaned, and prepared so that cluster modeling and other analysis could be performed. The Folium library was used for map related visualizations, and the standard python matplotlib was used for standard graphs. This was a fun, but complex project, enjoyed working through it.

7. References

[1] https://www.coursera.org/professional-certificates/ibm-data-science#howItWorks

[2] https://www.linkedin.com/in/naelqawass/

[3] https://nq.aibigdatasolutions.com/

[4] https://aibigdatasolutions.com/