Lab Assignment 2

Data Engineering 1

For this assignment, answer the following questions using a Jupyter Notebook, then output your asignment to HTML then to PDF. Submit the PDF to Gradescope on Canvas, as described in these instructions. When you submit your assignment to Gradescope, make sure to mark which pages on the PDF each question appears on.

You will need the following packages:

In [181…
```
import numpy as np
import pandas as pd
import requests
import json
import dotenv
import os
from bs4 import BeautifulSoup
```

For the following problems, you will need the following headers. Feel free to change the botname and version if you like.

In [182…
```
botname = 'ds6600'
version = '0.0'
useragent = f'{botname}/{version} python-requests/{requests.__version__}'
headers = {'User-Agent':useragent}
headers
```

Out[182…
```
{'User-Agent': 'ds6600/0.0 python-requests/2.32.5'}
```

Problem 1

CSV is such a straightforward, universal data storage format that we can be fooled into thinking that a CSV file will be easy to load into Python. In reality, many issues can arise with a CSV that can prevent us from loading the data without error or load the data in an incomplete or corrupted way. For an example, look at the bulk data download from Open Secrets.

Start by downloading the cands22.txt data file from Canvas (this is the unaltered candidates data from the 2022 elections from Open Secrets), or if you want to work with the whole data set yourself, register for a bulk data account here, then download the ZIP directory and find the cands22.txt file.

You might notice that trying to load the cands22.txt file via `pd.read_csv()` results in a "ParserError: Error tokenizing data". The Open Secrets Data Guide, page 5, says:

> The major Open Data tables are provided in a non-standard format that allows dirty data to be imported as we are provided some raw data fields that can contain formatting and other unprintable characters that choke many data systems. This format requires a more advanced level of skill to import than a conventional CSV file. In this bulk data, text fields are surrounded by the pipe character (ascii 124). Date and numeric fields are not. Commas separate all fields.

The pipe character is the vertical line that appears by pressing shift and the key immediately above enter/return.

Part a

Use `pandas` to load the data properly as a dataframe into your Python kernel. Make sure that the resulting data frame

- Has 8928 rows and 12 columns
- Does not use data as the column names. If you load the data properly, the columns will be denoted simply as 0, 1, 2, ..., 11. (If you want, you can replace the column names with `['Cycle', 'FECCandID', 'CID','FirstLastP','Party','DistIDRunFor','DistIDCurr','CurrCand','`
- Does not have any | pipes

For this problem, it will be helpful to look at the parameters of the `pd.read_csv()` method listed in the docstring: `help(pd.read_csv)` . [4 points]

Part b

Suppose we used the following code to load the candidates data:

```
cand = pd.read_csv('cands22.txt', on_bad_lines='skip')
```

What does this approach do, and how does it specifically address the problem that yields an error when running `pd.read_csv('cands22.txt')` ? Ignore for the moment the use of data for column names, and ignore the presence of the pipe character. Other than those problems, is this an appropriate way to load the data? Why or why not? [Hint: it might be helpful to run `cand = pd.read_csv('cands22.txt')` and think carefully about the error, and it may also help to click on the "cands22.txt" file to view the raw CSV data.] [4 points]

Problem 2

The National Basketball Association has saved data on all 30 teams' shooting statistics for the 2014-2015 season here: https://stats.nba.com/js/data/sportvu/2015/shootingTeamData.json. Take a moment and look at this JSON file in your web browser, or more neatly visualized here: https://jsonhero.io/j/CwKnNM18suZp. In this problem our goal is to use `pd.json_normalize()` to get the team-by-team data into a pandas dataframe. The following questions will guide you towards this goal.

Part a

Download the raw text of the NBA JSON file using `requests.get()`, and provide your user-agent to the NBA API by setting the `header` argument to the `headers` dictionary you created above. Then use `json.loads()` (or the `.json()` method applied to the `requests` output) to parse the dictionaries and lists in the JSON formatted data. [2 points]

Part b

Based on your observations of the JSON structure, describe in words the path that leads to the team-by-team data. [2 points]

Part c

Use the `pd.json_normalize()` method to pull the team-by-team data into a dataframe.

[Note: what makes this tricky is that one of the layers in the path to the data is named only `0`. This `0` is not a string like the other keys. Specifying `0` in the `record_path`, either with or without quotes yields an error. There are two ways to solve this. The easiest way is to just skip over the `0` key entirely when writing down the `record_path`, and `pd.json_normalize()` will include the `0` layer on its own as a default behavior. The other way is to index the JSON data with both `['resultSets'][0]` first before passing the data to `pd.json_normalize()`, then writing the remaining layers in `record_path`.]

If you are successful, you will have a dataframe with 30 rows and 33 columns. The first row will refer to the Golden State Warriors, the second row will refer to the San Antonio Spurs, and the third row will refer to the Cleveland Cavaliers. The columns will only be named 0, 1, 2, ... at this point. [2 points]

Part d

Find the path that leads to the headers (the column names), and extract these names as a list. Then set the `.columns` attribute of the dataframe you created in part c equal to this list. The result should be that the dataframe now has the correct column names.

[Note: In this case, there's no need for `pd.json_normalize()` as the headers already exist as a list.]

[2 points]

Part e

Save the NBA dataframe you extracted in problem 4 as a JSON-formatted text file on your local machine. Format the JSON so that it is organized as dictionary with three lists: `columns` lists the column names, `index` lists the row names, and `data` is a list-of-lists of data points, one list for each row. [Hint: this is possible with `.to_json()` method applied to a pandas dataframe.] [2 points]

Problem 3

Pull data in JSON format from Reddit's top 25 posts on /r/popular. Start by using `requests.get()` with the `headers=headers` parameter on the website: http://www.reddit.com/r/popular/top.json. Then use `json.loads()` to parse the JSON format as python dictionaries and lists. If you look at the result, you should see JSON data.

(If instead you see an access denied message, wait a couple minutes, restart your notebook and try again. If you still see an error, change your user agent and try again. You may have to be patient. Reddit is notoriously tight with its data, but it should work eventually if you try these two steps.)

If you were to use `pd.json_normalize()` at this point, you would pull all of the features in the data into one dataframe, resulting in a dataframe with 172 columns.

If we only wanted a few features, then looping across elements of the JSON list itself and extracting only the data we want may be a more efficient approach.

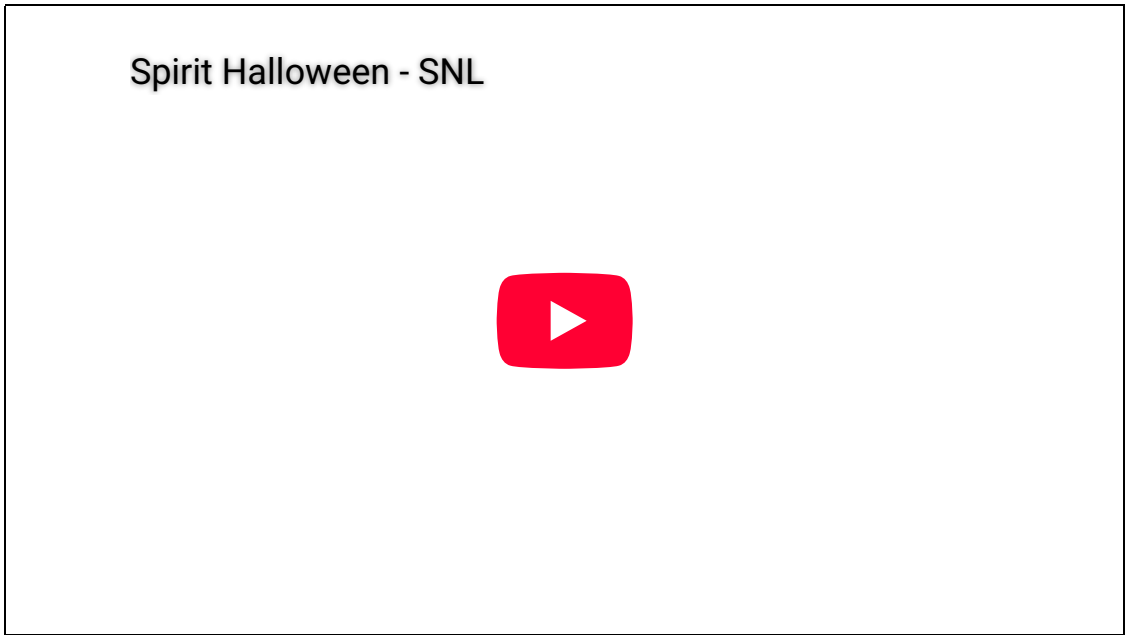
Use looping - and not `pd.read_json()` or `pd.json_normalize()` - to create a dataframe with 25 rows (one for each of the top 25 posts), and only columns for `subreddit`, `title`, `ups`, and `created_utc`. You can follow the example listed in section 3.3.3 of Surfing the Data Pipeline with Python as a way to start. (Your result might be a dataframe without any column headers. You can set the `.columns` attribute of the dataframe to the list `['subreddit', 'title', 'ups', 'created_utc']`.) [6 points]

Problem 4

Finding the hidden API:

APIs are the primary mechanism for transferring data over the internet, but most APIs are for internal use for a website and are not intended for outside users. When this is the case, there won't be an obvious link to the API and there won't be any documentation. You can still sometimes get access to the API. This exercise will guide you through one instance of finding and using a hidden API.

First, watch this video that appeared on Saturday Night Live on September 28, 2024:



It's the time of year for Spirit Halloween stores to pop up. They often take over the stores where large chains have recently met their demise. Go to the store locator page: https://stores.spirithalloween.com/ Notice that the top hit says "Former Roses". I want to know how often different former businesses appear in the descriptions of these stores. If you right click on this page and view source, the data that appears about the nearby stores does not appear. Instead this website is calling a hidden API that we can access.

For this problem, use the webpage inspector in the Google Chrome web browser. If you don't have Chrome, download it here: https://www.google.com/chrome/

Step 1: With Chrome, go to https://stores.spirithalloween.com/ Right click on this page, and select "Inspect".

Step 2: Inspect is a complicated but extremely useful tool. It reports all APIs and other web-based connections that a website makes. Click on Network to see these connections. These are all the calls the Spirit Halloween website makes to various APIs to display images, maps, ads, and addresses.

Step 3: In the right-hand window within the Inspect tool, click on Response. Our task is to sift through the various API calls and to look at the responses until we see the specific address information we need. This took me a long, long time, and I'll save you that trouble -- find the entry named getAsyncLocations?template=domain&level=domain,

click it, and look at the JSON that appears under Response. (If you don't see this entry, reload the page and look again.) Under "markers" you will see a list, and the first item in the list is JSON data that contains a key named "info". Look at the value attached to "info", and scroll to the right until you see the phrase "Former Big Lots".

Step 4: Now that we know the API call that Spirit Halloween used to get the addresses, we can deduce the root, endpoint, and some of the parameters. Hover your mouse over the spot that reads getAsyncLocations?template=domain&level=domain. It displays: https://maps.spirithalloween.com/api/getAsyncLocations?template=domain&level=domain. So the root should be https://maps.spirithalloween.com, the endpoint should be /api/getAsyncLocations, and two of the parameters should be template and level, both set equal to "domain".

Step 5: Take a closer look at Response. There are other parameters here we can use. At the bottom of the JSON output is a key named "options". I didn't know this for sure, but my bet was that the key-value pairs inside "options" can be changed as parameters in the call to the API. My goal is to get all of the Spirit Halloween stores, not just the ones near me. So some of the values I wanted to change are lat and lng, which define my location, radius which defines the distance in miles from my location, and limit, which I bet specifies the maximum number of results. I want these values to be 'lat': 40.380028, 'lng': -97.910156, which places my location in the middle of the country, 'radius': 1800 which captures the entire lower 48 states, and 'limit': 2000 which exceeds the number of Spirit Halloween stores (I had to guess-and-check that).

Part a

Issue an API call to https://maps.spirithalloween.com with endpoint /api/getAsyncLocations. Set the parameters as discussed in step 5. [2 points]

Part b

Next, we need to find a way to extract the descriptive phrases such as 'Former Sears' from each store's address and store them in a list. Let's take this step by step. First, examine the data returned by your requests call in part (a). The text is JSON, and inside this JSON is a list where each element of the list refers to one store. Extract this list and save it as a separate Python variable. [2 points]

Part c

Each element of your list is another JSON dictionary. One of the keys in this dictionary contains the location name, address, and if applicable, what the store used to be. Extract just these relevant keys, and save them in a new list. [2 points]

Part d

Your list should now contain many elements that all begin with `'<div class="tlsmap_popup">`. This code is HTML. We can extract data from HTML using `BeautifulSoup()` from the `bs4` package. Please refer to chapter 5 of Surfing the Data Pipeline with Python for more information about how to use `BeautifulSoup()`.

Write a function that works with one element of this list and accomplishes the following tasks:

1. It uses `BeautifulSoup()` on the element to enable Python to search through the HTML for particular data.
2. It uses the `.find()` method to identify `div` tags with class `tlsmap_popup`
3. The data we need exist in between the opening `<div>` and the closing `</div>` tags. Extract this data by calling the `.string` attribute on the result of `find()`.
4. The data looks like JSON data (or a Python dictionary), EXCEPT that there is a closing comma which will cause problems. Remove the closing comma by indexing the string with `[:-1]` (this tells Python to keep all characters in the string except for the last one)
5. Pass the remaining string to `json.loads()` to be able to work with it as a dictionary.
6. Extract the value of the key named `address_2`. Have your function return this value.

[6 points]

Part e

Use a list comprehension to apply your function from part (d) to all elements of the list you obtained in part (c). [2 points]

Part f

Filter this list to only the elements that contain the word "Former". You can use a list comprehension for this by typing something like

```
mylist = [x for x in mylist if 'Former' in x]
```

Then report the frequencies, which you can get by typing:

```
pd.Series(mylist).value_counts()
```

[2 points]