Inverted Index

Aurora Zuoris
aurora.zuoris101@alu.ulgpc.es

Alejandra Ruiz de Adana Fleitas
alejandra.ruiz104@alu.ulpgc.es

Lam Truong Nguyen
lam.nguyen101@alu.ulpgc.es

Aris Vazdekis Soria
aris.vazdekis101@alu.ulpgc.es

Jaime Ballesteros Domínguez
jaime.ballesteros101@alu.ulpgc.es

Anna Barbara Król
anna.krol101@alu.ulpgc.es

October 9, 2023

Abstract

In this project, we undertook the task of creating an inverted index in Python to enhance search efficiency among a wide variety of text documents. The primary challenge we faced was the need to quickly access documents containing specific terms within an extensive library. To accomplish this task, we followed these steps: we collected a hundred English books, tokenized them to split them into words, built an inverted index that could relate these terms to the specific ID of each book, and developed a series of tables that allowed us to associate both the tokenized words with the IDs and these IDs with the metadata specific to each book. As requested, we created several versions of indexers to determine which one was better. After conducting the respective tests, we concluded that the NLTK indexer is the most optimal.

1 Introduction

In today's world there is an ever increasing number of documents that are being created and stored digitally. Thus the need to easily find and access these documents becomes ever more important. This is where the inverted index comes in. An inverted index is a method of creating indices for a given set of documents that allows for fast searching of the documents based on the words they contain. This is done by creating a dictionary that maps words to sets of documents that contain them.

2 Methodology

First, we have made a data collection program that downloads hundreds of documents from the Gutenberg Project's collection of documents. Then,

we have made a data processing program that tokenizes the documents to split them into words and create the inverted index. For this task, we first gather the data and process the relevant documents by loading them from the specified directory. Next, we will create a reverse keyword index of words and documents to enable efficient searching of terms within the documents. Finally, we store the results of the indexing in an SQLite database for easy access and future querying.

2.1 Data collection

For an inverted index to be useful, there must first be a large set of documents to index. For this task, we will be using the Gutenberg Project's collection of documents.

First, we are going to import the libraries sys, os, json, re, asyncio, and aiohttp. Next, we'll create the directory and specify the path where we want it to be created. Now, we're going to create 5 functions: one for web scraping (transform), another to fetch the content of a book in text format, one to download the content of a book (fetch), one that creates a dictionary with information extracted from the text (save_text), and a function that handles the process of downloading and storing books (main).

The functions mentioned work as follows:

- In the "transform" function, we will create a list of tuples where each tuple contains the name we will assign to an attribute and a regular expression associated with that attribute to extract the corresponding information from the text.On the other hand, we will also create two regular expressions that will help us find the start and end of the book's content. Next, we will create a loop to search for matches in the text using regular expressions, storing that information in the variable "m." In this way, we will check if the variable "m" exists, which means a match has been found. If it is verified, we will extract the matching content and store it in a dictionary "data" under the name of the corresponding attribute; otherwise, it will raise an error message. Finally, we will take the variables that indicate the start and end of the book's content to extract only the text content in between. If an error occurs during this procedure, it will raise an exception.
- The "fetch" function is an asynchronous function used to make an HTTP GET request to a specific URL. We will use two arguments: "session" (an aiohttp session for asynchronous HTTP requests) and "id" (the book's identifier). We will use the "id" to construct the URL from which we will retrieve the book's content. Using the "session" and "id" arguments, we will initiate an asynchronous HTTP request and wait for the response using await response.text(), waiting for the HTTP request to be available and then getting the response content as text. This way, the function also retrieves the HTTP response status using response.status and returns it along with the book's content in text format

- The "save_text" function is an asynchronous function that uses the "fetch" function to retrieve data and save it to a JSON file. First, it retrieves some data using the "fetch" function and the "session" and "id" arguments. Next, it checks if the status returned by "fetch" is 200. If it's not 200, it means there's an error, and it displays an error message indicating that the data could not be obtained for the provided id. If the status is 200 (indicating a successful retrieval), it proceeds to process the data in the "text" variable. Specifically, it replaces Windows-style line breaks ("\r\n") with Unix-style line breaks. Then, it calls a function called "transform" with the "id" and the processed "text" as arguments to extract attributes of the book and the main content. The result is stored in the "data" variable. Finally, it opens a JSON file in text writing mode. The file name is constructed using the provided "id" as part of the file name, and it is created in the directory specified by the "datalake_dir" variable. The "json.dump" function is used to write the content of the "data" variable to the opened JSON file.
- The "main" function starts by creating an asynchronous HTTP session using aiohttp.ClientSession(). It then asynchronously calls the "save_text" function to process data in a specific range of values (0 to 999 in this case). This allows HTTP requests to be made concurrently to improve the efficiency and performance of the application. This concurrent approach is often used in web scraping or data retrieval tasks to speed up the process of fetching data from multiple sources.

Finally, the program will be executed if __name__ == "__main__":, and this will trigger the execution of the "main" function, running the entire program.

2.2 Data processing

Once we have the documents, we will need to process them to make them easier to work with. This part is implemented in two different ways, each one using a different library to tokenize the documents. Aditionally, both share some common code, which is abstracted away. This mostly consists of accessing the documents in the datalake and saving the results of the indexing in an SQLite database.

2.2.1 spaCy indexer

```
def indexer(documents):
  reverse_index = defaultdict(set)
  for doc_id, document in documents:
    texto = document.get("text")
    texto_sobrante = texto

    while len(texto_sobrante) > 0:
      if len(texto_sobrante) > 1_000_000:
        candidato = texto_sobrante[:1_000_000]
```

```
            index = candidato.rfind(" ")
            texto_seccion = texto_sobrante[:index]
            texto_sobrante = texto_sobrante[index:]
        else:
            texto_seccion, texto_sobrante = texto_sobrante, ""

        nlp = spacy.load("en_core_web_sm")
        doc = nlp(texto_seccion) # Crea un objeto de spacy tipo nlp
        lexical_tokens = [t.orth_ for t in doc if not (t.is_punct or t.is_stop)]
        unique_tokens = set(lexical_tokens)

        for word in unique_tokens:
            reverse_index[word].add(doc_id)
    return reverse_index
```

The invert indexer pretends to do the following tasks: The processing of JSON documents to extract unique token words, create a reverse index which is going to be implemented as a dictionary and the creation of a database.

The first part of the indexer is a function called indexer which takes the documents extracted with the scrapper in JSON format. The indexer purpose is to get the content of each document and create a unique_tokens list containing all the words of each document which is relevant and not having to account those who are not relevant as for example prepositions, conjunctions, questions words, etc.

To make this possible we implemented a library called spacy which is capable of reading all the characters in the documents and detect only the relevant words. These documents are read and divided into smaller sections of text, each containing a maximum of one million characters. The purpose of this division is to prevent memory issues as the spacy library only reads a maximum of four million characters. It ensures that these sections are split at appropriate positions (i.e., spaces) to avoid breaking words in the middle. The loop continues until all portions of the original text have been processed, making it suitable for handling extensive textual data efficiently.

Later we have processed the words and tokenized them in a list, those words are going to be placed in the reverse index. The reverse index is a dictionary where each word is a key, and the associated value is a set of document identifiers in which that word appears.

A function called document_generator generates document data by iterating through a directory route which is introduced by the user and must contain the documents generated by the datalake. For that iteration we used the tqdm library to indicate the progress of reading the books as the database is too long.

Finally when executing the file itself the program takes the arguments introduced by the user, expecting two (datalake directory and datamart path). Then it creates a table called inv_index which contains two columns: words and list of books. This is the inverted index table. Then it is created another table called documents which contains all the metadata of all the documents.

2.2.2 NLTK indexer

```
def indexer(document_list):
  index = defaultdict(set)
  for doc_id, document in document_list:
    text = document["text"]
    words = set(nltk.word_tokenize(text))
    for word in words:
      index[word].add(doc_id)
  return index
```

The second implementation of the indexer uses the NLTK library. This python script creates an inverted word index from a collection of documents in JSON format. The indexer function is defined to create the inverted word index. It takes a generator of documents as input. For each document, it tokenizes the text into words using NLTK and builds the inverted index, associating words with sets of document identifiers in which those words appear. With the NLTK library we did not encounter a problem of one million characters limit. The document_generator function generates documents from a specified directory (datalake) containing JSON files. For each JSON file in the directory, it creates a document identifier based on the file name and reads the JSON content, returning it as a tuple of the document identifier and document dictionary. The script checks the number of command-line arguments. It expects two argument, which should be the path to the datalake directory containing JSON documents and the second one which is the path where it saves the SQLite database file. Creating the Inverted Index: The main part of the code creates the inverted index by calling the indexer function with documents generated by the document_generator function, based on the specified datalake directory. Saving the Index to SQLite database: Create datamart function creates two tables in an SQLite database: "inv index" and "documents". The first table stores an inverted index mapping words to document IDs, while the second table stores information about documents, such as their titles, dates, and content. The code writes the inverted index data into it. In summary, this script processes JSON documents from a directory, tokenizes the text into words, and constructs an inverted index. This version of indexer can be used to efficiently search for and index words within the text data in our search engine project.

3 Results and conclusions

At the end of the tests, we have been able to see that indexing with the NLTK library is faster than with spaCy. Among the possible reasons we have found to understand this result are:

1. Model Complexity: SpaCy utilizes more advanced and complex language models designed for in-depth text analysis, which results in longer loading and processing times due to the depth of its analysis.
2. NLTK Optimization: NLTK has been optimized for specific text processing tasks like tokenization and text analysis, making it more

efficient in these tasks. NLTK is faster in extracting keywords and tokens.

3. Customization: SpaCy allows for model customization, which can increase accuracy but also slow down the indexing process when detailed adjustments are made.

In summary, NLTK is faster in indexing due to its lighter and task-specific text processing focus, whereas spaCy excels in deeper text analysis at the cost of higher resource consumption and time.

The code of this project can be found in the following repository: `https://github.com/Aurora2500/inverse-index-big-data`

4 Future work

There are various ways in which this project could be improved. One very critical one is to make it comply with the Gutenberg Project's terms of use when it comes to automated downloads. This would be done by using one of the mirrors provided by the project, as the main site is only meant to be used by humans. The issue with this is that surprisingly they're much less standardized than the main site, and it would require a lot of work to make the scrapper work with them. For example, instead of just using the id of the book to download it, one would need to split the id into its digits to get the path, and then there's also the issue of the file name sometimes having a suffix of `-0` and other times not. So it would be impossible to know which one is correct without trying both and seeing which one returns a 200 status code and which one doesn't.

In the future, it would be interesting to implement a custom indexer trough the use of, for example, regex to see if it is possible to improve the performance of the indexer and reduce the number of faux tokens. One beginning point of this would be to use a regex pattern like

```
(?<!\w(?:\w|'|-))\b|\b(?!(?:\w|'|-)\w)
```

This regex takes the word boundary (\b) as a base, and then tries to take into account the fact that words can contain apostrophes and hyphens, as these would be considered boundaries. This way, this regex would match the boundaries of words, including those with apostrophes and hyphens such as `don't` and `mother-in-law`, and then one could loop trough the matches to get the words. This would be a good starting point, but there is still a lot of work to be done for it to be a robust solution.

One of the biggest issues with this project and trying to deal with real world data in general is that it is very messy, and not necessarily standardized. For example, when looking at the metadata, the author field is sometimes reffered to as `author`, other times as `creator`, and sometimes as `contributor`, and there might be other variations of this as well. Another example is the delimiters between the metadata and the document content, which sometimes uses the word `START OF THIS PROJECT GUTENBERG EBOOK` and other times `START OF THE PROJECT GUTENBERG EBOOK`. These are just two of the many examples of the inconsistencies found in the data. This leads to the fact that one can't just be clever to make a solution in

one go, but instead one needs to just look trough all the data, create some solutions, find the issues with them, and then repeat the process until the results are satisfactory.

Aditionally, another issue is that the expectations of what might be inside a document aren't always accurate. For example, some documents were about listing some famous mathematical constants to a very big number of digits, so the edge case of a document consisting of mostly numbers with no word boundaries, or of these being split by newlines and what to do with these should also be considered. Another example is that some documents consisted of dictionaries of words or some other similar glossaries, so virtually every word would appear in these documents. And there may be many other different kinds of documents that contain their own edge cases. Some that we might not even be aware of.