wrangle_report

June 28, 2022

1 Project: Wrangling WeRateDogs Twitter Data

1.1 Introduction

In this project, I will be working with data from a Twitter account known as @WeRateDogs. This data will be gathered in three ways. For the first dataset I will use a downloaded archive of their 2017 tweets that the account owners downloaded and sent to Udacity in an email, the second dataset is an image prediction file that Udacity provided as part of the project, and the third dataset will be acquired through the Twitter API.

The objective of this project is to demonistrate the steps taken in order to gather the data from it's sources, asses it, clean it, store it, analyze it and draw conclussions, and finally create some cool visualizations.

1.2 Data Gathering

Datasets to gather: 1. A .csv file from their archive (twitter-archive-enhanced.csv) 2. An image prediction model as a .tsv file (image-predictions.tsv) 3. Tweets from the Twitter API that correspond to the ids in the archive file

1.2.1 Python Libraries Used

- json - used for reading json files
- requests - used for downloading the image prediction file from the internet
- os - used to access files on my disk and also to create folders and files
- numpy as np - used to access data as arrays (assigned to an alias np)
- pandas as pd - used for reading files into dataframes and manupilating them (assigned to an alias pd)
- matplotlib.pyplot as plt - used to create visualizations (assigned to an alias plt)
- seaborn as sns - used to create visualizations (assigned to an alias sns)
- tweepy - used to query data through the Twitter API
- default_timer as timer - used to calculate code running time

Import Libraries

```
import json
import json
import requests
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tweepy
from timeit import default_timer as timer
```

1.2.2 Read the archive file from local storage

I will use the pd.read_csv function to read the file from the local storage and then assign the resulting dataframe to a varible called **archive**.

```
archive = pd.read_csv('twitter-archive-enhanced.csv')
```

1.2.3 Use the Requests library to download the image prediction file

I write a generic function that accepts the file url and a folder name as parameters, that: * Uses the os.path.exists function to check if a folder with a given name exists on the machine * If a folder with the given name exists, it skips that operation * If a folder with the given name doesn't exist, it creates one with the given name and proceeds to: * Use the request library to acquire the response from a given url * Uses the with open function to open the given folder * Splits the given file url string at the occurence of the '/' symbol to create a list of elements * Uses the last element in the list as the file name (image-predictions.tsv) * Creates the file without making any changes ('wb') in the given folder * Displays an error message from the requests library if there is an error

I call the function passing in the url and folder name respectively as arguments.

I use pd.read_csv (passing a t̂o indicate a tab separator) to read the file into a dataframe called **predictions**.

Deliverable: All operations ran succesfully and the file was downloaded. Additionally, there was no error raised at the end of this request

```
def read_from_url(url, folder_name):
    """A function that creates a folder on disk then downloads
    data from the internet using the requests library and writes it to a file it creates
    by splitting the url at the '/' symbol to create a list of elements in the url string
    and grabs the last element in the list to use as the file name.
    a creates"""
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
        try:
            response = requests.get(url)
            with open(os.path.join(folder_name, url.split('/')[-1]), mode='wb') as url_file:
                url_file.write(response.content)
        except requests.exceptions.RequestException as e:
            raise SystemExit(e)
# save file
read_from_url('https://d17h27t6h515a5.cloudfront.net/topher/2017/August/599fd2ad_image-predicti
```

```
predictions = read_csv('dog_ratings_twitter/image-predictions.tsv','\t')
```

1.2.4 Use the Tweepy library to query data via the Twitter API

- I sign up for an account at the Twitter Developers platform
- I submit an applicatin to use the Twitter API in my app
- My application gets approved and I receive tokens to access the API! YAY!!

I write a Python class called Tweepy that: * Declares and assigns (from the disk using os.environ.get) the four tokens acquired as class attributes assigned every time the class is called (This is sufficient to use with my one Twitter account) * Defines a class method called **authenticate that: * Uses the tweepy library to authenticate my API request * Sets the wait_on_rate_limit and wait_on_rate_limit_notify to True in order to wait when the limit of 15 minutes is reached and to notify when waiting on rate limit. * Returns an authenticated tweepy.api.API object that I will use to retrieve data using tweet ids from the archive file

Tweepy class

```
class Tweepy:
    '''
    A class that takes in four api tokens from the Twitter API.
    Defines a class method that uses the tweepy library to
    create a tweepy.OAuthHandler object by passing
    in the consumer key and secret declared in the class.
    Use the created object to set the access tokens.
    Call the tweepy.API function by passing in the authenticated object
    and setting the wait_on_rate_limit and wait_on_rate_limit_notify to True
    in order to wait the limit of 15 minutes is reached and to notify when waiting on rate limi
    Return an authenticated tweepy.api.API object used to retrieve data.
    '''
    consumer_key = os.environ.get('CONSUMER_KEY')
    consumer_secret = os.environ.get('CONSUMER_SECRET')
    access_token = os.environ.get('ACCESS_TOKEN')
    access_secret = os.environ.get('ACCESS_SECRET')

    def authenticate(self):
        auth = tweepy.OAuthHandler(self.consumer_key, self.consumer_secret)
        auth.set_access_token(self.access_token, self.access_secret)
        api = tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_notify=True)
        return api
```

Retrieve Data From the Tweepy Object Created Created a function that holds the logic to: * Create an authenticated api instance called **api** from the Tweepy object created above. * Retrieve the tweet_id values from the archive dataframe using the numpy.values method, assing to **tweet_ids**. * Declare a **count** variable with the value of 0 to count the number of tweets retrieved. * Create an empty dictionary that will hold all the failed tweet_ids alongside the erro message. * Declare a timer object and assign it to the **start** variable. * Use the with context's open function to open a file (aliased it as **dump_file**). * Start a for loop inside the open function that does the following: * Loops through all **tweet_ids** in the archive dataframe assigning every iteration to the variable **tweet_id**. * Increments the **count** by 1 after every iteration. * Prints out the **count** concatenated with the current **tweet_id** as strings. * Opens a try clause to check for errors while retrieving data from Twitter: * Call the get_status method from tweepy on the **api** object we created by passing in the **tweet_id** and setting mode to extended in order to retrieve the entire untruncated text of the Tweet; then and assing it to a variable called **tweet**. * Print 'Success' if a **tweet** was retrieved successfully. * Convert the **tweet** as json dump. * Write to the **dump_file** created earlier adding a line after each **tweet** object. * Opens an except clause that raises an exception from tweepy. * Print 'failed' whenever there's an error. * Add the error into the **fails_dict** created earlier. * Create another timer instance called **end**. * Print the result of **end-start** (The time used to retrieve the tweets). * Print out the values from the **fails_dict**.

Call the get_tweets function to start the process.

Deliverables: * A tweet_json_txt file with all the tweets downloaded. (2356 tweets) * The number of minutes it took to retrieve the tweets. (about 10 minutes) * A print out of all the failed tweets. (31 errors)

Function to get tweets

```
def get_tweets():
    tweet_ids = archive.tweet_id.values
    api = Tweepy().authenticate()
    count = 0
    fails_dict = {}
    start = timer()

    with open('tweet_json.txt', 'w') as dump_file:
        for tweet_id in tweet_ids:
            count +=1
            print(str(count) + '+'+ str(tweet_id))

            try:
                tweet = api.get_status(tweet_id, tweet_mode='extended')
                print('Success')
                json.dump(tweet._json, dump_file)
                dump_file.write('\n')
            except tweepy.TweepError as err:
                print('failed')
                fails_dict[tweet_id] = err
                pass

    end = timer()
    print(end - start)
    print(len(fails_dict))
    print(fails_dict)
get_tweets()
```

Read Data From tweet_json.txt I used the pd.read_json function to read all the data into a dataframe in order to display all the columns in the dataframe and choose the ones to use in the analysis. There are 32 columns in the dataset but I will select a handful of them since the rest are in the archive dataframe.

I write a function called **json__to__df()** to read the data by directly selecting the columns I'm interested in. The function: * Declares and initializes an empty list called **tweet__list**. * Opens a try clause and uses a context to open the txt file created in the previous step. * Passes the file name and a 'utf-8' encoding into the context and assigns the file an alias of **f**. * Uses a list comprehension to loop through all lines inside **f** while parsing and converting them to a pythn dictionary using **json.loads. * Opens a for loop that loops through the list of dictionaries previously created by list comprehension. * Selects 5 columns of interest. * Names them ensuring that columns common in both the archive and the api tables have the same name to allow for smooth merging later. * Appends the column names as keys and the iterating variable as values to the** tweet_list** list to create a list of dictionaries. * Raises a json.JSONDecodeError in case of an error. * Uses pd.DataFrame passing in **tweet__list** as the list and assigning it to **tweet__df**. * Returns the pandas dataframe.

I call the **json__to__df** function on the variable **api__tweets** which will be used during the assessment phase.

Deliverables: * A pandas dataframe called api_tweets** * No errors thrown.

Function to Extract and Convert Data into a Pandas Dataframe

```
def json_to_df():
    tweet_list = []

    try:
        with open('tweet_json.txt', encoding='utf-8') as f:
            dicts_lst = [json.loads(line) for line in f]

            for dict_item in dicts_lst:
                tweet_list.append({'tweet_id': dict_item['id'],
                                   'lang': dict_item['lang'],
                                   'retweet_count': int(dict_item['retweet_count']),
                                   'favorite_count' : int(dict_item['favorite_count']),
                                   'timestamp': dict_item['created_at']})
    except json.JSONDecodeError as error:
        raise error
    tweet_df = pd.DataFrame(tweet_list)
    return  tweet_df
api_tweets = json_to_df()
```

1.3 Assessing Data

1.3.1 Archive Dataframe

The WeRateDogs Twitter archive contains basic tweet data for all 5000+ of their tweets by August 1, 2017, but not everything.

archive table feature description

- **tweet__id**: The unique identifier of the tweet
- **in__reply__to__status__id**: The identifier for a reply to a status
- **in__reply__to__user__id**: The identifier for reply to a user

- **timestamp**: The date and time the tweet was made
- **source**: The device the tweet was posted from
- **text**: The text of the tweet which includes the dog name and dog stage
- **retweeted_status_id**: The identifier of a retweet to a status
- **retweeted_status_user_id**: The identifier of a retweet to a user
- **retweeted_status_timestamp**: The date and time of the retweet
- **expanded_urls**: The link to the tweet (they have the tweet_id at the end)
- **rating_numerator**: The rating given to the dog
- **rating_denominator**: The denominator used for the rating the dogs
- **name**: The name of the dog
- **doggo**: A big pupper, usually older (according to the Dogtionary)
- **floofer**: Any dog really (according to the Dogtionary)
- **pupper**: A small doggo, usually younger (according to the Dogtionary)
- **puppo**: A transistional phase between pupper and doggo (according to the Dogtionary)

Assesing steps

- list all the columns in the table: `archive.columns`
- display more than five rows from th table: `archive`
- query a row that has data from beyond 2017: `archive.query('tweet_id == 666049248165822465')`
- filter a portion of the dataset that are retweets: `archive[~archive.retweeted_status_id.isnull()]`
- display the sum of duplicates in the expanded_urls column: `archive.expanded_urls.duplicated().sum()`
- more assessing of the expanded urls: `archive.expanded_urls.value_counts()`
- view information about the features i.e datatypes and number of rows: `archive.info`
- check the sum of null values: `archive.isnull()`
- more assesing: `archive.sample(40)`
- show the statistical summary of the rating numerator: `archive.rating_numerator.describe()`
- show the statistical summary of the rating denominator: `archive.rating_denominator.describe()`

1.3.2 Predictions Dataframe

A table full of image predictions (the top three only) alongside each tweet ID, image URL, and the image number that corresponded to the most confident prediction (numbered 1 to 4 since tweets can have up to four images).

Image predictions table feature description

- **tweet_id**: The unique identifier of the tweet
- **jpg_url**: The url to the dog's image
- **img_num**: The number of images used for the prediction
- **p1**: The algorithm's #1 prediction for the image in the tweet
- **p1_conf**: How confident the algorithm is in its #1 prediction
- **p1_dog**: Whether or not the #1 prediction is a breed of dog
- **p2**: The algorithm's second most likely prediction
- **p2_conf**: How confident the algorithm is in its #2 prediction

- **p2_dog** Whether or not the #2 prediction is a breed of dog
- **p3**: The algorithm's second most likely prediction
- **p3_conf**: How confident the algorithm is in its #2 prediction
- **p3_dog** Whether or not the #2 prediction is a breed of dog

Assesing steps

- list all the columns in the table: `predictions`
- view information about the features i.e datatypes and number of rows: `predictions.info`
- more assesing: `predictions.sample(10)`
- show the statistical summary of numerical values: `predictions.describe()`*
- display the sum of duplicates in the table: `predictions.duplicated().sum()`
- check the sum of null values: `predictions.isnull()`

1.3.3 Api Tweets Dataframe

Data queried from the Twitter API tha contains additional data corresponding to the data in the archive table; **retweet_count** and **favorite_count**.

api tweets table

- **tweet_id**: The unique identifier of the tweet
- **timestamp**: The date the tweet was posted
- **lang**: The language used in the tweet
- **retweet_count**: If it's been retweeted, how many retweets?
- **favorite_count**: How many likes does the tweet have?
- **full_text**: The untruncated text of the tweet

Assesing steps

- list all the columns in the table: `api_tweets`
- view information about the features i.e datatypes and number of rows: `api_tweets.info`
- more assesing: `api_tweets.sample(10)`
- show the statistical summary of numerical values: `nums_api_tweets.describe()`*
- display the sum of duplicates in the table: `api_tweets.duplicated().sum()`
- check the sum of null values: `api_tweets.isnull()`

1.3.4 Quality Issues

archive table

- Object datatype for timestamp instead of datetime datatype example: tweet_id 700847567345688576
- Some tweets are retweets
- Erroneous integer datatype for tweet_id
- Missing dog names misrepresented as None
- Missing dog stages misrepresented as None
- Non-logical names for dog in the name column, i.e quite, such, not
- Some rating_denominators are not equal to 10

image predictions table

- Inconsistent capitalization of names in p1, p2, and p3
- Erroneous integer datatype for tweet_id

api_tweets table

- Erroneous datatype integer for tweet_id

1.3.5 Tidiness issues

- Some of the expanded_urls rows in the archive table contain more than one url
- The in_reply_to_status_id, in_reply_to_user_id, retweeted_status_id, retweeted_status_user_id, retweeted_status_timestamp columns are not needed for this analysis
- The dog stages are spread across four columns; doggo, floofer, pupper, puppo
- There are tweets dated before 2017 i.e 2015, 2016. (the image prediction file only covers 2017)

1.4 Cleaning Data

Make copies of the datasets I made copies of the original datasets using the pd.copy function.

```
tweets_clean = api_tweets.copy()
archive_clean = archive.copy()
predictions_clean = predictions.copy()
```

In this phase, I proceed to clean the issues found in the all the datasets each one at a time. I will use these three steps in cleaning each issue: 1. **Issue**: State the issue to be cleaned and the table it belongs to. 2. **Define**: Define the steps to be taken to clean the issue stated. 3. **Test**: Run a test code to confirm that the issue has been cleaned.

1.5 Storing Data

I will merge and save gathered, assessed, and cleaned master dataset to a CSV file named "twitter_archive_master.csv".

1.5.1 Merge the three dataframes

```
 twitter_archive_master = archive_clean.merge(tweets_clean,on='tweet_id').merge(predictions_cl
```

1.5.2 Export the dataframe to a csv file using pd.to_csv()

```
twitter_archive_master.to_csv('twitter_archive_master.csv', index=False)
```

1.6 Regards

Thank you!