

▼ HW2 - Transforms!

In this homework we'll be practicing some data aggregations, joins, and flattening JSON files in order to make inferences from two datasets!

The first file is a dataset with 25,000,000 movie reviews. These are already in flat files and are clean. But, we obviously want to do some filtering (e.g. remove movies that only have a review or two). And we'll want to do a couple of aggregations and joins to distill down those 25 million reviews into something understandable and useable.

The other dataset will have you going to Yelp and making a dev account. You'll then access various bits of information about local businesses and their reviews in order to figure out which ones are doing the best in Covid times.

▼ Wrangling and aggregating movie review data

The website MovieLens.com has a research group which provides open access to millions of reviews, for free! We're going to work with those data for this homework. [Feel free to check out the website here.](#) You can go and download the raw data, but in order to make things a bit easier, the files have been uploaded the Google drive for fast direct downloads.

There are two datasets we'll be working with.

- movies - this is a file of 60,000+ movies
- reviews - this is a file of 25 *million* individual reviews for the 60k movies

The goal for this section of the homework is to do two types of data aggregations that will allow for someone to make inferences on which movies were the most popular, reviewed, polarizing and were cult classics.

To do this we'll first start by making a simple data set that brings just overall review properties together with movies. We'll then do some deeper groupings to create a dataset that looks at the same properties but over time.

▼ Data first

Let's bring in our two files and libraries. The ratings file is understandably large. So it's a good idea to download it and then save a copy as something else and work with that. This way if you mess up you don't have to download it all over again

```
# Libraries
```

```
import pandas as pd
from datetime import datetime
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

```
# Movies data
```

```
movies = pd.read_csv('https://itsa322.s3.us-east-2.amazonaws.com/movies.csv')
```

```
movies
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
1	2	Jumanji (1995)	Adventure Children Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama Romance
4	5	Father of the Bride Part II (1995)	Comedy
...
62418	209157	We (2018)	Drama
62419	209159	Window of the Soul (2001)	Documentary
62420	209163	Bad Poems (2018)	Comedy Drama
62421	209169	A Girl Thing (2001)	(no genres listed)
62422	209171	Women of Devil's Island (1962)	Action Adventure Drama

```
# Ratings data
```

```
ratings = pd.read_csv('https://itsa322.s3.us-east-2.amazonaws.com/ratings.csv')
```

```
# Make a copy of ratings; Generally, it's good practice to keep original copies saved
ratings_backup = ratings.copy
```

▼ Explore your data - [3 points]

Below take some time to explore your data.

In one cell, check the following items for **both** datasets:

- Head and tail
- Shape
- Datatypes

In a new cell:

- The number of NaN values for the rating column of the ratings dataset

Task Do the head,tail, and shape operations all in one cell. Count the number of NaNs in another.

```
# head, tail, shape of both datastes
```

```
print(movies.head())
print(movies.tail())
print(movies.shape)
```

```
print(ratings.head())
print(ratings.tail())
print(ratings.shape)
```

```

      movieId      title \
0          1      Toy Story (1995)
1          2      Jumanji (1995)
2          3  Grumpier Old Men (1995)
3          4  Waiting to Exhale (1995)
4          5  Father of the Bride Part II (1995)

```

```

                                genres
0  Adventure|Animation|Children|Comedy|Fantasy
1                                Adventure|Children|Fantasy
2                                Comedy|Romance
3                                Comedy|Drama|Romance
4                                Comedy

```

```

      movieId      title      genres
62418  209157      We (2018)      Drama
62419  209159  Window of the Soul (2001)  Documentary
62420  209163      Bad Poems (2018)  Comedy|Drama
62421  209169      A Girl Thing (2001)  (no genres listed)
62422  209171  Women of Devil's Island (1962)  Action|Adventure|Drama
(62423, 3)

```

```

      userId  movieId  rating  timestamp
0          1        296      5.0  1147880044
1          1        306      3.5  1147868817
2          1        307      5.0  1147868828
3          1        665      5.0  1147878820
4          1        899      3.5  1147868510

```

```

      userId  movieId  rating  timestamp
25000090  162541    50872      4.5  1240953372
25000091  162541    55768      2.5  1240951998
25000092  162541    56176      2.0  1240950697
25000093  162541    58559      4.0  1240953434
25000094  162541    63876      5.0  1240952515
(25000095, 4)

```

```
# Get count of NaNs
print(movies.isna().sum())
print(ratings.isna().sum())
```

```
movieId    0
title      0
genres     0
dtype: int64
userId     0
movieId    0
rating     0
timestamp  0
dtype: int64
```

▼ Convert timestamp in ratings to a datetime. - [1.5 points]

One issue that you can see from your exploration is that the ratings only have a timestamp. This timestamp is measured in the number of seconds since 00:00:00 on January 1st, 1970. You'll need to convert this to a datetime in order to actually do our later data aggregations.

You use `pd.to_datetime` on timestamps like this. [For full details on the various ways to use this function please look at the Pandas documentation.](#) Briefly, this is how it's used.

```
>>> pd.to_datetime(1490195805, unit='s')
Timestamp('2017-03-22 15:16:45')
```

Task Make a new column in the movies dataframe called `review_dt` that contains the data from the timestamp column but converted to a datetime datatype.

Note, it's good practice to do this by first assigning the output to a test vector first, rather than directly adding the output to the dataframe. This allows you to make sure your operation did what you wanted before modifying your dataframe. Once you're confident of the output, you can assign the vector to the `reviews` dataframe as a new column.

For example

```
test_vec = pd.to_datetime(arguments)
test_vec # to check what it contains
```

```
# Make test vector
test_vec = pd.to_datetime(ratings['timestamp'], unit='s')
test_vec
```

```

0          2006-05-17 15:34:04
1          2006-05-17 12:26:57
2          2006-05-17 12:27:08
3          2006-05-17 15:13:40
4          2006-05-17 12:21:50
...
25000090   2009-04-28 21:16:12
25000091   2009-04-28 20:53:18
25000092   2009-04-28 20:31:37
25000093   2009-04-28 21:17:14
25000094   2009-04-28 21:01:55
Name: timestamp, Length: 25000095, dtype: datetime64[ns]

```

```

# Check vector and datatype to make sure it make sense.
test_vec.describe(datetime_is_numeric=True)

```

```

count          25000095
mean    2008-07-09 11:04:03.121552128
min          1995-01-09 11:46:49
25%          2002-01-23 00:54:05
50%          2007-12-28 18:59:35
75%          2015-11-11 01:29:01.500000
max          2019-11-21 09:15:03
Name: timestamp, dtype: object

```

```

# Now ad this to the ratings dataframe
ratings['review_dt'] = test_vec

```

```

# Check the head of your dataframe again
ratings.head()

```

	userId	movieId	rating	timestamp	review_dt
0	1	296	5.0	1147880044	2006-05-17 15:34:04
1	1	306	3.5	1147868817	2006-05-17 12:26:57
2	1	307	5.0	1147868828	2006-05-17 12:27:08
3	1	665	5.0	1147878820	2006-05-17 15:13:40
4	1	899	3.5	1147868510	2006-05-17 12:21:50

```

# What's the oldest and newest review? It should be 1995-01-09 and 2019-11-21, respec
# oldest_review = ratings.min()
oldest_rating = ratings[ratings['review_dt'] == ratings['review_dt'].max()]
print(f"Oldest review: {oldest_rating}")

newest_rating = ratings[ratings['review_dt'] == ratings['review_dt'].min()]

```

```
print(f"Newest review: {newest_rating}")
```

Oldest review:	userId	movieId	rating	timestamp	review_dt
13207877 85523	149406	4.5	1574327703	2019-11-21 09:15:03	
Newest review:	userId	movieId	rating	timestamp	review_dt
326761 2262	21	3.0	789652009	1995-01-09 11:46:49	
326767 2262	47	5.0	789652009	1995-01-09 11:46:49	

▼ Your first aggregation and join - [4.5 points]

The first aggregation and join I want you to do is at the whole movie level. Your movies dataframe should only have a single row for each movie, but there are obviously thousands of individual reviews for each of those movies. Our goal here is to produce some summary statistics about the reviews for each movie and then join them to the movies dataframe.

Task: Do the following:

- Create an aggregated dataframe called `ratings_by_movie`. This dataframe should be grouped by movie. Use `.agg()` to calculate the mean, standard deviation ('std'), and the number of reviews for each movie.
- Rename the columns of that dataframe 'movieId', 'rating_mean', 'rating_std', and 'rating_count'
- Join this new `ratings_by_movie` dataframe such that it attaches all those summary statistics to their corresponding movies from the 'movies' dataframe.
- Call the joined dataframe `movies_with_ratings`

```
# Make ratings_by_movie
# ratings_by_movie = ...(..., as_index=False)....(...)
ratings_by_movie = ratings.groupby(['movieId'], as_index=False).agg({'rating': ['mean'

# Check it
ratings_by_movie.head()
```

```
# Rename columns
ratings_by_movie.columns = ['movieId', 'rating_mean', 'rating_std', 'rating_count']
ratings_by_movie.head(5)
```

	movieId	rating_mean	rating_std	rating_count
0	1	3.893708	0.921552	57309
1	2	3.251527	0.959851	24228
2	3	3.142028	1.008443	11804
3	4	2.853547	1.108531	2523
4	5	3.058434	0.996611	11714

```
# Join it and call movies with ratings
# Example: songs_agg = songs_agg.merge(artist_info, left_on='artist_name', right_on='artist_name')
movies_with_ratings = movies.merge(ratings_by_movie, left_on='movieId', right_on='movieId')

# Check movies_with_ratings.
movies_with_ratings.head(5)
```

	movieId	title	genres	rating_mean	rating_std
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy	3.893708	0.921552
1	2	Jumanji (1995)	Adventure Children Fantasy	3.251527	0.959851
2	3	Grumpier Old Men (1995)	Comedy Romance	3.142028	1.008443

Is your merged dataframe 62423 rows × 6 columns?

```
movies_with_ratings.shape

(62423, 6)
```

▼ Filtering and more transformations - [3 points]

Now we want to clean up this dataset a bit and then do a couple more transforms. One issue you can see from your check above is that many movies only have one rating. We're going to choose to

set a minimum number of reviews needed to be included. We also want to do some binning where movies with certain ratings levels

Task: Please do the following operations

- Filter `movies_with_ratings` so it only contains movies that have at least 10 ratings
- Use the function `cut()` to automatically bin our `rating_mean` column into three groups of 'bad', 'fine', or 'good' movies. Call this `rating_group`.
- Use the same function to take the standard deviation in rating and make three groups of 'agreement', 'average', 'controversial'. Thus, movies with low standard deviation have agreement in the rating, while movies with high standard deviation have controversy in the

```
# Filter first being sure to overwrite dataframe
movies_with_ratings = movies_with_ratings[(movies_with_ratings['rating_count'] >= 10)]
movies_with_ratings.head()
```

	movieId	title	genres	rating_mean	rating_stddev
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy	3.893708	0.169906
1	2	Jumanji (1995)	Adventure Children Fantasy	3.251527	0.171294
2	3	Grumpier Old Men (1995)	Comedy Romance	3.142028	1.169306

```
# Check how many rows you're left with. You should have a little over 24000
movies_with_ratings.shape

(24330, 6)
```

I didn't show you how to use `cut()` in the lesson, but it's a transform just like anything else. You could use a `np.where()` statement like we did, but `cut()` is a bit easier. All it does is take 1) a column as the first argument, 2) the number of bins you want to group it in as the second argument, and then 3) the labels you want to give those bins as the third. It automatically divides them up into equal sized bins.

For example, if I make the following list:

```
rating = [2, 4, 9, 8, 5, 3, 6, 10, 2, 1, 6, 7]
```

And run `cut()` on it with three bins and levels 'bad', 'fine', and 'good':

```
pd.cut(rating, 3, labels=['bad', 'fine', 'good'])
```


I get a return of:

```
[bad, bad, good, good, fine, ..., good, bad, bad, fine, fine]
Length: 12
Categories (3, object): [bad < fine < good]
```

Note how it orders them for you based on the order of the labels.

```
# If you want to test it!
rating = [2, 4, 9, 8, 5, 3, 6, 10, 2, 1, 6, 7]
pd.cut(rating, 3, labels=['bad', 'fine', 'good'])

['bad', 'bad', 'good', 'good', 'fine', ..., 'good', 'bad', 'bad', 'fine',
'fine']
Length: 12
Categories (3, object): ['bad' < 'fine' < 'good']

# Now make 'bad', 'fine', 'good' levels for ratings
# Assign to new column called 'rating_group'
# ...['...'] = pd.cut(...['...'], 3, labels=['bad', 'fine', 'good'])
rating_group = pd.cut(movies_with_ratings['rating_mean'], 3, labels=['bad', 'fine', 'good'])
movies_with_ratings['rating_group'] = rating_group

# Check it
movies_with_ratings
```

movieId title genres rating_mean

Do Toy Story and Jumanji have 'good' ratings? Does Grumpier Old Men have a 'fair' rating?

(1995)

```
# Now use cut() again to create your ratings_agreement column.
```

```
# Use three bins and order of 'agreement', 'average', and 'controversial'
```

```
movies_with_ratings.head()
```

```
ratings_agreement = pd.cut(movies_with_ratings['rating_mean'], 3, labels=['agreement',
```

```
movies_with_ratings['ratings_agreement'] = ratings_agreement
```

waiting to

```
# Check to make sure that your bin categories make sense. e.g. 'good' movies should have
```

```
movies_with_ratings.head()
```

```
# Interesting trivia, at least to me, Jumanji was made in my home state and filmed about
```

	movieId	title	genres	rating_mean	ratings_agreement
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy	3.893708	0.
1	2	Jumanji (1995)	Adventure Children Fantasy	3.251527	0.
2	3	Grumpier Old Men (1995)	Comedy Romance	3.142028	1.

▼ Exploring our data

Making bins like this allows us to figure out things like which movies are both bad, but have differing opinions on. For example, we can filter out movies that are in the bad category but have a lot of controversy about those ratings. This could mean they're 'cult classic' movies where despite the low rating some people actually really love the movies.

A dataset like this could be used in a recommendation engine where if you see people liking these 'bad but good' movies you could suggest others that meet the same criteria.

Task: There are no points here - only code to let you make figures to see what the dataset could be used for. Also, if this code works it means you probably did your answers above are right :)

```
bad_but_good = movies_with_ratings[(movies_with_ratings['rating_group'] == 'bad') &
                                     (movies_with_ratings['ratings_agreement'] == 'controversial') &
                                     (movies_with_ratings['rating_count'] >= 100)]
```

```
bad_but_good.head()
```

	movieId	title	genres	rating_mean	rating_std	rating_count	rating_group
--	---------	-------	--------	-------------	------------	--------------	--------------

Birdemic: Shock and Terror :D - I think I would prefer a Birdemic to a pandemic!

▼ Grouping within years - [3 point]

Now that we've done our overall grouping by movie, let's get a bit more detail about these ratings. Specifically, let's engineer a dataset that breaks down the average rating not only by movie, but also by the year the person provided the review. This would allow for someone to see which movies continue to do well over time, which ones become more popular, and which ones don't age well!

Task: You're going to do the following steps:

- Create a new `ratings_by_movie` that groups both by `movieId` but also by your `review_dt` column. I want you to group into year intervals.
- Join `movies` to `ratings_by_movie` so that you have the summary review statistics for each year the movie has been out
- Clean up and filter your dataframe

First, create `ratings_by_movie`. You can group by two levels by just adding a list of what levels you want to group by in the `groupby()` statement. I'll give you some help there, but you have to complete the rest in order to group by `movieId` first and `review_dt` second.

```
# Note the groupby syntax. I first am grouping by movieId
# But then also am calling dt.year on our datetime column
# This will then tell Python to do the aggregations within year as well
# ... = .....(['...', ratings['review_dt'].dt.year]).agg(...)
ratings_by_movie = ratings.groupby([ratings['movieId'], ratings['review_dt'].dt.year])

# Check
ratings_by_movie
```

		rating		
		mean	std	count
movieId	review_dt			
1	1996	4.132756	0.884990	6237
	1997	3.872500	0.894079	6000
	1998	3.889515	0.944972	887
	1999	3.975140	0.913960	2494

You need to rename columns, but this is a bit trickier as you have two levels of your dataframe index. I'm going to give you the code below. But, what it's doing is resetting that one level of the index `review_dt` and putting it back as a regular column. I'm then renaming the resulting columns.

```

209159    2019    3.000000    NaN    1
# Reset index
ratings_by_movie = ratings_by_movie.reset_index(level = 'review_dt')
ratings_by_movie.columns = ['year', 'rating_mean', 'rating_std', 'rating_count']
ratings_by_movie

```

	year	rating_mean	rating_std	rating_count	
movieId					
1	1996	4.132756	0.884990	6237	
	1997	3.872500	0.894079	6000	
	1998	3.889515	0.944972	887	
	1999	3.975140	0.913960	2494	
	2000	4.136634	0.865127	3535	
...	
209157	2019	1.500000	NaN	1	
209159	2019	3.000000	NaN	1	
209163	2019	4.500000	NaN	1	
209169	2019	3.000000	NaN	1	
209171	2019	3.000000	NaN	1	

323737 rows x 4 columns

```

# Now join the movie dataframe onto ratings_by_movie
# Example from above: movies_with_ratings = movies.merge(ratings_by_movie, left_on='movieId', right_on='movieId')
ratings_by_movie = movies.merge(ratings_by_movie, left_on='movieId', right_on='movieId')

```

```
# Check
ratings_by_movie.head()
```

	movieId	title	genres	year	rating_mean
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy	1996.0	4.132756
1	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy	1997.0	3.872500
		Tov			

```
# How many rows are there in the resulting dataframe?
ratings_by_movie.shape
# 327113 rows
```

```
(327113, 7)
```

In your dataframe do you see Toy Story having a mean rating of 4.132756 in 1996? Is your dataframe 323737 rows \times 7 columns?

Yes, mine matches.

▼ A quick plot

Now you have a dataset where one could explore how movies have done over time. I've made a couple plots below to show you what I mean.

```
# What movie has the max rating?
movies_with_ratings.loc[movies_with_ratings['rating_count'] == movies_with_ratings['rating_count'].max()]
```

movieId	title	genres	rating_mean	rating_std	rating_count
356	Forrest Gump	Drama	8.8	0.25	80000

```
# Grabbing just the ID for that movie.
movies_with_ratings['movieId'][movies_with_ratings['rating_count'] == movies_with_ratings['rating_count'].max()]

356
```

```
# Forrest gump has been reviewed over 80000 times!
# Let's extract that movie id to an object and then make just that dataframe
```

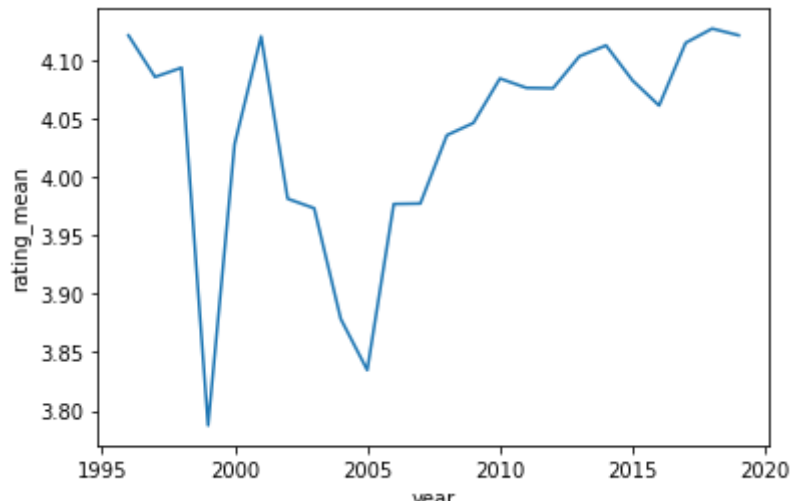
```
most_viewed_id = movies_with_ratings['movieId'][movies_with_ratings['rating_count'] ==
```

```
most_viewed_df = ratings_by_movie[ratings_by_movie['movieId'] == most_viewed_id]
most_viewed_df
```

	movieId	title	genres	year	rating_mean	rating_st
7927	356	Forrest Gump (1994)	Comedy Drama Romance War	1996.0	4.120873	0.93102
7928	356	Forrest Gump (1994)	Comedy Drama Romance War	1997.0	4.085352	0.91986
7929	356	Forrest Gump (1994)	Comedy Drama Romance War	1998.0	4.093245	1.04601
7930	356	Forrest Gump (1994)	Comedy Drama Romance War	1999.0	3.787516	1.16351
7931	356	Forrest Gump (1994)	Comedy Drama Romance War	2000.0	4.027959	1.05172
7932	356	Forrest Gump (1994)	Comedy Drama Romance War	2001.0	4.119985	0.99390
7933	356	Forrest Gump (1994)	Comedy Drama Romance War	2002.0	3.981262	1.01471
7934	356	Forrest Gump (1994)	Comedy Drama Romance War	2003.0	3.973144	0.97323
7935	356	Forrest Gump (1994)	Comedy Drama Romance War	2004.0	3.878659	0.95127
7936	356	Forrest Gump (1994)	Comedy Drama Romance War	2005.0	3.834838	0.95364
7937	356	Forrest Gump (1994)	Comedy Drama Romance War	2006.0	3.976765	0.87174
		Forrest				

```
# A quick lineplot shows that although forest gump has a really high rating on average
sns.lineplot(data = most_viewed_df, x = 'year', y = 'rating_mean')
```

<matplotlib.axes._subplots.AxesSubplot at 0x7f163d1ac6d0>



You can also look up some of your favorite movies. I actually love the movie 'Dredd', even though many people hated it. We can search in the titles for movies we like, and then call that ID to filter a new dataframe. We can then make some plots

```
# Search for Dredd
# Dredd is so bad, it is good. I too am a fan.
# Prometheus, technically a really bad movie. I have watched it at least 6 times. :-}
movies_with_ratings[movies_with_ratings['title'].str.contains('Dredd')]
```

	movieId	title	genres	rating_mean	rating_std	rating_count
171	173	Judge Dredd (1995)	Action Crime Sci-Fi	2.555360	1.039140	14758.0

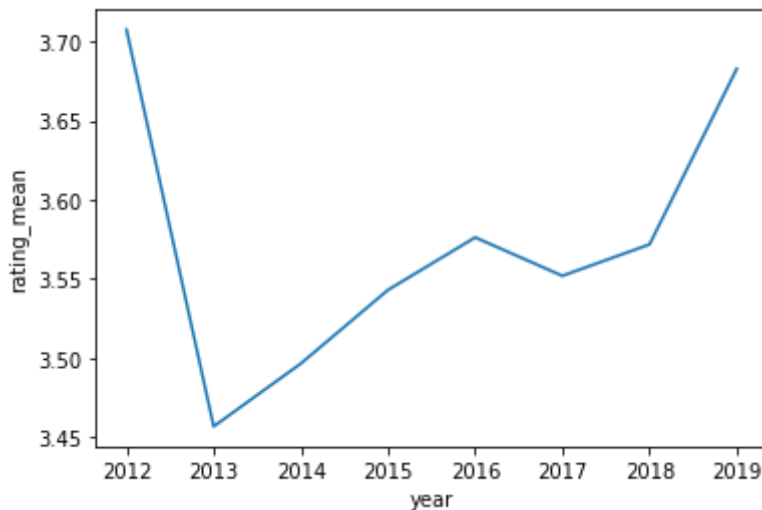
```
# Call the dredd to make a dataframe
dredd_id = movies_with_ratings['movieId'][movies_with_ratings['title'].str.contains('Dredd')]
dredd_df = ratings_by_movie[ratings_by_movie['movieId'] == dredd_id]
dredd_df
```

```

movieId  title  genres  year  rating_mean  rating_std  rating_cou
# Our plot shows that the year after release it was reviewed pretty poorly,
# but the score gradually grew as people realized how awesome it was :)
sns.lineplot(x = 'year', y = 'rating_mean', data = dredd_df)

```

<matplotlib.axes._subplots.AxesSubplot at 0x7f163d1025d0>



▼ Flattening JSON data from Yelp

For this second part of the assignment we're going to be using the Yelp API via the python package `yelpapi`. Like with the Spotify API, you need to go get a yelp developer account here:

<https://www.yelp.com/fusion>. When you try to create an app, it asks for name, description and contact info. Then it gives you clientid and api key.

The goal will be to make a dataset of local taco places that are doing well during Coronavirus. Specifically, we want to make a dataset that takes their average score since they've been open and compares it to the average score of the last three reviews. Shops that have been doing well should hopefully have these two averages be similar.

There will be three main steps to this process:

- First you'll search by location type and get aggregate information for everything that falls in the ice cream category.
- Second you'll get reviews for all those locations. Yelp only returns three reviews when you call an ID, but that still works. Given you can only query one ID at a time, you'll need to write a loop to create a dataframe of all the reviews.
- Third, you'll aggregate the latest review information to see how their average review score compares to their overall average review score.

```
# Mount google drive
```



```
import os, sys
from google.colab import drive
drive.mount('/content/mnt')
nb_path = '/content/notebooks'
os.symlink('/content/mnt/My Drive/Colab Notebooks', nb_path)
sys.path.insert(0, nb_path) # or append(nb_path)
```

Mounted at /content/mnt

Install yelpapi once. Tomorrow, you can skip this.

```
!pip install --target=$nb_path yelpapi
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheelhouse/pypi
Collecting yelpapi
  Downloading yelpapi-2.5.0-py3-none-any.whl (7.4 kB)
Collecting requests
  Downloading requests-2.28.1-py3-none-any.whl (62 kB)
    |████████████████████| 62 kB 1.2 MB/s
Collecting certifi>=2017.4.17
  Downloading certifi-2022.9.24-py3-none-any.whl (161 kB)
    |████████████████████| 161 kB 10.9 MB/s
Collecting idna<4,>=2.5
  Downloading idna-3.4-py3-none-any.whl (61 kB)
    |████████████████████| 61 kB 115 kB/s
Collecting charset-normalizer<3,>=2
  Downloading charset_normalizer-2.1.1-py3-none-any.whl (39 kB)
Collecting urllib3<1.27,>=1.21.1
  Downloading urllib3-1.26.12-py2.py3-none-any.whl (140 kB)
    |████████████████████| 140 kB 36.6 MB/s
Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
Successfully installed certifi-2022.9.24 charset-normalizer-2.1.1 idna-3.4 requests-2.28.1 yelpapi-2.5.0
WARNING: Target directory /content/notebooks/requests already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/idna-3.4.dist-info already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/requests-2.28.1.dist-info already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/urllib3-1.26.12.dist-info already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/certifi-2022.9.24.dist-info already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/yelpapi-2.5.0.dist-info already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/certifi already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/charset_normalizer already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/charset_normalizer-2.1.1.dist-info already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/yelpapi already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/urllib3 already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/idna already exists. Specify --upgrade
WARNING: Target directory /content/notebooks/bin already exists. Specify --upgrade
```

Now enter your API key so you can make requests

```
from yelpapi import YelpAPI
from pandas.io.json import json_normalize
# yelp_api = YelpAPI('ENTER YOUR KEY HERE')
"""
Yelp:
client id: ZDxy7f5b7NxwDdo7_T4TCg
api key: Wozh1kzMBWNj_Sdlofi5xHjaKFw18CXLPbObgmxxzMHdat16ERmlUsDP2F7kWIlbwRp2ddfu3Tka1q
```

```
"""
```

```
api_key = 'Wozh1kzMBWNj_Sdlofi5xHjaKFwl8CXLPbObgmxxzMHdatl6ERmlUsDP2F7kWIlbwRp2ddfU3Tka
yelp_api = YelpAPI(api_key)
```

Making calls to with Yelp API

There are many functions in the `yelpapi` package. The first one we'll use is `search_query()`. You can put in a term you want to search for followed up by the location and it'll give you all the locations that match. For example, the following would search for all ice cream places here in Tucson.

```
ice = yelp_api.search_query(term = 'ice cream', location = 'Tucson, AZ')
```

This will give you a response of all the ice cream places in Tucson. It'll be in JSON form so will need flattening before being useful.

▼ Getting all taco shops and flattening - [3 points]

Task: Start by making a dataframe that uses the `search_query()` function to search using the term 'taco'. Call this `taco_shops`. After that, flatten the json results to `taco_shops_df`.

```
# search for taco shops and store to taco_shops
taco_shops = yelp_api.search_query(term='taco', location='Tucson, AZ')
```

```
# Look at taco_shops
taco_shops
```

```
{
  'coordinates': {'latitude': 32.16826, 'longitude': -110.9772},
  'transactions': ['pickup', 'delivery'],
  'price': '$',
  'location': {'address1': '4573 S 12th Ave',
    'address2': '',
    'address3': None,
    'city': 'Tucson',
    'zip_code': '85714',
    'country': 'US',
    'state': 'AZ',
    'display_address': ['4573 S 12th Ave', 'Tucson, AZ 85714']},
  'phone': '+15203006289',
  'display_phone': '(520) 300-6289',
  'distance': 9562.690492450478,
  'id': 'W-qxgFKtw8TlibinTp5cYw',
  'alias': 'momos-tucson',
  'name': 'Momo's',
  'image_url': 'https://s3-
media4.fl.yelpcdn.com/bphoto/9tTn6Mid_U7Nl90nwpLNUA/o.jpg',
```

```

'is_closed': False,
'url': 'https://www.yelp.com/biz/momos-tucson?adjust\_creative=ZDxy7f5b7NxxwDdo7\_T4TCg&utm\_campaign=yelp\_api\_v3&utm\_medium=api\_v3\_campaign\_yelp\_android',
'review_count': 38,
'categories': [{'alias': 'mexican', 'title': 'Mexican'}],
'rating': 4.5,
'coordinates': {'latitude': 32.22755, 'longitude': -110.94414},
'transactions': [],
'location': {'address1': '1838 E 6th St',
'address2': '',
'address3': None,
'city': 'Tucson',
'zip_code': '85719',
'country': 'US',
'state': 'AZ',
'display_address': ['1838 E 6th St', 'Tucson, AZ 85719']},
'phone': '',
'display_phone': '',
'distance': 2279.119825738216},
{'id': 'AVBqx8FaXza6x-bb_FLOTQ',
'alias': 'anita-street-market-tucson',
'name': 'Anita Street Market',
'image_url': 'https://s3-media3.fl.yelpcdn.com/bphoto/dfr0Sk0lcDv28kAQEaQN6Q/o.jpg',
'is_closed': False,
'url': 'https://www.yelp.com/biz/anita-street-market-tucson?adjust\_creative=ZDxy7f5b7NxxwDdo7\_T4TCg&utm\_campaign=yelp\_api\_v3&utm\_medium=api\_v3\_campaign\_yelp\_android',
'review_count': 188,
'categories': [{'alias': 'mexican', 'title': 'Mexican'}],
'rating': 4.5,
'coordinates': {'latitude': 32.2329, 'longitude': -110.98112},
'transactions': ['pickup', 'delivery'],
'price': '$$',
'location': {'address1': '849 Anita Ave',
'address2': '',
'address3': ''},

```

What keys are present?

```
taco_shops.keys()
```

```
dict_keys(['businesses', 'total', 'region'])
```

Now use the `json_normalize` function to flatten `taco_shops`. Note that you need to select the key that contains the businesses when flattening. But this one is easier than the examples from the homework in that you don't need to provide any other arguments.

Store the result as `taco_shops_df`

After that select only the columns 'id', 'alias', 'name', 'review_count', and 'rating'.

```
# Flatten to taco_shops_df
taco_shops_df = pd.json_normalize(taco_shops['businesses'])
taco_shops_df.head() # Check
```

		id	alias	name	image_url
0	jmwasbZfgj3honf79qKsnA		street-taco-and-beer-co-tucson	Street-Taco and Beer Co.	https://s3-media4.fl.yelpcdn.com/bphoto/VOaRTn...
1	o3woQWQ-0HxFftItIEeNdw		el-rustico-tucson	El Rustico	https://s3-media1.fl.yelpcdn.com/bphoto/IWPIqw...
2	Nggy_QUDxaLlrcQAQf7GnQ		taqueria-juanitos-tucson-4	Taqueria Juanito's	https://s3-media1.fl.yelpcdn.com/bphoto/9dwZ5q...
3	A-5IN85MwL9F8wJRSDna6g		tacos-apson-tucson	Tacos Apson	https://s3-media2.fl.yelpcdn.com/bphoto/3fm1Nq...
4	nEaTbGFIU7d9eLU2kl6KBw		taqueria-el-pueblito-tucson-2	Taqueria El Pueblito	https://s3-media2.fl.yelpcdn.com/bphoto/DNINjV...

5 rows x 24 columns



```
# Select only necessary columns
# ... = ....[['...', '...', ...]]
taco_shops_df = taco_shops_df[['id', 'alias', 'name', 'review_count', 'rating']]
taco_shops_df.shape # Check shape. Is it 20 x 5?
```

(20, 5)

▼ Getting reviews for the taco shops - [6 points]

Now we're going to use the `reviews_query()` function to get the last three reviews for a given ID. The issue here is that you can only feed it one ID at a time. So, we'll have to write a loop that queries for each ID in `taco_shops_df` and builds out a dataframe of reviews.

Task: Write a for loop that does the following steps:

- First make an empty data frame outside of the loop called `taco_shops_reviews_df`
- Initialize your loop so that it runs the length of `taco_shops_df` you made earlier.
- For each `i` in loop, use the `id` from `taco_shops_df` to get reviews from yelp using the `reviews_query()` function in `yelp_api` and store it to an object called `reviews`.
- Flatten `reviews` to an object called `reviews_df`
- Add `location_id` to `reviews_df` - I gave you the code to do this :)
- Append `reviews_df` to `taco_shops_reviews_df` such that it builds out that dataframe with each `reviews_df` dataframe that's generated each loop.
- After the loop is done select only the columns `'id', 'text', 'rating', 'time_created', 'location_id'`

```
# Write our loop
taco_shop_reviews_df = pd.DataFrame()

for i in range(len(taco_shops_df)):
    reviews = yelp_api.reviews_query(taco_shops_df['id'][i])
    reviews_df = pd.json_normalize(reviews['reviews'])
    reviews_df['location_id'] = taco_shops_df['id'][i]
    taco_shop_reviews_df = taco_shop_reviews_df.append(reviews_df)

# Select columns 'id', 'text', 'rating', 'time_created', 'location_id'
taco_shop_reviews_df = taco_shop_reviews_df[['id', 'text', 'rating', 'time_created', 'location_id']]
taco_shop_reviews_df.head()
```

	id	text	rating	time_created	location
0	OddPiUmShngB2Rj8JWXwFA	I went here on a trip and great tacos for ever... Friendly staff	5	2022-09-06 21:18:24	jmwasbZfgj3honf79qK

► Aggregating your review data - [3 points]

Task: Now go and do a data aggregation to get the mean review score across the three reviews. Remember, we want this grouped by `location_id`. Call this dataframe `latest_reviews_agg`.

In your groupby you should set `as_index` to false to make joining on the ID values easier.

You should also rename the second column to `rating_mean` vs. leaving it as the dual level name.

[] ↪ 3 cells hidden

▼ Join your two datasets and one last transform - [3 points]

Task: Now it's time to join `latest_reviews_agg` back to your `taco_shops_df` dataframe. You're going to want to join them on the `location_id`, but remember it is called just 'id' in the `taco_shops_df` dataframe. Call this joined dataset `taco_shops_comp`

After you make that dataset do one last transform. In this I want you to make a new column called 'still_good' where the value is 'yes' if the `mean_rating` is greater than or equal to the average rating since they opened, or 'no' if the rating has dropped. The idea here is that this could be a value that one would use to see if their average score of the latest reviews has improved or suffered during Covid.

```
# Join and name taco_shops_comp
taco_shops_comp = latest_reviews_agg.merge(taco_shops_df, left_on='location_id', right
taco_shops_comp.head()
```

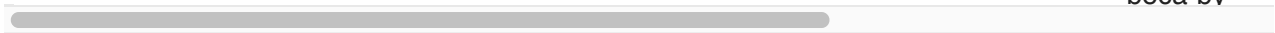
	location_id	mean_rating	id	alias
0	0Q_A1mtkXkovpuqgyT1BtA	5.000000	0Q_A1mtkXkovpuqgyT1BtA	taqueria-rosita-tucson
1	0ghzROkZWKWwQKDt1fkPAQ	4.666667	0ghzROkZWKWwQKDt1fkPAQ	el-guero-canelo-tucson-7
				boca-by-

```
# Make still_good column
taco_shops_comp.loc[taco_shops_comp['mean_rating'] >= taco_shops_comp['rating'], 'stil
taco_shops_comp.loc[taco_shops_comp['mean_rating'] < taco_shops_comp['rating'], 'still
taco_shops_comp.head()
```

```
# So, do any locations have a lower average in their last three reviews compared to the  
lower_ratings = taco_shops_comp[(taco_shops_comp['still_good'] == 'no')]  
print(f"There are {len(lower_ratings)} taco shops with lower ratings.")
```

There are 9 taco shops with lower ratings.

1	0ghzROkZWKWwQKDt1fkPAQ	4.666667	0ghzROkZWKWwQKDt1fkPAQ	canelo- tucson-7
				boca-by-



[Colab paid products](#) - [Cancel contracts here](#)

✓ 0s completed at 7:16 AM

