Danny Garfield

YipitData coding assignment

Assumptions

The assignment instructed the application to count “the number of times a certain search term is in the source content of that webpage.” I made the following interpretations/assumptions:

- “Source content” is all HTML present on a page;
- “Search term” can be any non-null set of characters;
- It is acceptable for users to run this application on their local machine;
- It is not a priority for results from the scraper to be stored in a persistent database;

Approach

To build the web scraper, I built a one-file Flask application in Python that listens and speaks to a React.js frontend built with create-react-app. I began by creating the backend functionality of requesting a web page from a URL, parsing the source content, and counting the number of times a search term appears. I leaned on the “BeautifulSoup” and “requests” Python libraries to support this functionality.

Next, I stood up the Flask web server and registered a route at `` `/` ``. When a web request is made to localhost/8000, it calls the single view function `scrape()`, which initially:

- Gathered the search term and URL contained in the client request;
- Made a GET request to the URL;
- Parsed the requested source content and identified a count of the search term;
- Returned a response to the client with a JSON object containing the search term, URL, and count;

After confirming that requests to the backend were succeeding, I created the React app. This frontend serves a single page where a user can enter a URL and search term into an HTML form. Submitting this form makes a request to the Flask app at `` `/` `` with the given URL and search term appended to the request URL, e.g. `` `?url=/https://www.nba.com/&search_term=knicks` ``. All submissions return a response to the frontend, which saves those results in local memory (the React app’s State) and appends them to a table on the page.

QA and error handling

I quickly encountered errors when making requests from the backend to bad URLs. In order to guide the user to make successful requests, I added validations and error handling to make failures noisy, and I added copy to guide the user.

The Flask view function (a) validates provided URLs with a third party library, and (b) times out its request to a URL after 0.5 seconds. If the function does not successfully pass both checkpoints, it responds to the client with `“count: 1”` in the body, indicating that the request failed. This information is displayed in the results table.