Homework 3 - Which book would you recomend?

Goal of the homework: Build a search engine over the "best books ever" list of GoodReads. Unless differently specified, all the functions must be implemented from scratch.

1. Data collection

For this homework, there is no provided dataset, but you have to build your own. Your search engine will run on text documents. So, here we detail the procedure to follow for the data collection.

1.1. Get the list of books

We start from the list of books to include in your corpus of documents. In particular, we focus on the best books ever list. From this list we want to **collect the url** associated to each book in the list. As you realize, the list is long and splitted in many pages. We ask you to retrieve only the urls of the books listed in the first 300 pages.

The output of this step is a `.txt` file whose single line corresponds to a book's url.

1.2. Crawl books

Once you get all the urls in the first 300 pages of the list, you:

1. Download the html corresponding to each of the collected urls.
2. After you collect a single page, immediatly save its `html` in a file. In this way, if your program stops, for any reason, you will not loose the data collected up to the stopping point. More details in **Important (2)**.
3. Organize the entire set of downloaded `html` pages into folders. Each folder will contain the `htmls` of the books in page 1, page 2, ... of the list of books.

Important

Due to the large amount of pages you need to download, we give you the following tipds that help you to speed up some time-consuming operations.

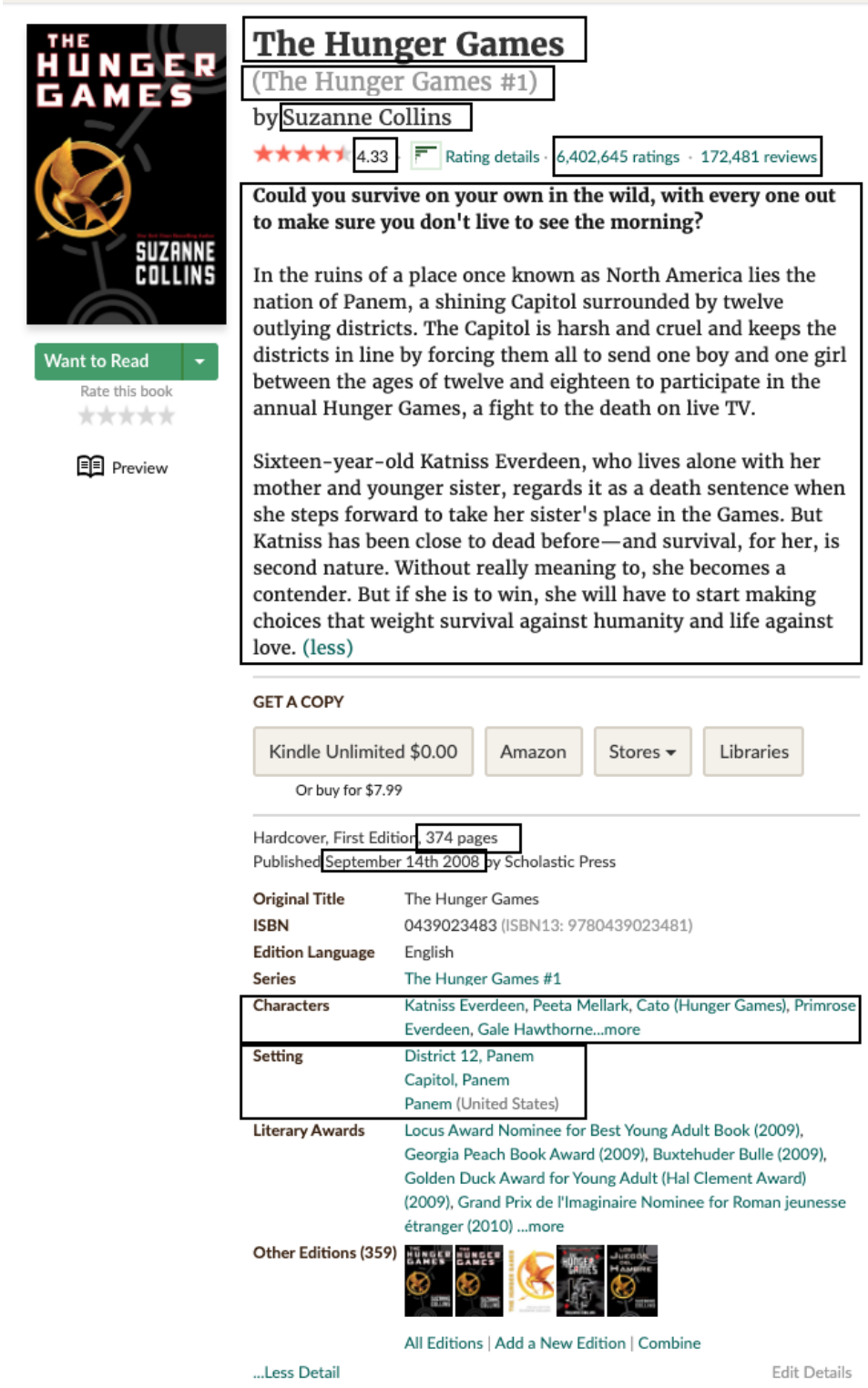
1. **[Save time downloading files]** You are asked to crawl tons of pages, and this will take a lot of time. To speed up the operation, we suggest you to work in parallel with your group's colleagues. In particular, using the same code, each component of the group can be in charge of downloading a subset of pages (e.g., the first 100). **PAY ATTENTION:** Once obtained all the pages, merge your results in a unique dataset. In fact, the search engine must look up for results in the whole set of documents.

2. **[Save your data]** It is not nice to restart a crawling procedure, given its runtime. For this reason, it is extremely important that for every time you crawl a page, you **must** save it with the name `article_i.html`, where `i` corresponds to the number of articles you have already downloaded. In such way, if something goes bad, you can restart your crawling procedure from the *i+1*-th document.

1.3 Parse downloaded pages

At this point, you should have all the html documents about the books of interest and you can start to extract the books informations. The list of information we desire for each book are the following:

1. Title (to save as `bookTitle`)
2. Series (to save as `bookSeries`)
3. Author(s), the first box in the picture below (to save as `bookAuthors`)
4. Ratings, average stars (to save as `ratingValue`)
5. Number of givent ratings (to save as `ratingCount`)
6. Number of reviews (to save as `reviewCount`)
7. The entire plot (to save as `Plot`)
8. Number of pages (to save as `NumberofPages`)
9. Published (Publishing Date)
10. Characters
11. Setting
12. Url



For each book, you create a `article_i.tsv` file of this structure:

```
bookTitle \t bookSeries \t  ... \t Setting
```

If an info it is not present, you just leave it as an empty string.

Example:

```
bookTitle \t bookSeries \t  ... \t Setting

The Hunger Games \t The Hunger Games \t ... \t District 12, Panem, Capitol, Panem,
Panem (United States)
```

Important

It may happen that the plot is not in English (e.g., The Pastures of Heaven). So, once you get the plot of a book, check if it is written in english. To do it, we suggest you to use langdetect. If the language is **not** English, just discard the book.

2. Search Engine

Now, we want to create two different Search Engines that, given as input a query, return the books that match the query.

First, you must pre-process all the information collected for each book by

1. Removing stopwords
2. Removing punctuation
3. Stemming
4. Anything else you think it's needed

For this purpose, you can use the nltk library.

2.1. Conjunctive query

For the first version of the search engine, we narrow our interest on the `Plot` of each document. It means that you will evaluate queries only with respect to the book's plot.

2.1.1) Create your index!

Before building the index,

- Create a file named `vocabulary`, in the format you prefer, that maps each word to an integer (`term_id`).

Then, the first brick of your homework is to create the Inverted Index. It will be a dictionary of this format:

```
{
term_id_1:[document_1, document_2, document_4],
term_id_2:[document_1, document_3, document_5, document_6],
...}
```

where *document_i* is the *id* of a document that contains the word.

Hint: Since you do not want to compute the inverted index every time you use the Search Engine, it is worth to think to store it in a separate file and load it in memory when needed.

2.1.2) Execute the query

Given a query, that you let the user enter:

```
survival games
```

the Search Engine is supposed to return a list of documents.

What documents do we want?

Since we are dealing with conjunctive queries (AND), each of the returned documents should contain all the words in the query. The final output of the query must return, if present, the following information for each of the selected documents:

- `bookTitle`
- `Plot`
- `Url`

Example Output:

bookTitle	Plot	Url
The Hunger Games	...	https://www.goodreads.com/book/show/2767052-the-hunger-games
Harry Potter and the Goblet of Fire	...	https://www.goodreads.com/book/show/6.Harry_Potter_and_the_Goblet_of_Fire
Catching Fire	...	https://www.goodreads.com/book/show/6148028-catching-fire

If everything works well in this step, you can go to the next point, and make your Search Engine more complex and better in answering queries.

2.2) Conjunctive query & Ranking score

For the second search engine, given a query, we want to get the *top-k* (the choice of *k* it's up to you!) documents related to the query. In particular:

- Find all the documents that contains all the words in the query.
- Sort them by their similarity with the query
- Return in output *k* documents, or all the documents with non-zero similarity with the query when the results are less than *k*. You **must** use a heap data structure (you can use Python libraries) for maintaining the *top-k* documents.

To solve this task, you will have to use the *tfIdf* score, and the *Cosine similarity*. The fielf to consider it is still the plot. Let's see how.

2.2.1) Inverted index

Your second Inverted Index must be of this format:

```
{
term_id_1:[(document1, tfIdf_{term,document1}), (document2,
tfIdf_{term,document2}), (document4, tfIdf_{term,document4}), ...],
term_id_2:[(document1, tfIdf_{term,document1}), (document3,
tfIdf_{term,document3}), (document5, tfIdf_{term,document5}), (document6,
tfIdf_{term,document6}), ...],
...}
```

Practically, for each word you want the list of documents in which it is contained in, and the relative *tfIdf* score.

Tip: *tfIdf* values are invariant with respect to the query, for this reason you can precalculate them.

2.2.2) Execute the query

In this new setting, given a query you get the right set of documents (i.e., those containing all the words in the query) and sort them according to their similairty to the query. For this purpose, as scoring function we will use the Cosine Similarity with respect to the *tfIdf* representations of the documents.

Given a query, that you let the user enter:

```
survival games
```

the search engine is supposed to return a list of documents, **ranked** by their Cosine Similarity with respect to the query entered in input.

More precisely, the output must contain:

- `bookTitle`
- `Plot`
- `Url`
- The similarity score of the documents with respect to the query

Example Output:

bookTitle	Plot	Url	Similarity
The Hunger Games	...	https://www.goodreads.com/book/show/2767052-the-hunger-games	0.96
Harry Potter and the Goblet of Fire	...	https://www.goodreads.com/book/show/6.Harry_Potter_and_the_Goblet_of_Fire	0.92
Catching Fire	...	https://www.goodreads.com/book/show/6148028-catching-fire	0.87

3. Define a new score!

Now it's your turn. Build a new metric to rank books based on the queries of their users.

In this scenario, a single user can give in input more information than the single textual query, so you need to take into account all this information, and think a creative and logical way on how to answer at user's requests.

Practically:

1. The user will enter you a text query. As a starting point, get the query-related documents by exploiting the search engine of Step 3.1.
2. Once you have the documents, you need to sort them according to your new score. In this step you won't have anymore to take into account just the `plot` of the documents, you **must** use the remaining variables in your dataset (or new possible variables that you can create from the existing ones...). You **must** use a heap data structure (you can use Python libraries) for maintaining the *top-k* documents.

> **Q:** How to sort them? **A:** Allow the user to specify more information, that you find in the documents, and define a new metric that ranks the results based on the new request.

N.B.: You have to define a **scoring function**, not a filter!

The output, must contain:

- `bookTitle`
- `Plot`
- `Url`
- The similarity score of the documents with respect to the query

---

4. Make a nice visualization!

IMPORTANT: This is a bonus step, thus it's not mandatory. You can get the maximum score also without doing this. We take this into account this, **only if** the rest of the homework has been completed.

Our goal is to quantify and visualize the writers' production.

1. Consider the first 10 BookSeries in order of appearance.

2. Build a 2-d plot where the x-axis is the *years since publication of the first book* (starting from 0), and y-axis there must be the *cumulative series page count* (all the Series start from (0,num_pages) point, which represents the first book). Since we want the **cumulative** number of page, the y-axis value of each book is added to the previous point.

[NOTE] Genrally, the book of a series is indicated as: title #number of the book in the series (e.g., The Hunger Games #1). Sometimes you will find the entire book serie as one book (e.g., The Hunger Games #1-3). You only retain the first type.

5. Algorithmic Question

You are given a string written in english capital letters, for example S="CADFECEILGJHABNOPSTIRYOEABILCNR." You are asked to find the maximum length of a subsequence of characters that is in alfabetical order. For example, here a subsequence of characters in alphabetical order is the "ACEGJSTY": "C**A**DFE**CE**IL**GJ**HABNOFP**ST**IR**Y**OEABILCNR." Among all the possible such sequences, you are asked to find the one that is the longest.

Define as X[i] = "the length of the longest sequence of characters in alphabetical order that terminates at the *i*-th character". One can prove that

```
X[i] = 1 + max{X[j]; j = 0, ..., i-1, such that S[j]<S[i]}
```

```
X[i] = 1, if there does not exist such a j.
```

1. Write a recursive program that, given a string, computes the length of the subsequence of maximum length that is in alphabetical order. Try some examples. Are the examples of short strings correct? Can you find examples that your algorithm does not terminate in reasonable time?
2. Show that the running time of the algorithm is exponential.
3. Write a program that computes the length of the subsequence of maximum length, using dynamic programming.
4. What is its runtime complexity?

BONUS

Prove that the formula for X[i] given above is correct.

Have fun!