Feedback-Augmented-Search

CS6111 Project 1

Team:

- **Your Name:** Pranjal Srivastava
  - **Columbia UNI:** ps3392
- **Teammate's Name:** Shreyas Chatterjee
  - **Columbia UNI:** sc5290

List of Files:

1. `feedback_augemented_search.py`
2. `query_manager/query_manager.py`
3. `query_manager/__init__.py`
4. `ui_manager/ui_manager.py`
5. `ui_manager/__init__.py`
6. `query_augmenter/query_augmenter.py`
7. `query_augmenter/__init__.py`
8. `README.md`
9. `requirements.txt`
10. `stop_words.txt`
11. `trancript.txt`
12. PDF version of `README.md`
13. PDF version of `trancript.txt`

Running the Program

To run the project, follow these steps:

1. **Setting Up Environment:**
   - Ensure you have Python installed on your system.
   - Make sure you run these two commands to ensure that you have pip installed and all the lates updates are installed

```
sudo apt update
sudo apt install python3-pip
```

2. **Install Dependencies and spacy model:** Run the following command in your working directory.

```
pip3 install -r requirements.txt
python3 -m spacy download en_core_web_md
```

3. **Running the main python file:**

```
python3 feedback_augmented_search.py [API_KEY] [SEARCH_ENGINE_ID] [TARGET_PRECISION] [INITIAL_Q
```

Internal Design

The project consists of the following main modules:

`feedback_augmented_search.py`

- Orchestrates project execution.
- Initiates QueryManager, UIManager, and QueryAugmenter instances.
- Manages the query feedback loop until the desired precision is reached.

`query_manager`

- Communicates with the Google Custom Search Engine API.
- Parses and verifies API responses.
- Filters and processes search results.

`ui_manager`

- Manages user interface interactions.
- Displays initial information and feedback summaries.
- Collects user feedback on search results.

`query_augmenter`

- Implements the core query modification method.
- Extracts relevant information from search results.
- Selects new keywords based on various criteria.

External Libraries

- numpy: for calculating the mean and getting the precision@10 scores when feedback is received.
- regex: for cleaning the titles and snippets from documents and getting a list of words present in them. also used for calculating the number of times a particular order is found in a document.

- math.log: to enhance the weights by a factor of log(1+x) while ranking the words.
- spacy: to implement dependency parsing
- itertools.permutations: to check all the possible orderings of the words

Query Augmentation Method

The QueryAugmenter module uses a four step process to select words to augment to the query. These are explained below in detail.

1. **Construction of inverse-lists**

- the `construct_inverse_list` method builds inverse lists for each word encountered in the search results. The inverse list is a list of all the documents the word is present in, formatted as follows:

```
{
    "word_1": {
        "doc_1": {"frequency": int, "close_to_query": True/False},
        "doc_2": {"frequency": int, "close_to_query": True/False},
        ...
    },
    "word_2": {
        "doc_3": {"frequency": int, "close_to_query": True/False},
        ...
    },
    ...
}
```

Here, close_to_query is True if a query term is encountered in a window parametrised by window_size in QueryAugmenter. the default window_size is 2.

2. **Filtering words_to_search based on ratio of relevant documents**

We get a list of words called words_to_search which are our candidates for being appended to the query. To filter them, we select all words such that the ratio of relevant documents in their inverse list is greater than some parameter k(=0.6 by default).

$words_to_search = {word\ |\ \frac{|documents\ \in\ inverse_list[word]|}{No. of total\ documents} \geq k}$

3. **Ranking words**

The ranking of words in the QueryAugmenter module involves a calculation of the Gini gain for each word, which serves as the basis for initializing rankings.

i. Gini Gain Calculation and Initialization

The Gini gain of a word is computed using the Gini impurity metric, which quantifies the effectiveness of the word in differentiating between relevant and non-relevant documents. This calculation is expressed as follows:

$gini_gain(word) = \textit{gini$_$impurity(base results)} - \textit{gini$_$impurity(word)} $

where gini of a set of documents is defined as:

$gini = 1 - \frac{\text{No. of relevant docs}}{\text{No. of total docs}}^2 - \frac{\text{No. of irrelevant docs}}{\text{No. of total docs}}^2$

Here:

- `gini_impurity(base results)` : Represents the baseline Gini of the entire result set.
- `gini_impurity(word)` : Reflects the weighted average of gini of the two sets: docs containing the word and docs not containing the word

Gini gain is a better heuristic than IDF because it allows us to focus on the discriminatory qualities of a word instead of its rarity in the corpus.

ii. Adding frequency weights

For each word, we consider it's frequency in the relevant documents. We then update the rankings using the following formula:

$ranking(word) = gini_gain(word) + tf_{word}$

where $tf_{word} = log(1+freq_{word})$

iii. Adding dependency weights

Finally, we update the rankings using the dependencies amongst the words. We use Spacy dependency parser. For each token, we get children and head that mark the direction of dependencies between words.

We count the following dependency relation for a word:

- Number of times it appears as a child of a query term
- Number of times it appears a child of some ROOT term along with other query terms.

Let d_{word} be the number of such dependency occurences. Then, the final ranking of a word is given as:

$ranking(word) = gini_gain(word) + tf_{word} + log(1+d_{word})$

- **iv. Selecting and ordering terms for the new query**
    - The original query keywords are kept as it is.
    - The best new candidate keywords are chosen based on highest ranking for query augmentation.
    - Either the best or the two best candidates are chosen based on thresholding of the weight difference between the highest two ranked words
    - Permuations of the final new query words are considered
    - The best permutation is chosen based on the highest number of times that particular permutation has appeared in the relevant documents (both title and snippet are considered).

Google Custom Search Engine API Key and Engine ID:

Pranjal Srivastava

- **API Key:** [Your API Key]
- **Engine ID:** [Your Engine ID]

Shreyas Chatterjee

- **API Key:** AIzaSyBdyvstyytiLSY0jPXM1FV9JfGmDaoZ3iY
- **Engine ID:** 91edb5a90f6c44a6a

Additional Information:

- Ensure your Google Custom Search Engine is configured to allow the specified API Key and Engine ID.
- Detailed information on handling non-HTML files: we decided to not specifically handle non html files since most of application/pdf type files also contained the 3 things needed by us - link, title and snippet and we wanted to use them too.

References:

- Project Idea : https://www.cs.columbia.edu/~gravano/cs6111/proj1.html
- Gini Coefficient : https://en.wikipedia.org/wiki/Gini_coefficient
- Spacy - https://realpython.com/natural-language-processing-spacy-python/