Personalized Book Search

- **Author**: Minyang Wang

- **Date**: Sept. 14, 2022

Table of Contents

1. Instructions on Running the Code

Create a Python virtual environment, activate it, and install the requirements.

Go to `main.py`, modify `user_id` (line 11) and `query` (line 12) to interested. Run the script.

There are also 3 optional parameters. You may simply leave them unchanged:

- `K`: the number of book-search results retrieving; defaults to 20;
- `to_read_boosting_factor`: the factor we multipy the final score with when the book is on the user's to-read list (further elaborated in Section 5.4); defaults to 1.5;
- `save_res`: whether to write search results to `.csv` under folder `saved_results`; defaults to `True`.

You may also take a look at `Demo.ipynb/html` for reference.

2. Problem Formulation

Given a user's past book ratings and future reading plan and a text for book query, what books should a search engine show to the user?

3. Goals

We want to build a book search engine that returns relevant books given a piece of text; we also want to build a recommender system that predicts the user's rating of a new book, given their past book ratings. We need to combine these two features to give the user a ranked list of personalized book search results.

4. Non-Goals

- We will **not** scrape book contents from the web; we only rely on book title, author, year published... in the provided datasets;
- We will **not** build a user interface for such application.

5. Features and Models

5.1 Rating

To make the search results personalized, we need to predict the rating of a user given a new book.
Thus, we need to train a Recommender System. One popular approach matrix factorization, which factorization can be realized using a package called Surprise. I applied a Singular Value Decomposition (SVD) model in Surprise reached a high r^2 score of around 0.6 - we would use it for user-item pair rating predictions.
We also saved the model weights to local.
The Model can be found at `Recommender.py` and training can be found at `RecommderTraining.ipynb/html`.

5.2 Sentence Similarity between Query Text and Book Titles

In the case when a user does not type the exact name of a book, we need to infer what book they might be referring to. For instance, if a user searches for "movie", most logically, we should return books with titles containing its synonym `film` as well. Thus, we compute the sentence similarity score between query text and all book titles.
We apply a Hugging Face sentence transformer for such a purpose. As the query a user types might be completely unrelated to any book titles in our dataset, a pre-trained model such as Hugging Face would do the trick. The model would return a similarity score between -1 and 1. We instead modify the minimum score to be 0.05 - a small positive value useful for later final score calculation.
Code for this part can be found in the `calculate_word_sim` method in `SearchEngine.py`.

5.3 Existence of Author(s) in Query Text

A user might type an author name instead of a book name! Most logically, even if I don't like Harry Potter, when I search for J. K. Rowling, Harry Potter should still be the top result. Therefore, we need to take the authors' names into the modeling.
However, for names, we can no longer adapt the sentence similarity strategy: what the transformer looks at is the semantic similarity between sentences, so it would yield low similarity scores on unrelated sentences, such as people's names. For example, say Book 1 is coauthored by Pam Beesly and Joy Nguyen, and Book 2 is written by Happy Nguyen; when we search for `Joy Nguyen`, its similarity score with `Happy Richardson` would be higher than the one with `Pam Beesly, Joy Richardson`, which doesn't help our searching.
Thus, for authors, we created an inverted index engine for the author-book pairs. If we found the query text includes an author, or the query is a substring of an author (for instance, if we type `rowling`, we should be

able to tell it refers to `J. K. Rowling`), we would give a boosting factor 30 (multiply the final score by 30). We deliberately picked this value to make sure that books by the given author would show up at the top of the list.
Of course, cleaning on author names is required, as some authors have non-English names, and we should not require, or expect, the user to type the full name. However, such a method does not allow for typo correction, which we will discuss more in Section 8.1.
Code for this part can be found at `AuthorParser.py` and in the `author_boost` method in `SearchEngine.py`.

5.4 To-Read List

If a book is on the user's to-read list, naturally, we should move its ranking up to encourage the reader to start reading it!
This part of modeling is rather straightforward, if a book is on one's to-read list, we give it a 1.5 boosting factor. Of course, we give the option of changing this boosting factor: a larger to-read boosting factor means the search engine really wants the user to begin reading the books on their to-read list!
Code for this part can be found in the `to_read_boost` method in `SearchEngine.py`.

5.5 Ranking

From the previous 4 parts, we get 4 values: `rating`, `sentence similarity score`, `author boosting factor`, and `to-read boosting factor`. We simply multiply these 4 values together and call the result `final score`. We can sort by `final score` in non-descending order and take the first `K` results as our outputs.

6. Evaluation Metrics

6.1 Average Precision @10

When a user performs a search, we ask the user to give feedback on which of the top 10 results meet their expetation.

We can define $$\text{Precision at } k = P_k := \frac{\text{# of results that meet expetaction from result 1 to k}}{k}$$ and $$\text{Average Precision @10} = \bar P_{10}:= \frac{1}{10} \cdot \sum_{i=1}^{10} P_i \in [0,1]$$

Users pay the most attention to the first result, then the second, etc., and usually people would only look at the first 10 results. Thus, a search engine should always put the most relevant results at the top. We asked the user if the results meet their expectation, so the answer would include their rating of the results.

The closer the average precision @10 to 1, the more accurate our search engine is. An average score of 0 indicates a search engine that shows stochastic results.

One disadvantage of this metric is that users have to be willing to fill out the survey and be honest, which is not the case in real life. Thus, we will purpose a second metric.

6.2 Average Click Score

In an ideal search engine, the user clicks the first result and realizes it's the website/book s/he is looking for; that's it. So there are two features: as few clicks as possible, and the clicked results should be at the top of the list.

Therefore, we define $$ \text{Click Score for a Query j} = C_{j} := \begin{cases} \frac{1}{m} \sum_{i=1}^m {(i \cdot p_i)}^{-1} & m > 0\text{ clicks have been made for this query j}\ 0 & \text{no click has been made} \end{cases}
$$ where p_i is the index (1-based) of the result of click i. $$ \text{Average Click Score} = \bar C := \frac{1}{N}\sum_{n=1}^{N}C_n \in [0,1] $$ where N is the total number of queries made during a period of time. If for every single query, the user clicks and only clicks on the first result, this metric will be 1, indicating an ideal search engine.

If for every single query, the user makes no click, or clicks on too many rear-located results , the metric would be very close to, if not equal to, 0, indicating a search engine that shows stochastic results.

This metric is unsupervised and therefore also allows for A/B testing to study the effectiveness of newly-added features.

7. Features for Future Development

7.1 Categorical Search

We have a dataset on book tags! Yet we did not use it at all in the current search engine. For future development, we can: a. allow for search of category names, such as `horror books`; b. compute a categorical similarity score, i.e. number of shared tags, between books to further improve the personalized search result.

7.2 Exact Search

Inspired by Google Search, we can add the exact search feature: when user put some words or phrases between quotes, we'll only show book results that contain those exact words or phrases, such as `"yes prime minister"`. To do so, we will need to create a Inverted Index Engine for word-book pairs, word-bigram-book pairs, word-trigram-book pairs, and so on, and we would simply set the similarity score to 1 for matching books and -1 for non-matching books.

8. Other Things to Try

8.1 Typo Detection

Our current search engine would not return the optimal results if we have some misspellings in the query. This problem is extremely serious when we search for author names: if we type `J K Rowling` as `J K Rolling` (that was my mistake when I tested my model 😦), the model, searching for an exact match of author names, simply does not display her books at the top of the list. Currently, Hugging Face has this transformer for typo detection. However, its problem is that it does not work well with people's names, nor would it return the inferred "correct" word or phrase based on the sentence context. Google uses Query Rank that is calculated based on query frequency: how often this exact query is searched, and revision frequency, how often is this query refined. A low query rank - low query frequency and high revision frequency - means the engine should more aggressively fix the query. It then considers semantic, syntactic, behavioral, or any combination of the above to suggest the correct query. We do not have any user interaction data to this does is applicable to our project, but adding such a feature in the future would significantly improve user experience.

8.2 Research into New Algorithms for Book Search

Currently, one problem with our model is that it calculates the sentence similarity score between the query text and every single book title; with encoding and large-scale calculations, the transformer is infamously slow: it takes around 40 seconds for one query search on my 2019 8-GB RAM laptop (yes I know I should get a new one 😦). Just think about the appalling absurdity of asking the user to wait for 40 seconds before seeing a result... Absolutely intolerable for a real-world search engine, let alone that we only have 10k books.

Google's PageRank pays the most attention to the number of other Web pages that link to the page in the query to determine one web page's relevance, but such an approach does not fit our project. PageRank also looks at the frequency keyword appears on one web page, but we are only given book titles here, so computing the similarity score should still be considered a good approach here.

Nevertheless, I have noticed that some book search websites I frequently use such as Library Genesis use exact text matching instead of considering synonyms. This approach would increase the search speed, but it does pose a challenge requirement to the users: the users will have to get the exact title from another search engine such as Google first. However, a search engine should be "selfish", trying its best not to redirect the user to other websites.

Calculating all similarity scores and looking solely at exact matches are two ends of the extreme, and I will keep looking into this topic and try to find a new algorithm that is closer to the balance point.

9. Afterwords

As I always do, when more fundamental knowledge on search engines and recommender systems has been gathered, in the fullness of time, at the appropriate juncture, when the moment is ripe, I will definitely come back to revisit this project and make necessary updates using newly-learned techniques!

Any comments and/or suggestions to the current version are very much appreciated!