Browser Extension for Intelligent Query Matching

Marcus Hwai Yik Tan (leader)	Omid Afshar	Xue Ying Lin
htan8	oafshar2	xylin2

1. Introduction

The built-in find (ctrl-f) function in popular browsers such as Safari and Chrome only allows for exact query matching, i.e., words in the webpage are highlighted only if they agree with the query text word-for-word in the same order and ignoring word case. Such a functionality is probably satisfactory for matching a query containing a few keywords only since the chance of exact matching diminishes with more keywords. Furthermore, the user should have accurate prior knowledge of the keyword(s) for exact match to work well. Some use cases where exact match may not be sufficient are the following:

- Searching for text units (sentences, text nodes etc.) containing some/all multiple keywords that may appear in different orders and may not appear continuously in the webpage
- Searching for text units containing not just the keyword(s) but any synonym of the keyword(s)

Intelligent Browsing is a Chrome Extension that extends exact match and caters to the above use cases by ranking/matching sentences/text nodes in a webpage based on a query text and/or enhancing the query text with synonyms of each query word.

2. Software Usage

2.1 Requirements

Google Chrome and Python 3 with the Python libraries listed in `requirements.txt` located in the `flask` folder such as `flask`, `gensim`, `nltk`, `numpy` and `pandas` are required. The Python packages can be installed by executing the following command in the terminal with `flask` as the current directory:

```
pip3 install -r requirements.txt
```

The code has been tested with Python 3.7 and Google Chrome Version 96.0. Newer versions of Python 3 such as 3.8 and 3.9 are also expected to work.

2.2 Installation

A video showing how to install and use the extension is shown here <link to tutorial video>. Sections 2.2 to 2.4 contain the same information as the video.

Start the `flask` server by running the Python script `main.py` with `flask` as the current directory:

```
python3 ./main.py
```

To load the extension, go to the URL `chrome://extensions/` in the Chrome browser. Make sure to toggle “Developer Mode” in the top-right corner so it is enabled. Click "Load unpacked" and select the extension directory. Click the Extensions icon to the right of the URL bar and the Bookmark icon. The Intelligent Browsing extension should be visible. You may pin the extension to facilitate access. By design, Chrome does not allow extensions to remain open when Chrome is no longer viewing the current window.

2.3 Use Cases

2.4 How to Use the Extension?

Type any query text (word, multiple keywords, phrases or sentences) in the search bar and press "return". Matching words of sentences are highlighted in yellow/cyan and presented in ranking order, i.e, the highest rank sentence is shown first. As the down/up arrow is clicked, the browser scrolls to the next/previous sentence and highlights the sentence in cyan. The corresponding matching words in the sentence are highlighted in yellow.

BM25 [1] is the default ranking method. Pivoted Length Normalization Vector Space Model (PLNVSM) [2] or exact match can also be selected.

By default, each text node is considered a distinct document for ranking. The user can also set each sentence as one distinct document.

2.5 Ratings (Optional)

To compare performance of different ranking functions, one can indicate the relevance (rating) of a current result with the like or dislike button. More details are available in Section 3.5.

3. Implementation Details

There are two source folders: `extension` and `flask`. The `extension` folder contains the frontend scripts to create the GUI of the Chrome extension, extract text nodes from a web page, and present results from the backend model. The `flask` folder contains Python scripts to create a backend server and model and handle user feedback. The model takes the text nodes and query text from the frontend, splits the text nodes into sentences, and ranks the sentences based on the query text.

3.1 Frontend

The structure of the `extension` folder follows the standard format of Chrome extensions [3]. Within this folder, the `manifest.json` is a required configuration for an extension that specifies various properties, including the name and version of the extension, as well as information about where Chrome can find the icons of the extension as well as the files related to the popup user interface of the extension (pictured above).

The popup UI is handled by the `popup.html`, `popup.css`, and `popup.js`. The HTML file contains the markup of the UI elements in the popup, and the CSS (Cascading Style Sheet) file adds styling to each of the elements and the overall popup. The core logic of the extension lives in `popup.js`. Within this file, event handlers are set up to listen for user interactions with the search box, search button, navigation arrows, and like/dislike buttons in the popup.

Upon receiving a user interaction to search for text, the handler for the search button, `onSearch`, will execute a script on the currently displayed page to gather up all the visible text nodes within the document, and will execute an asynchronous HTTP (AJAX) request to the backend server to obtain the rankings of the text nodes. Subsequently, the handler will execute another script on the currently displayed page to highlight the relevant text nodes accordingly.

Highlighting of search results is handled in `dom.js`. The `highlight()` function iterates through the offsets provided from the backend. The text node containing a match is split up into separate text nodes based on the provided offsets, then reinserted as new `<span>` elements into the DOM that are styled with a background color using a CSS. Since inserting these elements involves removing existing text from the current text node, and we want to highlight in a single pass, care must be taken to keep track of the already consumed offset positions. This is achieved by using the `subtracted` variable for tracking.

When executing a new search, or dismissing the search box, the `clearHighlights()` function is called to remove all highlights from the document. This finds all highlight spans created by `highlight()` and simply merges their text node content back into their parent nodes.

In order to navigate between ranked search results in the document, the extension keeps track of all the inserted `<span>` elements in an array, and the current position, and uses the `Element.scrollIntoView()` API to ensure the element containing the text node is visible in the browser viewport.

3.2 Backend server protocol

The server endpoint, `POST /search` expects the body to be a JSON object like this:

```
{
    "search_text": "some",
    "doc_content": {
        "text_nodes": [
            "some text",
            "another text"
            "this is something some"
        ]
    }
}
```
`text_nodes` is the nodeValue of all non-whitespace DOM text nodes in the document, in the order in which they appear, and `search_text` is the text the user is searching for.

The response from the endpoint is a JSON array containing indexes from text_nodes that match, and the offsets of matching text within those nodes. It can optionally also contain a wordOffsets array that gives the index of words within each match.
```
[
  {
    "index": 0,
    "offsets": [0, 4]
  },
  {
    "index": 2,
    "offsets": [0, 30]
    "wordOffsets": [
      [8, 12],
      [18, 22]
    ]
  }
]
```

3.3 Text preprocessing

`intelligentMatch.py` in the `flask` folder contains the class `IntelligentMatch`, which includes methods to preprocess the incoming text nodes and query text. Those methods use built in functions available from the Gensim package [4]. Depending on the user selection, each text node is treated as a document or split into multiple sentences, each corresponding to a document. Words in each document are first tokenized to remove white space characters and create a list of words. Any stop word given by those in the file `stopwords.txt` is removed from the document tokens. This is followed by word stemming. Next, a dictionary is built using the resulting words from all documents. Finally, each document is represented by a bag of words (BoW) using the dictionary. A query is subject to the same preprocessing and the query BoW is created with the dictionary obtained from the documents. An option to add synonyms of each query word to the query text has also been implemented using the Natural Language Toolkit (NLTK) package [5]

3.4 Document (text node/sentence) ranking

Ranking of the documents is implemented as a method in `IntelligentMatch`, which calls ranking functions in `rankingFunctions.py`. Two ranking functions are available: (i) BM25 [1] and (ii) PLNVSM [2]. For (i), we set k=k1=1.5 and b=0. For (2), we set b=0. It is also possible to perform exact matching with the extension.

3.5 Ratings (Optional)

The handlers for the like/dislike buttons, `onLike` and `onDislike`, will send AJAX requests to the backend to store a record of each result the user liked or disliked.

The backend server endpoint `POST /rate` expects the request body to contain a JSON object with the following fields:

```
{
    "url": "<the URL of the web page>",
    "query": "<the user-submitted query>",
    "result_index": "<the index of the result, based on its relevance
ranking>"
    "liked": "<a boolean value indicating if the user liked or disliked the
result>"
    "ranking_method": "<indicates the ranking method selected by the user>"
}
```

The ratings will be saved to a CSV file along with records of the average precision of the operation. The endpoint will respond with a JSON object indicating the success or failure of the operation, like so:

```
{
    "status": "<success || failure>"
}
```

`ratings.py` contains the logic to store and process user ratings of the ranking results. The current rating together with website url, query, ranking function and rank are output to the file `ratings.csv` in a directory called `ratings`. Average precision and mean average precision are stored in the files `top3-avg-precisions.csv` and `top3-mean-avg-precisions.csv`. For each website, query and ranking function, the average precision is calculated using the top-k results, where k=3 by default. It is defined as the number of relevant results divided by the rank of the last relevant result. The mean average precision is calculated for each ranking method/function over all its recorded average precisions. Ability to output such information would allow for more rigorous comparison of different ranking functions and parameters in the future.

References

[1] BM25
S. E. Robertson and S. Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval, in Proceedings of the ACM SIGIR, 1994

[2] Pivoted Length Normalization Vector Space Model
A Singhal, C. Buckley and M. Mitra. Pivoted document length normalization, in Proceedings of ACM SIGIR, 1996

[3] Getting started with Chrome extension for developers
https://developer.chrome.com/docs/extensions/mv3/getstarted/

[4] Topic modeling library (Gensim)
https://pypi.org/project/gensim/

[5] Natural Language Toolkit (NLTK)
https://www.nltk.org