Semi-Structured Text Document Search using Vector Space Model

Report Project One

Anthony Semaan – Issa Makki

Abstract— This report documents the first project in the IDPA course. It focuses on comparing semi-structured text documents by applying Vector Space Model (VSM) similarity measures. The project serves two main purposes: determining the similarity between two text files and searching for relevant documents in a database based on user queries. Users can submit queries in the form of either semi-structured text documents or regular text. The application then returns relevant documents based on the results of the similarity computations.

Table of Contents

1 - Introduction 1
2 – Documents Comparison and Similarity Computation 2
  2.1 – Semi-Structured Text Document Format 2
  2.2 – Text Preprocessing 2
  2.3 – Procedure 3
    2.3.1 – Content only 3
    2.3.2 – Content and Structure 3
    2.3.3 – Weight Strategy 3
    2.3.4 – Similarity Strategy 3
3 – Search in a Collection based on Query 4
  3.1 – Collection 4
  3.2 – Indexing 4
  3.3 – Search using Keywords 4
  3.4 – Search using Semi-Structured Document 4
4 – Frontend 5
6 – Conclusion 5
7 – References 5
8 – Appendix 6
  8.1 – Example data frame weights when computing Content Only VSM Similarity 6
  8.2 – Example data frame weights when computing Content and Structure VSM Similarity 6

1 - Introduction

This project involves the development of a search tool for comparing similarities in both semi-structured and flat text using the Vector Space Model. In the initial phase, users can input two text documents and gauge their similarity using selected metrics for similarity and weight computation. For semi-structured documents, a preprocessing step transforms them into XML, and similarity is computed using a learned XML-based approach. Conversely, for non-structured (free text) input, a traditional information retrieval (IR) approach is employed for comparison and similarity computation.

The second phase allows users to input a query, which can be either a semi-structured text document or flat text. Additionally, users have the flexibility to select the weight strategy model and similarity measure based on the type of query submitted. The tool then ranks documents from the dataset according to their similarity results with the query. To enhance efficiency, an optional indexing feature is integrated into the project, allowing programmers to choose its inclusion or omission. The implementation of the backend side is carried out using the Python programming language, while the frontend has been developed using react and JavaScript.

2 – Documents Comparison and Similarity Computation

As previously stated, the application supports the comparison of semi-structured documents through the application of the VSM similarity measure. The files' content may adhere to a predefined document structure in the case of semi-structured documents, or it may be free text where no specific structure is considered during the similarity computation. In the following sections, we will outline the structure of semi-structured documents and emphasize various parameters associated with them.

2.1 – Semi-Structured Text Document Format

The semi-structure format used in our analysis follows the following format:

```
                          Text Document Search Tool
1. Title
Text document Search Tool.
2. Objective
The main goal of this project is to implement a text document search tool,
which searches for documents sharing structure-and-content similarity using a
Vector Space Model (VSM-based) approach.
2.1. Guidelines
Null
2.2. Framework
It will be developed in the context of the IDPA course.
3. Background and Motivations
Text-rich documents are becoming increasingly available to describe all kinds
of data, from academic reports and medical reports.
```

The document should start with a title, followed by sections numbered as 1., 2., 3., … Each section should have title. Each section can have a content section, or subsections like 2.1. and 2.2. subsections in the above figure. While the titles cannot be more than one line, the content sections can expand to multiple lines.

The structure of the generated XML file is as follows:

```
<xml>
  <document>
     <title>
        <dimension> … </dimension>
        <dimension> … </dimension>
     </title>
     <sections>
        <section>
           <number> … </number>
           <title>
              <dimension> … </dimension>
              <dimension> … </dimension>
           </title>
           <content>
               <dimension> … </dimension>
               <dimension> … </dimension>
           </content>
           <subsections>
              <subsection>
                 <number> … </number>
                 <title>
                    <dimension> … </dimension>
                    <dimension> … </dimension>
                 </title>
                 <content>
                    <dimension> … </dimension>
                    <dimension> … </dimension>
              </content>
             </subsection>
          </subsections>
       </section>
    </document>
</xml>
```

2.2 – Text Preprocessing

To clean the input text documents, the following steps are performed:

1. **Transform characters to lowercase:** Convert all characters in the text to lowercase.

2. **Tokenize using nltk.word_tokenize**: Break the text into individual words or tokens using the NLTK library's word tokenizer.
3. **Remove stop words and non-alphabetical values**: Eliminate common words (stop words) and non-alphabetical characters from the text.
4. **Lemmatize using nltk.stem.WordNetLemmatizer:** Reduce words to their base or dictionary form (lemma) considering the context and part of speech. For example, "running" becomes "run," and "better" becomes "good."
5. **Stem using nltk.stem.PorterStemmer**: Shorten words to their base form (stem) by removing prefixes or suffixes. This is a more aggressive and rule-based approach compared to lemmatization, and it may not always result in valid words. For example, "running" might be stemmed to "run," but "better" could be stemmed to "better" itself.

2.3 – Procedure

2.3.1 – Content only

In scenarios where the "content only" option is activated, the document's structure is disregarded. Consequently, the initial action involves applying the document cleaning step using the previously mentioned approach. Following this, the vector weight of each document is calculated using either Term Frequency (TF) or Term Frequency-Inverse Document Frequency (TF-IDF). Because similarity computation demands that both vectors have the same dimensions, the OR (Overlapping Ratio) of both vectors is computed. A data frame is then generated, with dimensions as rows and vector one and two as columns. Check Appendix 8.1 for reference. Leveraging these vectors, the similarity measure is subsequently computed.

2.3.2 – Content and Structure

In case the input files are semi-structured and follows the predefined format, content and structure similarity computation can be performed. Thus, the first step requires transforming the semi-structured document to XML using the approach mentioned above. Second, we compute the Term Context of each of the elements inside the <dimension> tag. Next, we compute the vector weights based on the chosen strategy (either TF or TF-IDF). Check Appendix 8.2 for reference. Finally, based on the generated vectors, Similarity is computed.

2.3.3 – Weight Strategy

While TF counts the number of occurrences of a specific term, the IDF formula used is the following:

$$\mathrm{IDF}(t_i, C) = \log \frac{N}{\mathrm{DF}(t_i, C)} \quad \in [0, \infty[$$

To compute TF-IDF, a simple multiplication between both TF and IDF is performed.

$$w_D(t_i) = \mathrm{TF}(t_i, D) \times \mathrm{IDF}(t_i, C)$$

2.3.4 – Similarity Strategy

The application provides numerous vector-based similarity measures to choose from. These similarity measures were discussed in chapter 6 of the course:

Similarity Measure	Formula
Dice	$\mathrm{Sim}_{Dice}(\vec{A},\vec{B}) = \frac{2 \times \vec{A} \times \vec{B}}{\lvert\vec{A}\rvert^2 + \lvert\vec{B}\rvert^2} \quad \in [0,1]$
Cosine	$\mathrm{Sim}_{Cosine}(\vec{A},\vec{B}) = \frac{\sum_{i=1\ldots N} A_i \times B_i}{\sqrt{\sum_{i=1\ldots N} A_i^2 \times \sum_{i=1\ldots N} B_i^2}} \quad \in [-1,1]$

Pearson Correlation Coefficient	$\mathbf{Sim}_{PCC}(\vec{\mathbf{A}},\vec{\mathbf{B}}) = \frac{\sum_{i=1...N} (\mathbf{A}_i - \bar{\mathbf{A}}) \times (\mathbf{B}_i - \bar{\mathbf{B}})}{\sqrt{\sum_{i=1...N} (\mathbf{A}_i - \bar{\mathbf{A}})^2 \times \sum_{i=1...N} (\mathbf{B}_i - \bar{\mathbf{B}})^2}} \quad \in [-1,1]$
Manhattan	$\mathbf{Sim}_{Manh}(\vec{\mathbf{A}},\vec{\mathbf{B}}) = \frac{1}{1+\mathbf{Dist}_{Manh}(\vec{\mathbf{A}},\vec{\mathbf{B}})} \quad \in \ ]0,1]$ $\Rightarrow \text{having} \quad \mathbf{Dist}_{Manh}(\vec{\mathbf{A}},\vec{\mathbf{B}}) = \sum_{i=1...N} \lvert \mathbf{A}_i - \mathbf{B}_i \rvert \quad \in [0,+\infty[$
Euclidian	$\mathbf{Sim}_{Euclid}(\vec{\mathbf{A}},\vec{\mathbf{B}}) = \frac{1}{1+\mathbf{Dist}_{Euclid}(\vec{\mathbf{A}},\vec{\mathbf{B}})} \quad \in \ ]0,1]$ $\Rightarrow \text{having} \quad \mathbf{Dist}_{Euclid}(\vec{\mathbf{A}},\vec{\mathbf{B}}) = \sqrt{\sum_{i=1...N} (\mathbf{A}_i - \mathbf{B}_i)^2} \quad \in [0,+\infty[$

Table 1 - Similarity Measures

3 – Search in a Collection based on Query

3.1 – Collection

As the application accommodates the comparison of both free text and semi-structured documents, a comprehensive collection encompassing both categories has been curated. For free text files, the dataset comprises poems sourced from Kaggle. Each file within this dataset corresponds to an individual poem. Conversely, semi-structured files adhering to a predefined structure have been specifically developed. To ensure the accurate computation of semi-structured documents using the content and structure approach, these files have been divided between two directories. This organization aims to maintain a clear distinction and facilitate effective processing for each document type.

3.2 – Indexing

To expedite the search process, an indexing feature can be employed. To activate this functionality, an indexing table utilizing the reverse indexing approach is precomputed and generated. In this approach, keys consist of cleaned words present in all documents within the collection, and the corresponding values are the paths of files containing that key. When searching for files, a loop is initiated over each word within the query. Relevant files—those containing at least one common dimension between the query and the file—are then selected. This optimized approach allows the information retrieval (IR) process to iterate exclusively over the identified relevant files, rather than the entire collection.

3.3 – Search using Keywords

The application's user interface allows users to initiate searches through input queries, which are treated as free text. Consequently, the VSM similarity is computed utilizing the approach detailed in section 2.3.1. In this scenario, the input query vector weights are computed, and the similarity between this vector and the vectors of relevant files is determined. Subsequently, leveraging the obtained similarity measures, the most relevant files are ranked using the K-Nearest Neighbor (KNN) algorithm. This process ensures an effective and ordered presentation of files based on their relevance to the input query.

3.4 – Search using Semi-Structured Document

Likewise, the user interface facilitates querying with semi-structured documents. When a user inputs a semi-structured document, the initial step involves its conversion into XML format. Subsequently, the VSM similarity measure, as mentioned in section 2.3.2, is applied. As previously discussed, to identify the most relevant documents, the(KNN algorithm is executed on the similarity results obtained between the input document and all documents shortlisted from the indexing table. This method ensures the retrieval and ranking of the most contextually relevant documents for a given semi-structured query.

4 – Frontend

To build our user interface we used Visual Studio Code as our IDE. The user interface was built using React JS and TypeScript:

- The application runs on port 5173.
- We utilized the Material UI library for our front-end components.
- We utilized the axios library for our backend integration.
- The user interface consists of a home page and two main pages called Difference and Search.
- The Difference page below allows users to input two text files and compare them according to the parameters provided. The output is the similarity ratio in addition to the time needed to find the ratio.
- The Search page allows users to search by writing a text query or uploading a file and choosing the desired parameters:
  - We utilized the file-saver library to allow users to download a file from one of their search results.
  - The number of results will depend on the KNN parameter or the number of files with positive similarity.

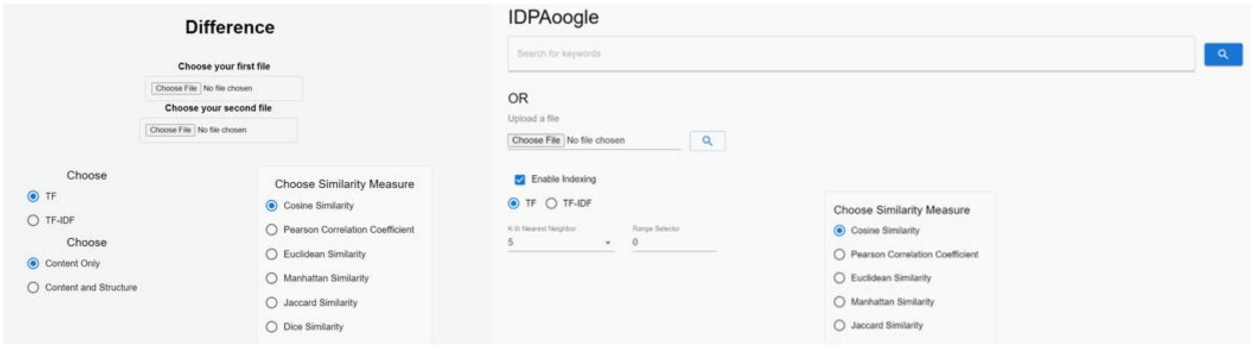

Figure 1 - User Interface for Comparison and Search

6 – Conclusion

In summary, this project has effectively implemented the Vector Space Model to create a versatile text comparison and retrieval tool. Its two-phase approach accommodates both semi-structured and flat text documents, providing users with a comprehensive solution for document-document and document-query similarity. The incorporation of a learned XML-based approach, optional indexing, and a thoughtful choice of technologies enhance the tool's adaptability and efficiency, making it an asset for information retrieval in diverse contexts.

7 – References

[1] K. Pykes, “Vector space models,” Medium, https://towardsdatascience.com/vector-space-models-48b42a15d86d (accessed Nov. 13, 2023).

[2] Michaelarman, “Poems dataset (NLP),” Kaggle, https://www.kaggle.com/datasets/michaelarman/poemsdataset (accessed Nov. 13, 2023).

8 – Appendix

8.1 – Example data frame weights when computing Content Only VSM Similarity

	Vector1	***Vector2***
dismal	0	1
hello	1	2
silk	0	1
shine	0	2
cover	0	1
save	1	0
document	8	0
rule	0	1
charcoal	0	1
bodi	1	0
silver	0	1
scope	1	0

8.2 – Example data frame weights when computing Content and Structure VSM Similarity

	Vector 1	***Vector 2***
use,xml/document/sections/section/subsections/subsection/title/dimension	0	1
paragraph,xml/document/sections/section/content/dimension	1	0
user,xml/document/sections/section/content/dimension	1	0
mission,xml/document/sections/section/title/dimension	0	1
organ,xml/document/sections/section/content/dimension	0	1
motiv,xml/document/sections/section/content/dimension	1	0
search,xml/document/sections/section/content/dimension	0	1
optim,xml/document/sections/section/subsections/subsection/content/dimension	0	1
quick,xml/document/sections/section/content/dimension	0	1
manag,xml/document/sections/section/content/dimension	0	1
document,xml/document/sections/section/subsections/subsection/content/dimension	0	1
idea,xml/document/sections/section/content/dimension	1	0