Table of Contents

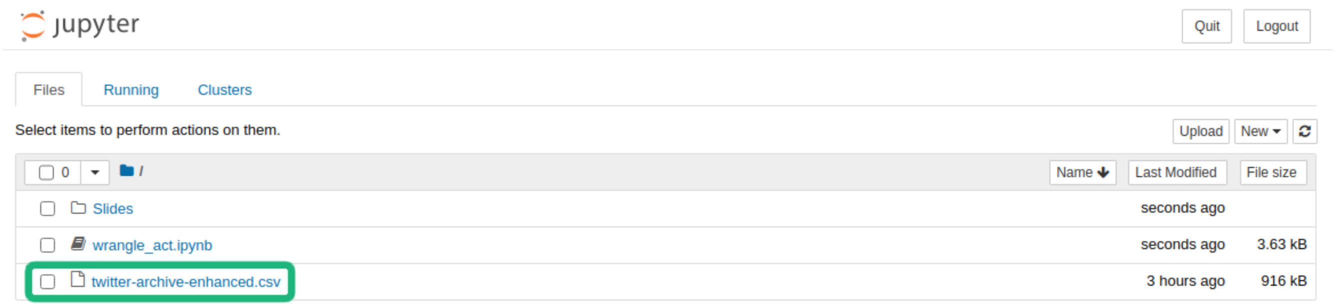
Gathering

This project requires gathering data from 3 different sources each requires a different technique of acquiring the data. This is very convenient as it enabled me to try all three techniques we learned in the nanodegree.

First source

The first data source was the file **“twitter-archive-enhanced.csv”** which is available to be downloaded directly from the project details page.

Needless to say, this was the easiest source to acquire the data from. All I needed to do was to download the file into the project folder. I used pandas read_csv to read the downloaded file into a pandas DataFrame and named it **archive_tweet_df.**



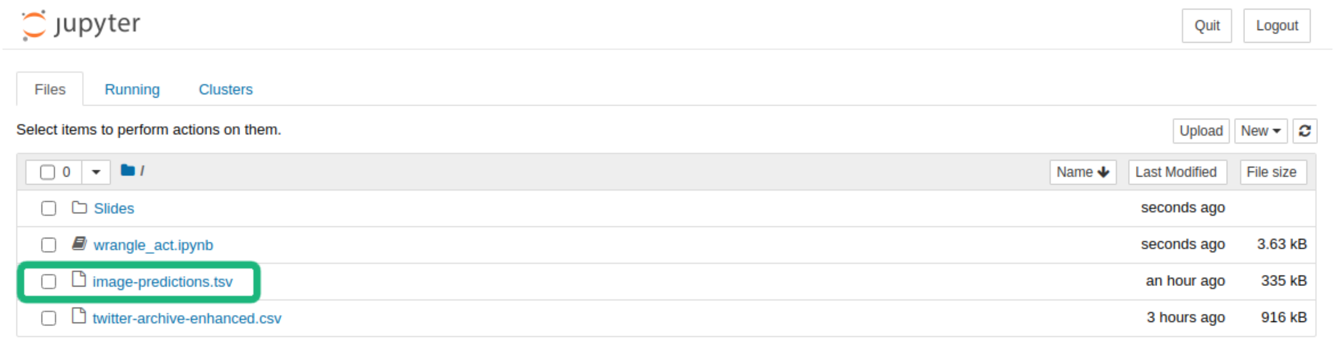
Select items to perform actions on them.

Name	Last Modified	File size
Slides	seconds ago	
wrangle_act.ipynb	seconds ago	3.63 kB
twitter-archive-enhanced.csv	3 hours ago	916 kB

Second source

The second data source was slightly trickier to acquire, the file “image_predictions.tsv” is hosted on Udacity’s servers. I downloaded it programmatically from within the jupyter notebook.

I used the **requests** python module to download the file from the URL pointing to it. Being run from within my jupyter notebook, the code to download the file saved it directly to the project folder. I used pandas’ **read_csv** to read the downloaded file into a pandas DataFrame and named it **img_pred_df.**



Select items to perform actions on them.

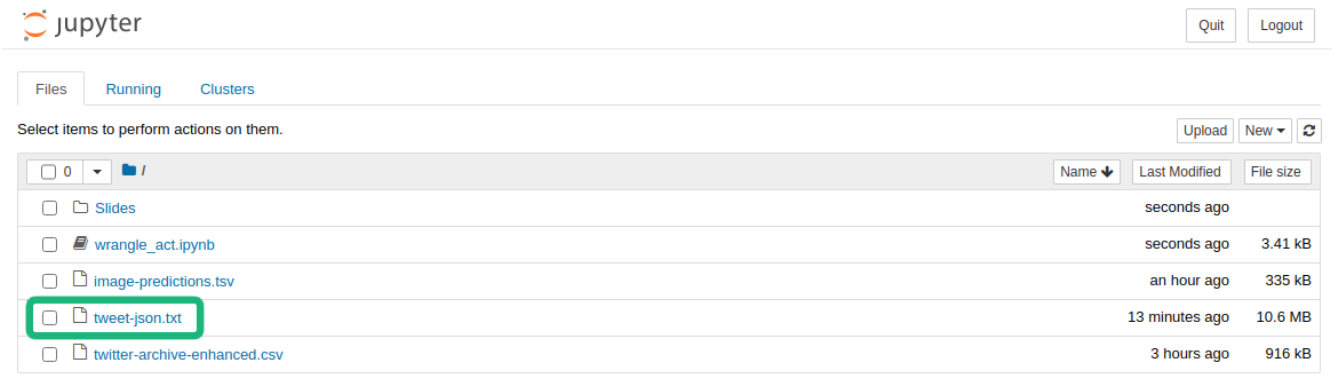
Name	Last Modified	File size
Slides	seconds ago	
wrangle_act.ipynb	seconds ago	3.63 kB
image-predictions.tsv	an hour ago	335 kB
twitter-archive-enhanced.csv	3 hours ago	916 kB

Third source

The third data source required the most complex approach. First I had to sign up for a Twitter developer account. After my application was approved, I had to make sure my API key will be hidden when I publish the project to my GitHub or send it to Udacity for review.

I dealt with this using the environment variables approach. I created a .env file and saved my API key in it. I used a python module called “python-dotenv” to load the API key from the aforementioned .env file.

I used the tweets’ IDs from the first data source to hit the Twitter API through **tweepy** module. For each tweet that still exists, I saved its **JSON** string representation to the "tweet-json.txt" file on a new line. I loaded the data from “tweet-json.txt” into a pandas DataFrame called **current_tweet_df** using pandas’ read_json.



Select items to perform actions on them.

Slides	seconds ago	
wrangle_act.ipynb	seconds ago	3.41 kB
image-predictions.tsv	an hour ago	335 kB
tweet-json.txt	13 minutes ago	10.6 MB
twitter-archive-enhanced.csv	3 hours ago	916 kB

Assessing

I used both programmatic and visual methods to assess the gathered data. For programmatic assessment, the functions info and describe provided an overview of the values, data types and some extra statistics. For visual assessment, head, tail and sample functions came in handy.

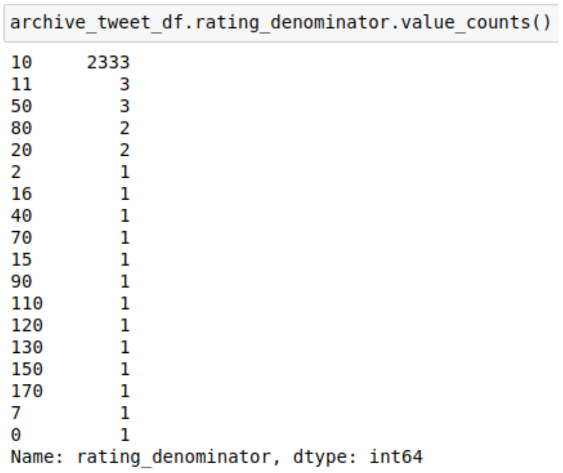
After the assessment, I found that **archive_tweet_df** and **current_tweet_df** have more issues than **img_pred_df.** Here is a list of all the issues I found:

Quality Issues

archive_tweet_df

1. Missing values

2. Invalid values in rating_denominator



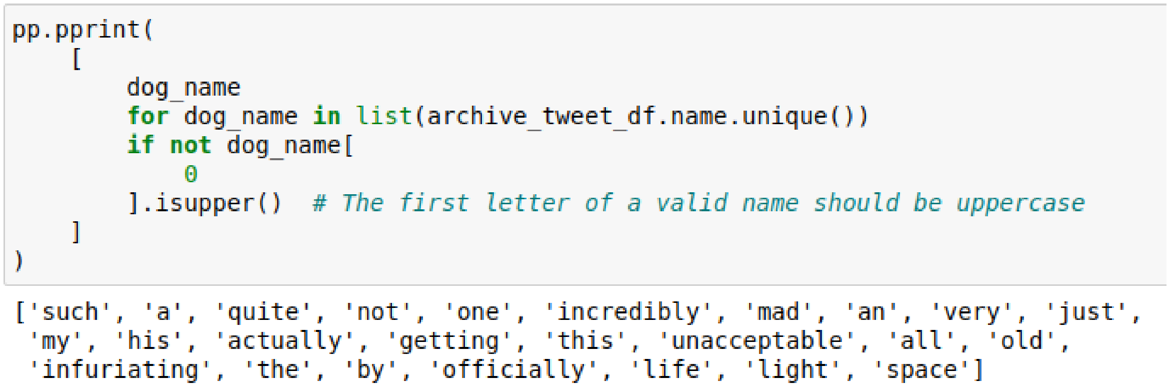
```
archive_tweet_df.rating_denominator.value_counts()
```

```
10     2333
11        3
50        3
80        2
20        2
2         1
16        1
40        1
70        1
15        1
90        1
110       1
120       1
130       1
150       1
170       1
7         1
0         1
Name: rating_denominator, dtype: int64
```

3. Erroneous data type for the timestamp column

4. "None" is used to refer to missing information in the name column

5. Inaccurately parsed dog names in name such as "a", "an", "my" and "by"



```
pp.pprint(
    [
        dog_name
        for dog_name in list(archive_tweet_df.name.unique())
        if not dog_name[
            0
        ].isupper()  # The first letter of a valid name should be uppercase
    ]
)
```

```
['such', 'a', 'quite', 'not', 'one', 'incredibly', 'mad', 'an', 'very', 'just',
 'my', 'his', 'actually', 'getting', 'this', 'unacceptable', 'all', 'old',
 'infuriating', 'the', 'by', 'officially', 'life', 'light', 'space']
```

img_pred_df

6. Records of images that the image prediction algorithm could not identify a dog in

current_tweet_df

7. Missing values

8. The table is cluttered with irrelevant information, id and id_str being exact copies of each other for instance. id, retweet_count and favorite_count columns are the only relevant columns in this table.

Quality issues not related to single table

9. Inconsistent number of records among the three tables:

archive_tweet_df: 2356 entries

img_pred_df: 2075 entries

current_tweet_df: 2331 entries

<u>Tidiness Issues</u>

archive_tweet_df

1. The dog age information is distributed over 4 columns (doggo, pupper, floofer and puppo), this violates the first tidy data requirement "Each variable forms a column"

img_pred_df

2. We have 6 columns (p1, p1_conf, p1_dog, ...etc.) that provide one piece of information which is the **dog breed**. They should all be reduced to one column so we do not violate the first tidy data requirement "Each variable forms a column".

Tidiness issues not related to a single table

3. Having tweets' data scattered over the two tables **archive_tweet_df** and **current_tweet_df** violates the third tidy data requirement "Each observational unit forms a table"

Cleaning

Before making any changes to the data, I made a copy of it using pandas' copy function that makes a deep copy by default. A deep copy means that whatever changes are made to it don't affect the gathered data.

```
# Fortunately, deep is set to True by default
archive_tweet_clean = archive_tweet_df.copy()
img_pred_clean = img_pred_df.copy()
current_tweet_clean = current_tweet_df.copy()
```

I followed the define-code-test steps to clean every issue I found in the assessing step. While doing so, I also dropped columns that will not be needed anymore, they were mostly tweets' metadata in **archive_tweet_df** and **current_tweet_df.** Clearing the clutter helped me see other issues that I had missed in my first iteration of the assessment. I went back and added any extra issues I found to my assessment findings.

All my cleaning efforts are in the Jupyter notebook, it would be an unnecessary duplication to just repeat everything here so I am going to select key cleaning steps to talk about.

Rating Denominators and Numerators

It is understandable that the rating numerators were greater than the denominators and that this unique rating system is part of the popularity of WeRateDogs. However, as data analysts, we need to represent data in a concise way that facilitates our end goal which is extracting insights and drawing conclusions.

To my surprise, I found that rating denominators are not always 10 which means that there is not a unified schema that this rating system conforms to. So I decided to reduce these two columns to one column and named it rating, because this what these two variables actually represent. I calculated the values in the rating column by dividing the values in the rating numerator by 10. Afterwards, I dropped the rating numerator and denominator columns.

"doggo", "pupper", "floofer" and "puppo" columns

This one was easy to catch in the assessment but not as easy to clean. I reduced all four columns to one column called "age". For every record that had "None" in all four columns, I added a NaN in the "age" column. For every other record that had a value in one of the four columns, I added the value to the "age" column. I am not very proud of the way I did it, it involved a list comprehension and some tricks that I feel should not be needed when using pandas but I am glad it worked.

I also made sure this column is treated as a categorical one using pandas' astype function. Afterwards, I dropped all four column as they do not provide any information now.

Inconsistent number of records among the three tables

At first, I thought this was going to be a hard one to deal with. But I found a very convenient function called pandas.Index.intersection. I first set the index of all three tables to “tweet_id” as it is the unique id that existed in all three tables and then used the aforementioned intersection function to find the common tweets. Afterwards, it required only one line of code per table to use the common index to get the common records.