Deep Finders - Documentation

CS 410 Course Project | Theme: Intelligent Browsing

Team Members

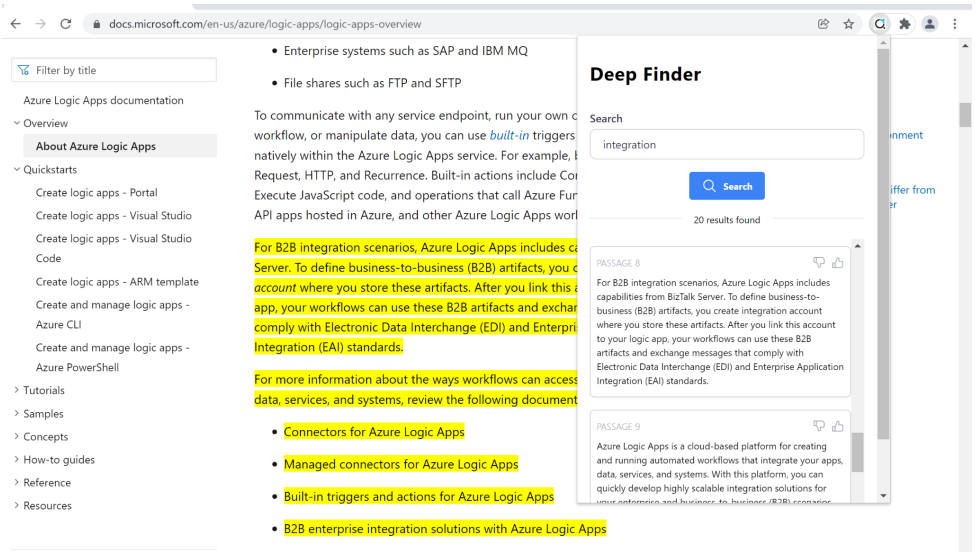
Diego Carreno (lead)	diegoac3	diegoac3@illinois.edu
Jason Ho	jasonho4	jasonho4@illinois.edu
Chris Kabat	cjkabat2	cjkabat2@illinois.edu
Robbie Li	robbiel2	robbiel2@illinois.edu

Link to video tutorial: https://www.youtube.com/watch?v=aDdWwVJH5vo

Overview

Search engines excel at surfacing results to users' queries. However, after the user has clicked into a returned result, they lose context on what parts of the document match their original query. Generally, users use the native browser find (i.e. CTRL + F or Cmd + F) and cycle through matched results to find relevant passages in the document. However, there are 2 downsides of this approach that can lead to user frustration. First, longer documents, such as technical documents, can provide too many matches than are useful to the user (ex. 100+), requiring the user to make multiple queries. Second, providing results based on exact matches could potentially miss on key pieces of information in the document that the user isn't aware of.

Our goal is to provide a mechanism for users to search over a document using BM25 by providing a ranked list of passages in the document that are most relevant to the user's query. The user will be able to click on each search result and be taken directly to that passage in the document.



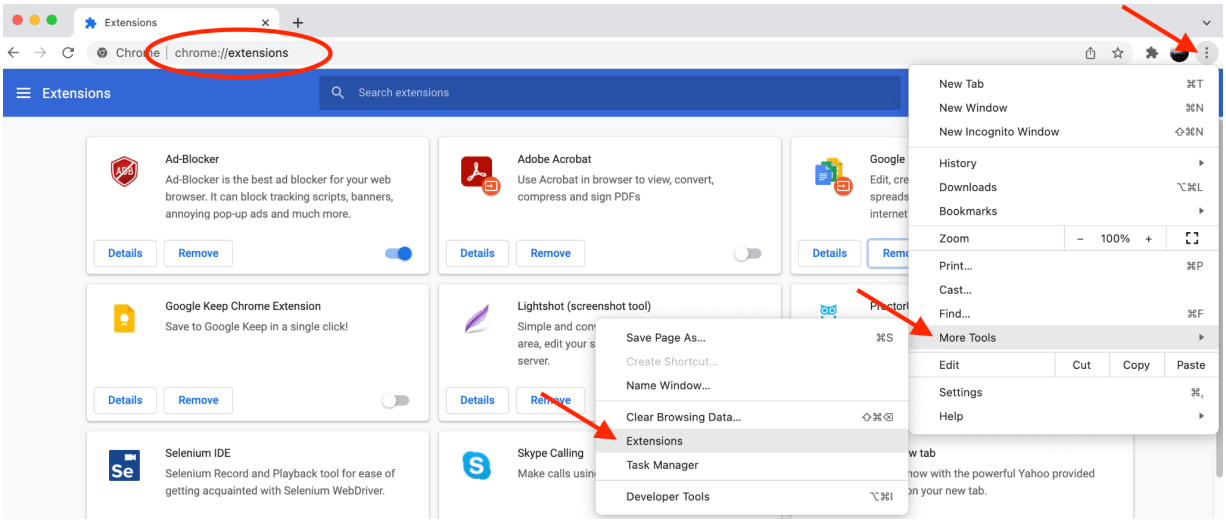
Software usage

Installation

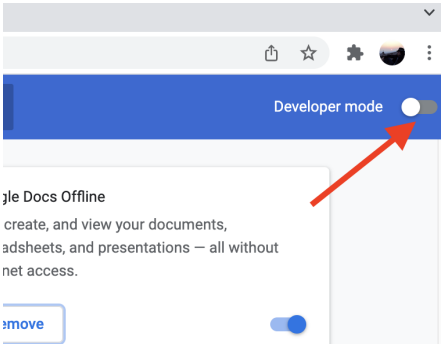
Follow these steps to install the Chrome Extension:

1. Clone the [project's Git repository] into your machine. To clone the repository, you can run the following command on terminal:
   `git clone https://github.com/deep-finders/CourseProject`

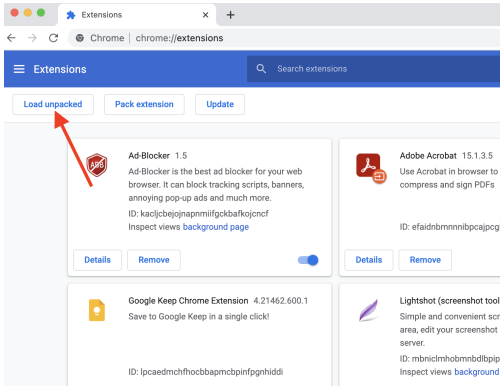
2. Open Chrome on your computer. At the top right, click More (three vertical dots) > More tools > Extensions. (Or simply go to `chrome://extensions` on the navigation bar).



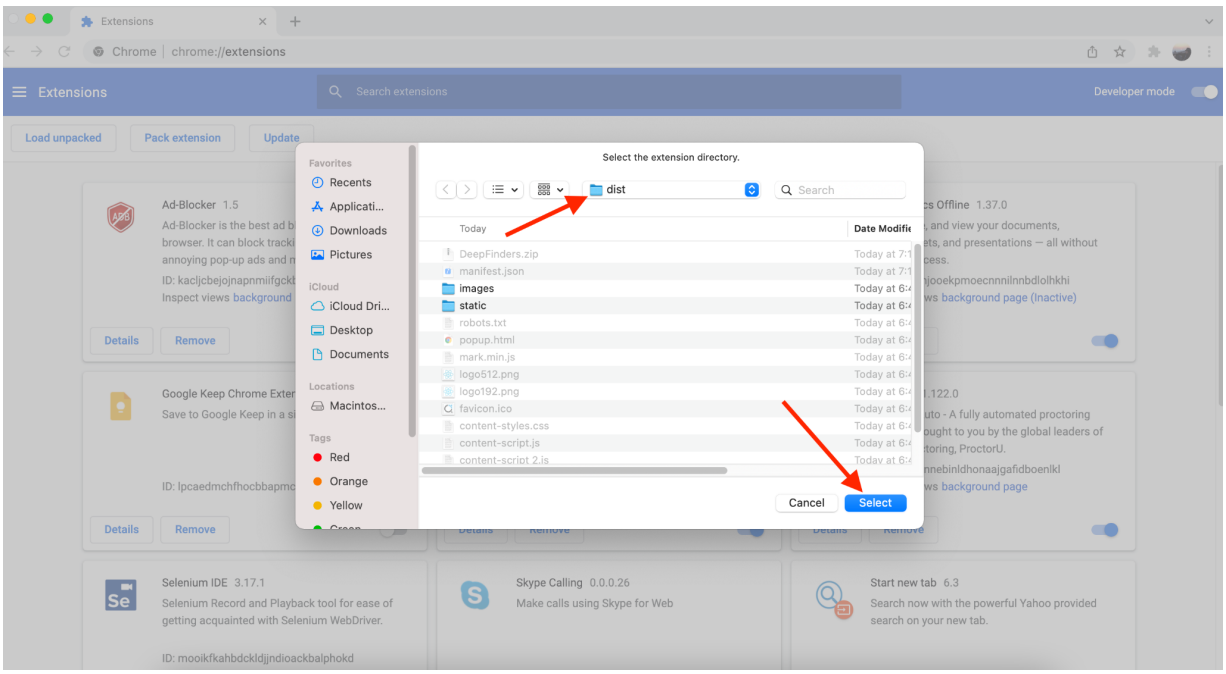
3. Within the chrome extensions view, enable developer mode on the top right.



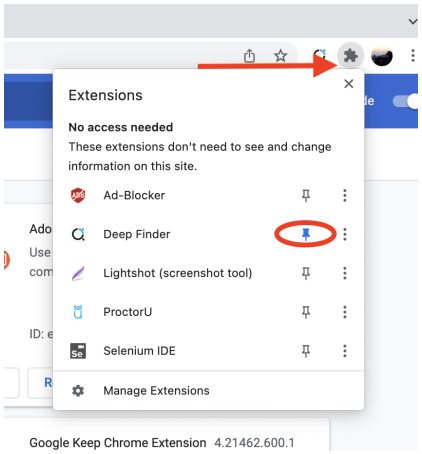
4. Once developer mode has been enabled, click on “Load unpacked” on the top left.



5. Go to the local clone of the git repository from step 1. Then go to the `deep-finder-extension / dist` folder. The dist folder within deep-finder-extension contains the chrome extension files. Click on “Select” once you’re in the dist folder.



6. Ensure the extension is visible by clicking on “Extensions” at the top right and ensuring the Deep Finder extension has been pinned.



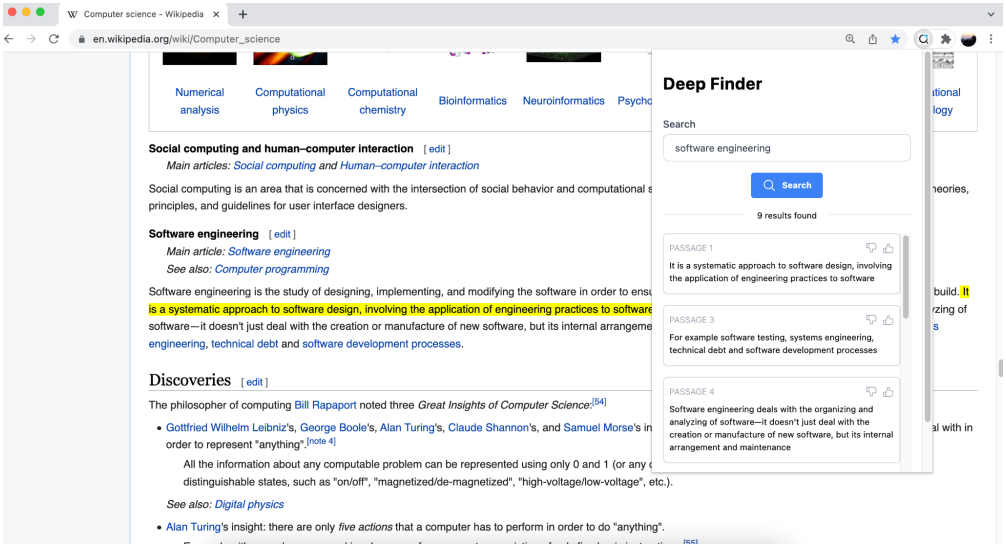
Using DeepFinder

The extension has 2 main UI features: a form with a search input and button, and a list of cards each with thumbs up and down icons.

Searching for results

To search for a result on a page, enter your query into the “Search” input field, and submit the form by either pressing the “Enter” key or clicking on the “Search” button. Once search results are found and validated, the section below it will populate with visible passages on the page.

Clicking on a passage card will highlight the passage in your active browser tab and scroll to that passage.



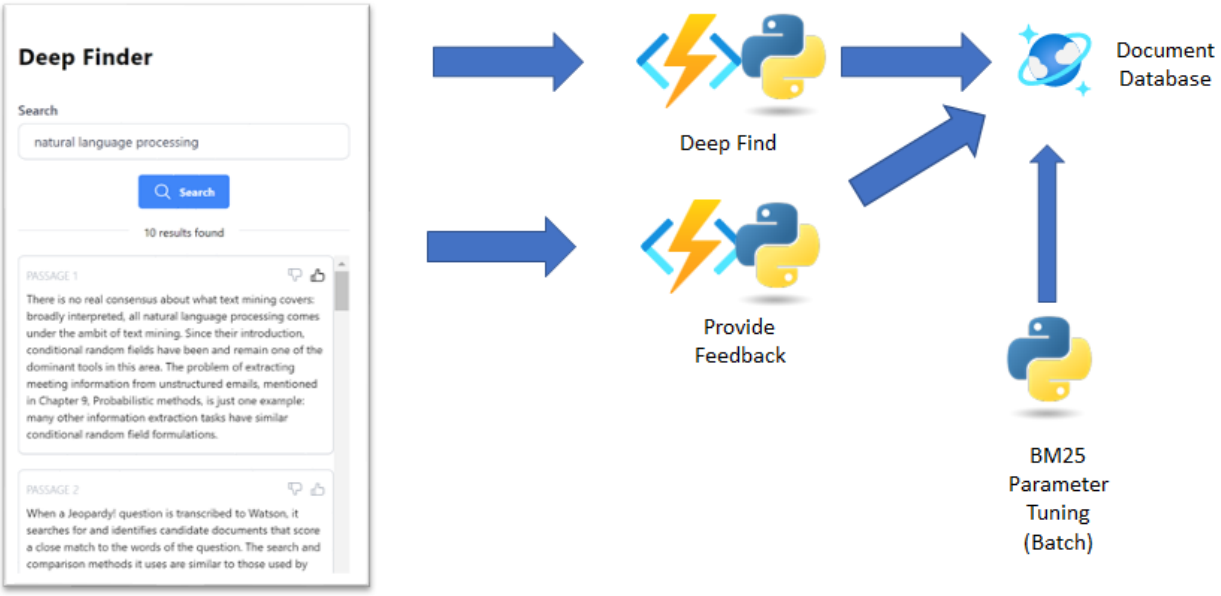
Sending feedback

If passages are returned for your search query, you may click on the thumbs up or down icons to denote if the passage was relevant or not.

Note: You will only be able to provide feedback for each search result once per returned results list!

Software implementation

The software is made of both front end and back end components. The front end component is a Google Chrome extension developed in React. The back end consists of python scripts hosted as Azure functions. It also includes a parameter tuning script.



Frontend

Deep Finder Extension

The frontend of the extension is built using React with Create React App, which provides sensible default build tools and configs for compiling and building the UI. It is styled with tailwindcss, which is a utility-first approach to styling web pages. The extension is built according to the Manifest V3 specs. Here is a quick overview of the important parts of this Chrome Extension:

- **DeepFinder.js:** The top-level React component that manages the UI state. It interfaces with *content-script.js* via the established port, and facilitates calls to the backend Azure functions.
- **content-script.js:** A plain JavaScript file that runs on the browser tab when the extension is loaded. This file instantiates a long-living connection for 2-way communication between the extension and the active tab.
- **components**: Contains reusable presentational components that represent slices of the UI.
- **services**: Contains small, reusable, modular functions that are used to call the Azure backend services. Primarily used by DeepFinder.js.

Communication with the browser

Since the Chrome Extension runs in its own separate execution environment, it does not have direct access to the DOM of the active tab. For this reason, a content-script is used in order to allow the Chrome Extension UI to communicate with the active web page. DeepFinders uses a long-lived connection by persisting a named port for content-script.js and DeepFinder.js to

communicate with one another. Each file posts and listens for specific messages that are specified in both files. Each message has the following structure:

```
{
      "action": "request_page",
      "payload": { pageHTML: '...' },
}
```

Here are the messages used to communicate between the two components:

Action	Posted By	Handled By	Description
clear_highlights	DeepFinder	content-script	Tells content-script to use mark.js to clear all highlighted passages, called both on a new search and when a passage is selected
request_page	DeepFinder	content-script	Upon clicking “Search”, DeepFinder tells content-script to grab the page HTML
send_page	content-script	DeepFinder	content-script sends DeepFinder the page HTML
find_passages	DeepFinder	content-script	DeepFinder passes the links of passages returned by the backend, and content-script pre-searches for those passages with mark.js
passage_found	content-script	DeepFinder	When a passage is found, content-script will pass it back to DeepFinder, which filters the results list to only show visible passages on the page
select_result	DeepFinder	content-script	When user clicks on a result, DeepFinder sends the passage to content-script, which looks for and highlights the passage on the page using mark.js

Communication with Azure functions

DeepFinder declares the functions that use services to communicate with the respective backend (i.e., for ranking and for feedback). The functions are passed down to child components, which define them as callback functions to click events:

- SearchForm.js triggers the search service call when the form is submitted
- SearchResult.js triggers the submit feedback call when the user clicks on either the thumbs up or thumbs down icons

Libraries Used

The following libraries were used to implement the back end software (Only the most relevant libraries are documented here):

Library	Purpose
react	JavaScript library used to build out the chrome extension UI. Creates UI structure and manages state.
mark.js	Finding and highlighting relevant passages. Only loaded into the content-script.js context via manifest.json
tailwindcss	A utility-first CSS framework used to style the chrome extension.

Other libraries used are documented in package.json and mostly provide tooling around local development and build.

Backend

The backend software consisted of 4 major components:

1. The ParagraphRanker class for executing the BM25 evaluation and creation of search results
2. The RankerDAL that communicates with the Cosmos DB database
3. Two Azure Functions. One for executing the DeepFind search and one for providing search feedback.
4. The Parameter Tuning script. An initial script to help tune the b and k1 parameters.

ParagraphRanker

The ParagraphRanker is a class that performs the preprocessing and ranking steps of our Chrome extension. Two arguments are required (raw_html and query) and there are seven optional arguments that affect the number of results returned, how paragraphs on a webpage are defined, the BM25 ranker parameters, and whether or not the paragraphs should be stemmed. These arguments can be passed through the command line or through an input JSON. The following are the arguments:

Argument: raw_html
Command Line Flags: -r, --raw_html
Type: String
Required: Yes
Description: The raw html of the web page to return paragraph rankings for

Argument: query
Command Line Flags: -q, --query
Type: String
Required: Yes
Description: The query to be searched

Argument: top_n
Command Line Flags: -t, --top_n
Type: Integer
Required: No
Description: The number of results returned in the ranking

Argument: mode
Command Line Flags: -m, --mode
Type: String
Required: No
Description: The mode we build our paragraphs with (pseudo, tag)

Argument: split_by
Command Line Flags: -s, --split_by
Type: String
Required: No
Description: Denotes how we split our text in pseudo mode

Argument: num_elements
Command Line Flags: -n, --num_elements
Type: Integer
Required: No
Description: The number of sentences per pseudo paragraph

Argument: k1
Command Line Flags: -k, --k1
Type: Float
Required: No
Description: The k1 parameter of the BM25 ranker

Argument: b
Command Line Flags: -b, --b
Type: Float
Required: No
Description: The b parameter of the BM25 ranker

Argument: stem
Command Line Flags: -c, --stem

Type: String
Required: No
Description: Determines if the text is stemmed or not. Possible values (Y, y, N, n)

At the preprocessing step, the provided raw html of the webpage is processed into paragraphs. Each paragraph is considered a document in our corpus to be ranked. This is determined by the mode specified, tag or pseudo.

In tag mode, the <p> and <td> tags (if available) of the raw html are retrieved. We consider each tag to be a paragraph. In doing so, we assume that each <p> or <td> tag consists of a reasonable amount of text data to be its own paragraph.

In pseudo mode, paragraphs are formed based on the number of “sentences” specified. These can be customized with the optional parameters. For example, if we set split_by to be ‘.’, then we assume that each sentence in the webpage is separated by periods. If num_elements is set to 3 for example, then every three sentences separated by periods will form its own paragraph. Alternatively, we could set split_by to be ‘\n’ which assumes that each sentence takes up an entire line in the webpage. This mode opens up many possibilities in how paragraphs are formed and also allows the extension to work on webpages that do not have structured <p> and <td> tags. The mode also allows the size of each paragraph to be fairly consistent.

Additionally, the default mode we use for the extension is both. This essentially processes the page through both modes and picks the mode that performs better. We determine which mode is better by the sum of the BM25 scores. The mode that returns paragraphs with the highest sum of BM25 scores is chosen.

After the paragraphs are formed, we do some cleaning before passing them to the BM25 ranker. This includes removing stop words, punctuation, stemming (optional parameter), and dropping empty paragraphs from the corpus. Once the preprocessing step is done, the corpus of paragraphs from the webpage and provided query are tokenized (separated into arrays of words), and then passed to the BM25 ranker for the ranking step. Optionally, the k1 and b parameters can be customized from the optional arguments. If not, the default parameters of k1=1.5 and b=0.75 are used.

The BM25 ranker used is the OkapiBM25 variant, which defines the IDF function differently from the variant covered in lecture. Instead of IDF = log((M+1) / (df(w))) where M is the total number of documents and df(w) is the document frequency of word w, Okapi uses IDF = log((M-df(w)+0.5) / (df(w)+0.5)). The Okapi variant’s IDF function actually allows the possibility of negative IDF values. Specifically, when a word occurs in over half of the documents in the corpus, the IDF value will be negative, resulting in a negative BM25 score. We have only noticed this occurring in our test examples with very few paragraphs so this is unlikely to pose an issue in actual web pages.

The OkapiBM25 ranker computes scores for each paragraph and then we rank each corresponding paragraph accordingly. We output a JSON file that includes the results for each paragraph. This consists of the paragraph id (for reference), rank of the paragraph, paragraph score, and paragraph text.

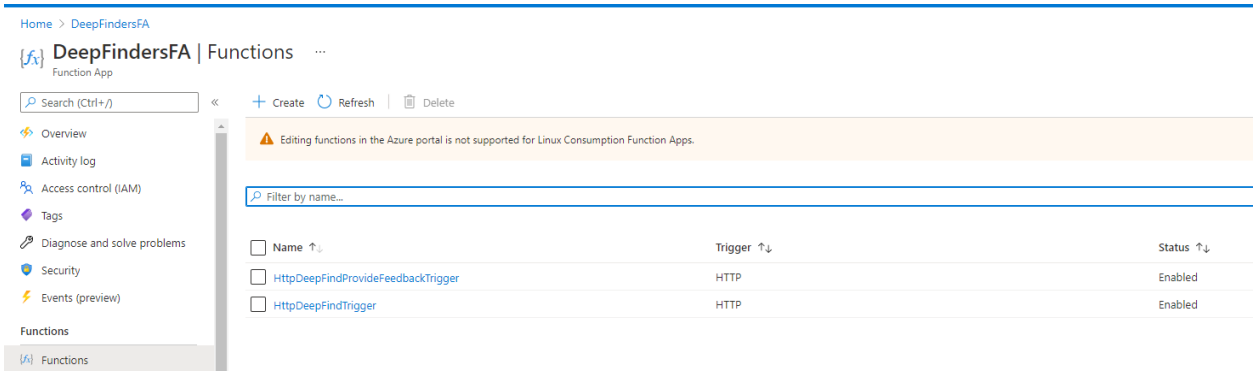
RankerDAL

The Ranker DAL is a simple class that enables easier interaction with the Cosmos DB for storing results, updating feedback, and retrieving test documents. The key to the Cosmos DB is stored as an environment variable so it is not persisted in the code. There are 3 methods:

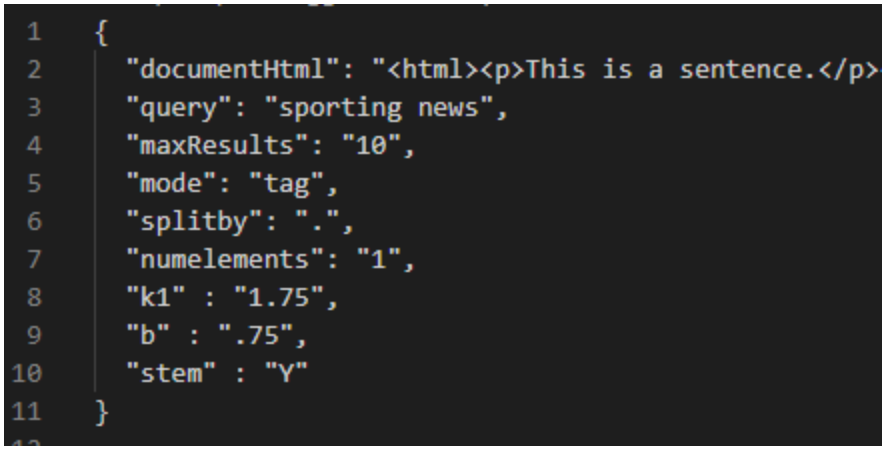
- store_rankings: Stores the results that are passed back to the extension.
- update_feedback: Using the UUID of a result and a feedback value, updates the appropriate document in Cosmos DB
- get_testset: Retrieves the entire set of documents that have any feedback provided.

Azure Functions

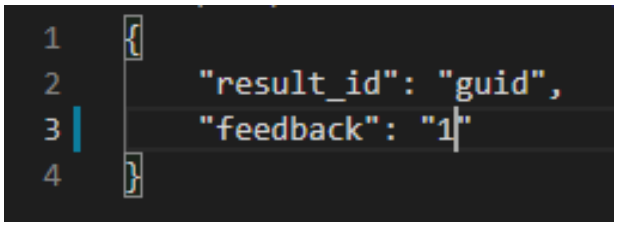
Azure Functions are a serverless compute capability that can host a Python script. The functions that we deployed allow REST calls via a HTTP trigger. The HTTP trigger passes the request a script called __init__.py. Each function has it's own function.json and host.json files that describe how it should be deployed. We deployed 2 functions in a single function app in Azure.



HttpDeepFindTrigger - Using the request structure below, this function calls the ParagraphRanker class.



HttpDeepFindProvideFeedbackTrigger - Using the request structure below, this function calls the RankerDAL class to write feedback about a result to the Cosmos DB.



Parameter Tuning Script

The parameter tuning script is a simple python file that is used to determine the best values for the BM25 k1 and b parameters. It accomplished this by varying the b1 and k parameters using a gridsearch approach with 10 b1 values between 0 and 1 and 30 k values between 0 and 3. This accounted for 300 tests each time it was run. For each run, the mean average precision (MAP) is calculated for the set of documents. Documents that had feedback provided are retrieved from the Cosmos DB database to be used as the test set of documents.

Given the short project timeline, not a lot of testing/feedback was implemented, but given the small test set we have, here is an example of the results:

	mode	b	k1	score
299	both	1	3	0.766667
298	both	1	2.9	0.766667
296	both	1	2.69	0.766667
295	both	1	2.59	0.766667
294	both	1	2.48	0.766667
293	both	1	2.38	0.766667
292	both	1	2.28	0.766667
297	both	1	2.79	0.766667
268	both	0.89	2.9	0.735417
267	both	0.89	2.79	0.735417
266	both	0.89	2.69	0.735417
269	both	0.89	3	0.735417
264	both	0.89	2.48	0.735417
263	both	0.89	2.38	0.735417
262	both	0.89	2.28	0.735417
261	both	0.89	2.17	0.735417

With the current set of test data, the MAP implies that higher b and k1 parameter settings are more effective. However, this should be tested with much more data.

Database

The database used was a CosmosDB document database using the SQL model. Below is a sample of a representative document being stored:

```
{
    "id": "c8d3cc83-a1ab-4153-af84-de737fb3de82",
    "query": "disease funding",
    "results": [
        {
            "id": "ec9e6861-be1b-41b4-9a65-4da13180d559",
            "rank": 1,
            "score": 2.9688465147063106,
            "passage": "Congress has to pass a bill to keep the government funded by Friday night to avert a partial
government shutdown.",
            "feedback": "0"
        },
        {
            "id": "74a3f62c-ce9b-457e-8c44-0b81b0c67258",
            "rank": 2,
            "score": 2.156817583766379,
            "passage": "Congress is scrambling to pass government funding ahead of a Friday deadline. But GOP objections and
delays in the process could trigger a short-lived shutdown over the weekend.",
            "feedback": "0"
        },
        {
            "id": "05e42509-8d9b-4049-be2c-5c2542d5ef5e",
            "rank": 3,
            "score": 1.4242337985256492,
            "passage": "While the initial distribution of the vaccine took longer than federal projections had indicated, in
recent months the U.S. has made great leaps in the worldwide race to administer vaccinations -- and some states are faring
far better than others. Under the current system, led by the White House COVID-19 Response Team, the Centers for Disease
Control and Prevention sends states limited shipments of the vaccine as well as funding and tasks them with distributing the
vaccine in accordance with relatively loose federal guidelines. The distribution of the vaccine is based on the size of the
adult population in every state, which -- according to some experts -- can create inequities in states where the spread of
COVID-19 is worse and a larger share of the population is at risk.",
            "feedback": "1"
        },
```


Libraries Used

The following libraries were used to implement the back end software (Only the most relevant libraries are documented here):

Library	Purpose
rank-bm25	Compute BM25 scores of each paragraph to be used for ranking the relevancy of text on a webpage.
bs4	Parse <p> and <td> tags to be used as paragraphs in tag mode.
nltk	Used to tokenize paragraph text and optionally stem the paragraph text.
gensim	Provides the list of stopwords to be removed from paragraphs.
goose3	Used to easily retrieve the entire raw text from the given raw html for the pseudo mode.
argparse	Used to allow command line arguments for the ParagraphRanker.
azure-cosmos	Provides simple access to the Azure API for reading and writing to a Cosmos DB database.

Contribution of team members

The following is a brief description of contribution of each team member:

- Jason: Implemented most of the Paragraph Ranker including the preprocessing (paragraph formation from tag and pseudo modes, text cleaning) and ranking steps.
- Chris: Implemented the Azure functions to call the Paragraph Ranker object, data access code to the Cosmos DB database, and the parameter tuning script.
- Robbie: Created initial extension and refined user interface, implemented connection with ranking and feedback mechanisms in the backend (Azure functions).
- Diego: Worked on extension development, including highlighting and navigating to selected passages. Did research and tests on using Goose-3 (for html parsing), and on using Mark.js (for highlighting).