Project #1 - Indexer/Search Engine

Daniel Marin and Jennifer Vicentes

October , 2024

Contents

1 Class Diagram

The Class Diagram contained several changes throughout the development of the code. The original Class Diagram we developed was:

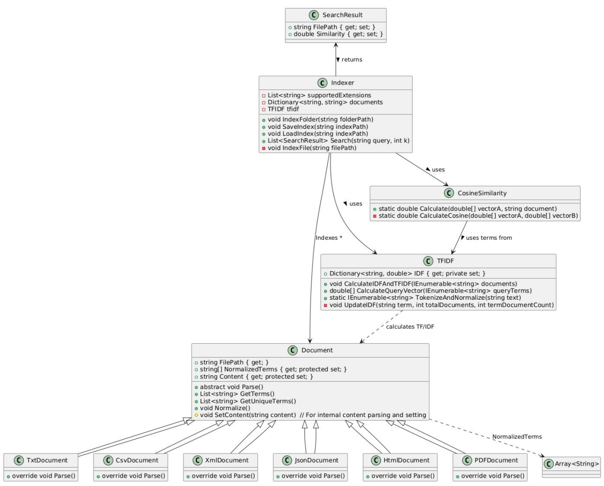

Figure 1: Image explaining the Class Diagram we developed at the beginning.

- > `index -f` <*folderPath*> `-t tfidf -dis cosine`: this command uses the standard method for both the indexer and the search engine, the **TF-IDF** for indexing and the **Cosine Similarity** for searching values.
- > `index -f` <*folderPath*> `-t tfidf -dis euclidean`: this command uses the **TF-IDF** algorithm for indexing the files and the **Euclidean Similarity** algorithm for searching values.
- > `index -f` <*folderPath*> `-t vectorizer -dis euclidean`: this command uses the **Normalized Bag of Words** algorithm for indexing the files and the **Cosine Similarity** algorithm for searching values.
- > `index -f` <*folderPath*> `-t vectorizer -dis cosine`: this command uses the **Normalized Bag of Words** algorithm for indexing the files and the **Euclidean Similarity** algorithm for searching values.
- > `load -p` <*indexPath*>: uses the path to the "`.Json`" document that holds the information of an indexed folder.
- > `search -q` <*query*> `-k` <*numberResults*>: gets the query from the user and the number of results it should return form <*numberResults*>.

This diagram managed to encompass the overall design we ended up using throughout the development of this code. Yet, various changes ended up being made to the design in the midst of the development process. The class diagram that ended up encompassing the overall flow of the backend of the program is.

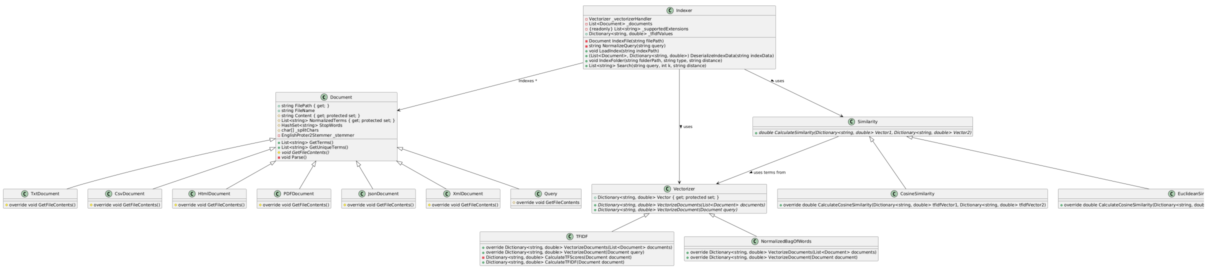

Figure 2: Image explaining the Updated Class Diagram we developed once the backend was complete. The image is cropped in the right side due to the size.

This diagram contains the end design of the program for the development of the code. It explains the various different classes and their relationships in the code without the need of accessing the program.

2 Graphical User Interface (GUI)

3 Command Line Interface (CLI)

This section will explain how each command may be inputted into the command line of the program. Following are all command variants that are valid in our project. The item list mentioned before contains all valid formats of command. All values inbetween characters '<' and '>' represent variables in the command which may change depending on what the user wants.

4 Code Overview

For this section of the documentation we will be focusing on the back-end of the code. Explaining the overall code structure and how each part correlates to eachother. Let's begin our conversation with the document handling and natural language processing section of the code.

4.1 Document Hierarchy

The document hierarchy is the way we ended up organizing how each file is handled. It is composed of a parent class, **Document** which is in charge of the overall construction of each instance, except for the **Special Child** / Query, and the contents parsing.

This class does this by allowing containing the abstract method `GetFileContents` which each child has a unique method of performing. After, extracting the files contents it finally manages to parse the contents by removing all **stop words**, **split chars**, and **stemming**each word to extract the very essence of the contents of each file[1]. With the followingmethod:

```
public void Parse(){
    string[] words = Content.ToLower().Split(_splitChars,
            StringSplitOptions.RemoveEmptyEntries);
    foreach (var term in words)
    {
        if (!StopWords.Contains(term))
        {
            var stemmedWord = _stemmer.Stem(term);
            NormalizedTerms.Add(stemmedWord.Value);
        }
    }
}
```

Figure 3: Parsing Method. Found in Document.cs.

4.1.1 The Common Children Classes

The following children classes are named the common children since they all use the same constructor, defined in the base class, but all override the `GetFileContents` method to fit their respective file to handle. The constructor method they use is the following.

```
public Document()
{
    NormalizedTerms = new List<string>();
}
public Document(string filePath) : this()
{
    FilePath = filePath;
    GetFileContents();
    Parse();
}
```

Figure 4: Constructor Functions. Found in Document.cs.

Each class in the itemized list that follows has this exact same behavior when an instance of itself gets performed.

1. `CsvDocument Class`
2. `HtmlDocument Class`
3. `JsonDocument Class`
4. `PDFDocument Class`
5. `TxtDocument Class`
6. `XmlDocument Class`

They compose all the files that the code should be able to handle, based on project prerequisites.

[1]This algorithm uses the Porter2Stemmer algorithm from: `https://www.nuget.org/packages/Porter2Stemmer`

4.1.2 The Special Child

The **Special Child** is the QueryDocument Class, which is an abstraction we implemented on the user query to maintain similar modularization throughout our code it allowed the query to be treated as a document. It was the only "concrete" class of the Document Hierarchy that never overrided the `GetFileContents` method since it never needed to do so. Since, it initialized itself using the following constructor method.

```
public Query(string query) : base()
{
    if (string.IsNullOrWhiteSpace(query))
    {
        throw new ArgumentException("Query␣cannot␣be␣null␣or␣empty.");
    }
    Content = query; // Assign the query to the content
    Parse();
}
```

Figure 5: Constructor Functions. Found in Document.cs.

Thus, causing it and others to use the exact same code for parsing, NLP and "document" handling. Lastly, any documents not covered in section 2.1 would appear as null and not handled on other sections.

4.2 Similarity Hierarchy

The similarity section of the code is in charge of defining the two different methods of searching the code was expected to have. It contains a single abstract method named `CalculateSimilarity` that defines that its children should have a way of calculating how similar two vectors are, and based on each child class it performs a different method.

4.2.1 Cosine Similarity Class

The `CosineSimilarity` class is the algorithm we were expected to create for the development of this project. Using vectors it calculates how similar the direction of two vectors is, with the following math formula.

$$\textbf{Similarity}(A, B) = cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \cdot \|\mathbf{B}\|}$$

In essence this class override the `CalculateSimilarity` method to use the similarity mathematical equation. This algorithm, for calculating how similar two vectors are, is very efficient since it works with the direction of the vector so it isn't heavily influenced by disparity in sizes. So, it is pretty good when handling a small query size. To further understand the reuslts of similarities lets look at the following picture.

Yet, it is very computationally intensive for handling big query sizes (*Document Contents as a Query*). That is why we decided on using another algorithm for our implementation of the search engine that would give it more power when handling big query sizes.

4.2.2 Euclidean Similarity Class

The algorithm that would eventually give it this edge in computation is the `EuclideanSimilarity` class. Which overrides it with a method that allows it to give it this edge. This method is in charge of calculating the difference between documents and returning the inverse output of it. Using this:

$$\textbf{Similarity}(A, B) = \frac{1}{1 + \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}}$$

Where, the bigger the difference between vectors the closer the less similar the contents of each vector are to eachother. This method is heavily influenced by the disparity in sizes between vectors A and B. Thus, it gives us more efficiency with big query sizes, but lacks precision when small query's are utilized. To, further understand the differences between both lets look at images of what each is calculating.

4.2.3 Differences between the Children

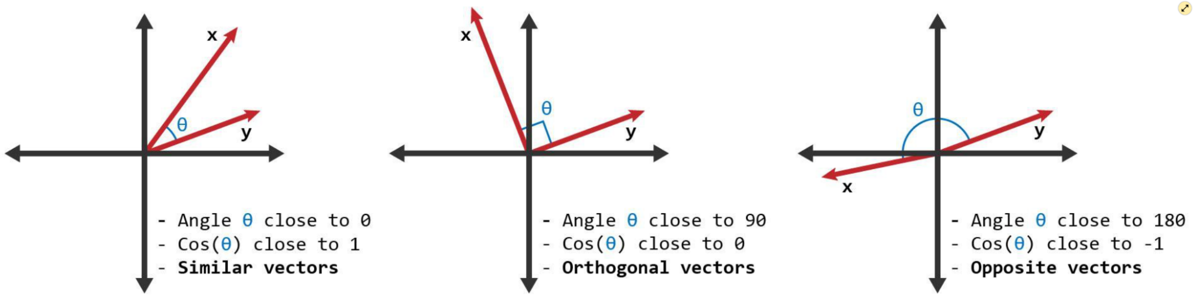

Figure 6: Image for Representing Cosine Similarity. Retrieved from: https://www.learndatasci.com/glossary/cosine-similarity/

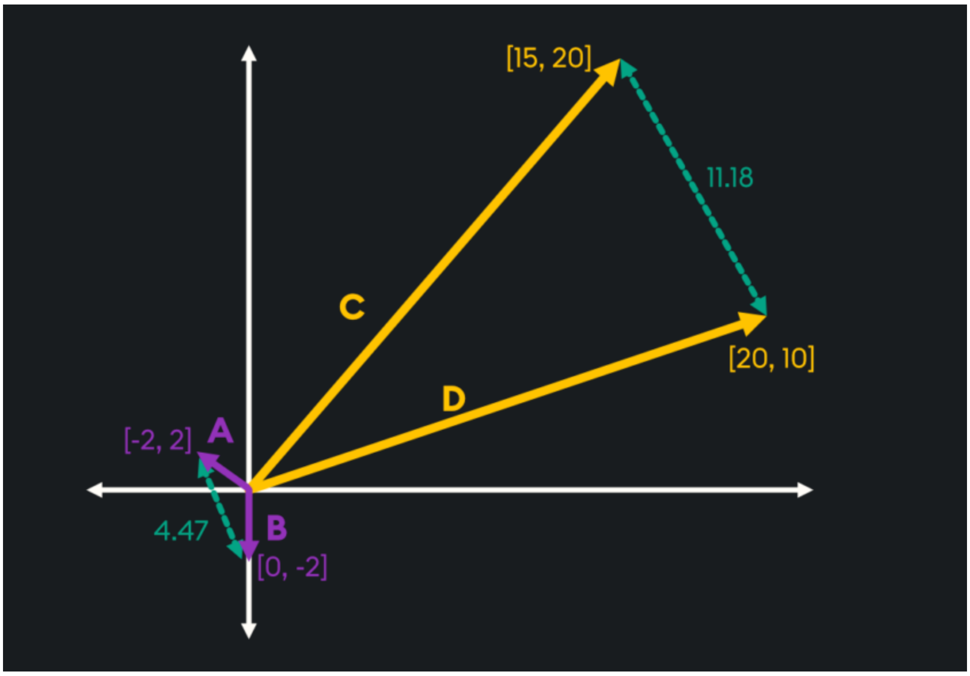

Figure 7: Image for Representing Euclidean Similarity. Retrieved from: https://kdb.ai/learning-hub/articles/methods-of-similarity/

As we can visualize from figure 6 we can understand that the Cosine Similarity looks at the 'similarity' of vectors as directions. Meanwhile in figure 7 we understand that Euclidean Similarity looks at it out of how similar each vector is in terms of *magnitude*.

As a recommendation for users when utilizing the code we developed, I recommend that when looking up queries of **small sizes** they utilize the Cosine Similarity. Sizes are relative dependent on the average size of the documents in the folder to be indexed.

4.3 Vectorizer Hierarchy

The Vectorizer Hierarchy is the class of families in charge of the overall indexing of the folders, it is in charge of creating the abstractions of how the files are interpret in memory. It's named **vectorizer** due to its ability of vectorizing what is input to it. For the development of this project we decided to create our own indexing algorithm. The first algorithm we used is the **TF-IDF** algorithm.

4.3.1 Term Frequency - Inverse Document Frequency (TF-IDF) Algorithm

The TFIDF child class is in charge of vectorizing each document using the three formulas. The **Term Frequency (TF)** manages to get the frequency in which each term appears in a document.

$$\text{TF}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Where, the variables represent:

- $f_{t,d}$ is the frequency of the term t in the document d.
- $\sum_{t' \in d} f_{t',d}$ is the sum of frequencies of all terms in the document d.

Figure 8: Formula used for Term Frequency calculation of a single term in a document

$$\text{IDF}(t,D) = \log\left(\frac{N}{|\{d \in D : t \in d\}|}\right)$$

Where, the terms represent:

- N is the total number of documents in the corpus D,
- $|\{d \in D : t \in d\}|$ is the number of documents in which the term t appears.

Figure 9: Formula used for Calculating the Inverse Document Frequency of a single term

$$\text{TF-IDF}(t,d,D) = \text{TF}(t,d) \times \text{IDF}(t,D)$$

Figure 10: Formula used for Calculating the TFIDF of a document considering the TF and IDF

For the development of this code we used **TF-IDF** algorithm as an interpretation of these formulas in code. Considering that each formula returns a vector that can be handled by the other classes of the code. This is one of the abstractions utilized for the vectorization of the files.

This algorithm was what was expected of us to develop initially for the creation of the **Indexer and Search Engine**, one of the initial requirements.

4.3.2 Custom Algorithm: Normalized BoW

This algorithm is the other version of the indexer we had to develop, we used a custom algorithm which mixes concepts of the **Bag of Words**[2] algorithm for indexing, which is just considering each file as a vector which contains the count of each term in a document. We decided on Normalizing this vector for lenghts to be reduced and for an easier manipulation. Thus, after indexing the files it is easier to handle them since the individual components of the vector now represent the overall relevance of a term in a file. It uses a similar formula to the one explained in figure 8. This algorithm is a lot less computationally intensive and still maintains appropiate behavior when being handled by the **Search Engine**.

4.4 Indexer Class

The indexer class is in charge of linking the logic of all class hierarchies and the methods that where passed to us as requirements of this code. To create this we must look back at the figure 3 which represent the available levels of functionality of the code. In this class all the logic that ends up retrieving the

[2] A quick explanation of the Bag of Words model is found in: `https://www.ibm.com/topics/bag-of-words`

output of the program is placed here. It allows for the **GUI** and the **CLI** to use nearly identical interpretations.

5 Unit Test

The **Unit Test** section of the code is in charge of checking that all concrete methods, have the expected behavior. We managed to test all the functionality of the code. Yet, certain tests might be missing out due to our lack of experience in developing tests. Even, though we did manage to be properly generate Unit Test for the Document, Similarity, Vectorizer and Indexer hierarchy. The Unit Testing was done in a straightforward manner, where mock documents and instances of classes where generated. For a further understanding please look at the Folder containing the `UnitTests` for all classes.

6 References