Information Retrieval: Assignment 1

Document Search and Ranked Retrieval

Philippe Voet s0183222

Github: https://github.com/Triple1312/IR_assignment1

Introduction:

Almost everyone will have made this assignment in python because it is the easiest option. However, i hate python. And since I was allowed to use any language I wanted, I opted for my favourite language “dartlang” (https://dart.dev/) . There is only one problem, there are basically no libraries for the language , at least not that have anything to do with information retrieval. This was also a reason why I wanted to work alone for this project. I want to expand my own library on my own.

Since you’ve probably never heard of dartlang, ill give a short introduction: Dart is an object oriented typesafe language with a garbage collector. It’s syntax looks very much like java, but without the bad java stuff. You can run it either with its JIT compiler, transpile it to javascript/wasm or compile it to bytecode. It’s mostly used for it’s framework Flutter for desktop, mobile and webapps. Canonical is slowly making it the new framework for the default ubuntu applications.

Now since there are no pandas or scikit or numpy, I had to write everything myself. I already had written some of the code I needed because I was planning on writing those libraries for dart anyway, however, I had to rewrite a lot of stuff since I never tried to load 500k files and run my algorithms on them. This project gave me serious time and memory constraints, which I think were good for the development of my code. Since I was working on remaking pandas, scikit and numpy in dart, you’ll see that a lot of classes have the same names and methods, however their implementation differs immensely.

Data:

A library like pandas does not exist in dart, so we have no dataframes. So lets make them. In DataFrame.dart we have the DataFrame class. It is basically a list of columns called “DFColumn”. DFColumn is a template/generic class of a specified type. The DataFrame class has a lot of extra methods that are never used in this project but I still implemented so that it is almost as powerfull as pandas. All methods have the same names (if possible) as the pandas library. This concept will be the same for all implementations. DataFrame is mostly used to load in the csv/tsv files wich you can do with its DataFrame.csv() constructor. You have the option to give the columnTypes as a parameter for parsing to the right template class.

Another DataStructure we will need is a more mathematical. I will both need Matrices and Vectors. If you look at “Matrix.dart” you see its another template/generics class, but it works a bit weird. It has a member variable called “MatrixData<T> data”. This is to select the matrix type I want. I had to implement a normal matrix, csr matrix and a Mapped matrix for this project. I went a little overboard and implemented a lot of mathematical operation implementations for matrices like different decompositions that are’nt relevant to this project.

Vectors were especially important because for cosine similarity I need to do dot products of vectors and normalization. Optimizing these were crucial for this project. If I hadn't put time into optimizing both dot and norm , I wouldn't have been able to complete the project.

Then finally and maybe the most important datastructures are for how to handle documents. I made a Document class to hold documents. A Document consists of a filename and the content. It does not however have to be loaded into memory. You can instantiate a Document class either with all this data or you can give a file on the disk and instantiate it without even loading it. This would be especially important for implementing SPIMI.

SPIMI is also implemented. You call "fit_corpus()" which uses the implementation of Document to only load data until a given threshold, writes all the data to individual files and at the end merges all those files one by one.

We will often work with lots of documents, so instead of just putting them into a list, I made a class called "Corpus". It is basically a class with a list of Documents, but it has some other interesting properties. It allows you to load all files in a folder for example, which I had to do for this project. Since 500k+ files are a lot to load, instead of loading them one at a time, I load them asynchronous 5000 at a time in batches. Something it also can do is add modifiers for all documents in a list and automatically add them for any new documents that get added in the future. Something it also lets you do is automatically load all files in a folder asynchronous for Disk IO optimization. This sped up loading files immensly.

Indexing:

The first DataStructure we had to implement for The Project was an InvertedIndex. It is written in "InvertedIndex.dart". It simply is just a Hashmap with as key the word and a List of integers as the document indeces.I opted for a Hashmap instead of a List just because with large datasets its much faster. To create the indexer you have to call the "fit" method on a class instance. It has a method called "query" which has a parameter a String and will return all documents that contain the words of that String.

The positionalIndex structure can be found in "PositionalIndexer.dart". It uses the same fit method and the same query method as in InvertedIndexer except that the query method does not only return the relevant document containing all the words, but also returns the positions of the words in the queries.

The SPIMI structure is the same as the invertedIndex only it processes the documents differently. Instead of giving it a list of strings it gets a corpus, expected to be unloaded documents and It will load with a maximum batch size the documents only once in memory.

Preprocessing:

For extra Preprocessing I firstly implemented a stopwords remover in "stopwords.dart" It's such a long time ago a that I don't remember where I got this list from. It has a method "removeStopWords" that removes stopwords.

The second preprocessing I implemented is a PorterStemmer. It is based on the paper and Martin's extention. I tested it thoroughly using nltk, If you want to do it yourself, make sure to disable nltk's own extention. The Porterstemmer has a parameter to also remove stopwords. The implementation is based on the original paper

VSM: TFIDF:

My vsm implemnentation is written in “TfidfVectorizer.dart”. From the name alone you can see the parallels with scikit. All methods have the same names as the one from scikit and it looks the same from the outside, but the implementations are vastly different. For one, I don’t only implemented just a normal VSM, I implemented every type of SMART Tfidf

To create the model, as always you have to call “fit_transform” but since we are working with a corpus, I will call “fit_transform_corpus”. For my implementation I used a List of hashmaps with integers as a key and integers as a value. This has as an idea that it has extremely fast access and addition. Every map in the list corresponds to a document with as key the vocabulary index and as value the amount of times that word is present in the document. This made it so that I at first didn’t have to save all the idfvalues, but I changed this to saving all the idfvalues and the initial matrix and remove the map to save space. This helped me save space and do much faster query. I also make a vocabulary list to translate the vocab indexes, which is also a map for fast access and also a Map that defines in what document every word appears.

You would think that I save a lot of things for just a VSM and you’d be right. It takes a significat amount of space, but this way I wanted to minimalize the amount of time it would take to index the documents, which did work. It is very fast for the amount of documents it has to go through.

After a fit you can use transform on the vectorizer to get the scores. I also built a “query” method. This method gives the k most relevant documents. It does this by taking the cosine similarity with every document. For this method to be fast I had to really optimize my dot product and my norms. Even though I made them about a thousand times faster(no kidding).

Even though I did so much to make the query faster at first it wasn’t fast enough (would have taken about 100 hours), but I dug into my code and saw that I repeated a lot of operations that I could avoid. In the code I first write everything to a file and then reread all the data from the file. I did this as an intermediate step. Then the processing is done further in the “checkpredictions()” function in main.

In “QueryPerformance.dart” I made some methods for the Map@k and Mar@k and ran them. The results:

- Map@3: 0.44444444444444486
- Map@10: 0.2976121702291529
- Mar@3: 0.0884864514917354
- Mar@10: 0.19470504639887443

I definitely wouldn’t say they are the best results, but still, for just a tfidf on 500k+ documents with cosine similarity I am pretty satisfied. I think I do a lot of operations that are unnecessary and lose some values due to rounding in early stages. I still think it could have been better. I would have thought I would get a higher value with Map@3. I don’t know how much other engines score, but If other engines can get a score of 1 than this wouldn’t be that great. Seeing how “low” my Map@3 is, I would expect my Map@10 to be much worse that it actually is.

Something I saw immediately was how low my Mar@3 was. At first I was a bit shocked, but then I went to look at the result set and saw that at some queries there were way more than 10 relevant documents (which I expected everywhere). This affects the Mar@k very heavily. The Mar@k is also lower for the same reason.

Conclusion:

I had a hard time with this assignment mostly because my implementation was lacklustre at first. I redid a lot of computations again and again unnecessarily and in testing you don't feel that, but when you're trying to do that on 500k files you definitely feel it. I thought it was pretty fun to do it in a language with no support where I had to do everything form scratch and am glad I did it. I definitely learned how to dig into your code and look at what can be optimized.