THE UNIVERSITY OF KANSAS

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

RESUME SCREENING USING VECTOR SPACE MODEL

EECS 767 – Information Retrieval

Instructor: Bo Luo

Spring 2023

Project Report

Team members:

Student Name – Chaitanya Addepalli Student Id – 3127642

Student Name – Srijanya Chetikaneni Student Id – 3128038

Student Name – Sree Lasya Thatigutla Student Id – 3128034

I. INTRODUCTION

In this project, we have proposed a simple yet powerful tool that helps screen resumes for recruiters. Technology has revolutionized the way of our life, and the recruitment process is also catching up with the latest trends. With the increase in internet connectivity, there has been a change in the recruitment process in every organization. With the help of online job postings, recruiters are able to draw more candidates for their openings. Though e-recruitment has brought convenience to both recruiters and candidates, it also brings challenges. Today, organizations receive huge applications daily, and reviewing them quickly is difficult.

The main challenge is to review the resumes of all the applicants. It is tedious for recruiters to manually screen every resume and match the candidate's skills, abilities, and experience with the job description. With only a limited number of vacancies and a large pool of applicants, it is difficult for recruiters to make better decisions and choose suitable candidates. Due to the high volume of applications, recruiters never know if they have hired the best candidate unless they screen through the entire pool of applications. The recruiters do not have time for this; in most cases, the best candidate's profile is not even reviewed.

In order to improve the effectiveness of resume shortlisting, it is essential to explore more efficient methods. Our proposed system fills this gap by retrieving qualified candidate resumes by matching the keywords with the job description.

II. ARCHITECTURE OVERVIEW

The vector space model is a simple yet most-used technique in information retrieval and content-based recommendation systems [1]. We proposed a system that uses the vector space model and cosine similarity to retrieve resumes for a given job description. Our model assists in accelerating recruitment by simplifying the screening process.

The architecture of our model is divided into two parts: the user interface (UI) and the backend component. We have developed both of these using Python. On a high level, our system works as follows:

Whenever a user enters a query in the UI, our model begins reading the text from the resumes in the dataset. The text is then preprocessed and tokenized. The tokens are then indexed using the reverse index. Document vectors are created by calculating the TF-IDF weights. We have built a dynamic model that stores the index and document vectors and updates them whenever new documents are added to the database. The query vector is also created employing the same process used for document vectors. The vectors are then normalized for calculating cosine similarity. Finally, the resumes are ranked using the similarity values, and the top candidates' profiles are returned along with their similarity values on the UI.

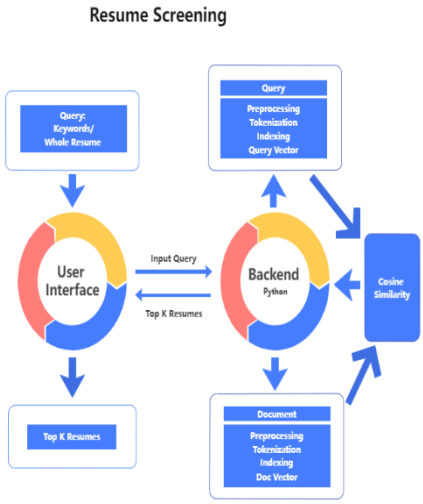

Figure 1 – Application Architecture

III. IMPLEMENTATION

3.1. Reading the dataset:

Our dataset consists of 2484 resumes from various domains. These resumes have been taken from Kaggle, an online machine learning and data science community. All these documents are in PDF format. Our program reads these documents from the stored repository in a sequential manner. We have used Python's PyPDF2 library, which allows the extraction of text from all the pages of a PDF file. Our program stores the text corresponding to each document separately, as a string, for further processing.

3.2. Raw Text Processing:

Text processing involves a series of steps on the raw text extracted from all the documents and the query. The first step in this process is tokenization. In this step, the text is converted to lowercase and cleaned by removing all the punctuations, numbers, and non-alphabetic characters. Then the text is split into tokens. In our program's first version, we split the text into tokens based on the white space delimiter. This approach resulted in many compound words as tokens. It is probably because of the nature of the dataset. The problem with this is that these compound words are considered as new tokens, while the individual words also have separate tokens. This increases the size of the index and the processing time as well. We have imported and used a pre-trained language model called "wordninja" to overcome this problem. This model effectively splits sentences into tokens even if no delimiters separate the words. This helped us reduce the number of unique tokens in the index.

The next step implemented is the removal of stopwords. Stopwords are words that appear frequently in many different documents or many sections within the same document. These words usually carry little information about the part of the text they belong to. Stopwords increase the size of the index and the document vectors, thereby increasing the overall processing time. Removing these stopwords can improve the signal-to-noise ratio in the unstructured text, thereby increasing the significance of other important terms that describe the document [3]. We have used the standard set of stopwords provided in the Natural Language Tool Kit (NLTK) package. Some of these stopwords are: "are", " be", " the", and" but". Our program removes all these stopwords from the tokens generated during tokenization.

The last step in our text preprocessing is stemming. Stemming is performed by removing the suffixes of a given word, thereby reducing it to its root form. This root word does not always need to have a linguistic meaning. Any given text can have multiple words with the same roots. These words usually have similar meanings. So, reducing these words to their root forms will improve the performance of our IR system as the number of unique terms in our system reduces. Porter's Stemmer algorithm is a standard method used for stemming in literature. Each word is subjected to a sequence of 5 steps of suffix removal, following specific rules in each step [2]. We have used the Porter Stemmer implementation from Python's NLTK package in our program. The duplicate roots are removed after stemming and stored for indexing.

3.3. Indexing

We have used the inverted index in our IR system. For the 2484 resumes, a total of 17747 unique tokens. All the unique preprocessed tokens are sequentially added to the index, and for each token, the ids of documents in which these tokens are present, and the term frequency (TF) of that token in that document are stored. The index format is {token: {doc_id_1 : TF_1, doc_id_2 : TF_2, ….doc_id_n : TF_n}}.

We have built a dynamic program that stores all the document ids and the generated index to text files in the first run and then updates these files in consecutive runs. The program looks for new files in the resume repository in every consecutive run. If there are no new files, the index does not need to be updated again. The program reads the index stored previously and proceeds further. However, if there are any new files, the text corresponding to these files is extracted and preprocessed, and then the tokens are generated, as discussed in the previous sections. The index stored in the previous run is also extracted and updated for all the tokens generated from the newly added documents. If a token already exists in the index, its data is updated by adding the new document ids and the term frequencies of these tokens in the corresponding documents. If the token is not in the index, it is inserted, and the corresponding data is updated.

If there is any update to the index in the current run, it is written to the stored index file. Else the index file remains the same for the future run.

3.4. TF-IDF weighting

Now that we have the index for all the tokens, we assign weight to each token based on its importance and relevance to the corresponding document. For this, we implemented TF-IDF weighting, the most used technique for weighting the terms in the literature today. TF (term frequency) of a term in a given document is defined as the number of times the term appears in that document. A high TF indicates that the document is more relevant to the term. IDF captures the importance of a term. If a term is present in every document or the majority of the documents, it carries little information and contributes little to retrieving the relevant data. So, it is intuitive to assign low weights to such terms.

On the other hand, rare terms can facilitate fetching the documents relevant to the query. So, these terms should be assigned a higher weight. Hence, we use document frequency to calculate the IDF of a term. A term with high document frequency should have low weight and vice versa. So, IDF is the inverse document frequency of the term. We calculate the TF-IDF weight as:

$$w_{t,d} = tf_{t,d} * idf_t \text{ [5]}$$

$$idf_t = \log_{10} N/df_t \text{ [5]}$$

Where,

$tf_{t,d}$ stands for term frequency of a term ‘t’ in a document ‘d’
df_t stands for document frequency of a term ‘t’
idf_t stands for inverse document frequency of a term ‘t’

Logarithm is applied in the IDF calculation to dampen the effect of IDF in the weights.

3.5. Document vectors

We used the bag of words model to represent each document as an entity. Each document is considered a bag of words and is represented as a vector for further computations. The dimensionality of these vectors is equal to the number of unique terms collected from all the documents in our database. All the unique terms are first sorted, and then for each word in this collection, a document vector holds the corresponding TF-IDF weights of the terms in this document in the sorted order. The weight is set to zero if the document does not contain a term. The document vectors are generated this way for each resume in our collection. These vectors are stored in a text file to avoid calculations for the same resumes again and again in the future. In every run, these vectors are read from the database and updated if new resumes are found. The new document vectors are created for every new resume added to the repository. These vectors will have an increased dimension as new unique terms are added to our collection with every new resume. The dimension of older document vectors is also updated by inserting zeroes for every new unique term in the corresponding positions. Now that we have the vectors for all the resumes, we update them in the database.

The document vectors are then normalized to unit vectors by dividing each value with the square root of the sum of squares of all the values in the vector. The normalized vectors are then stored in a separate file in the database. Storing these normalized

vectors reduces the time taken in the resume retrieval for future runs. If the resume database remains the same, that is, if no new resumes are added, the document vectors also remain the same. In this case, instead of reading the non-normalized document vectors and normalizing them again in the run time, our program reads the stored normalized vectors and cuts processing time.

3.6. Query processing:

A query is the input data obtained from the end user. Our program takes the input through an interactive user interface developed in Python. More details about the user interface are discussed in the section 3.8. The input query fed to our program can be a job description or an entire resume. The program then reads the input as a string and does the preprocessing steps like tokenization, stopword removal, and stemming using the porter stemmer algorithm. All these steps are applied to the query in the same way as the documents are processed to maintain uniformity. Otherwise, there will be a mismatch between the tokens generated from the query and the unique token collection generated from the documents. Once the processed tokens are available from the query, the query vectors are generated. Query vectors also have the same dimension as that of document vectors. If any token generated from the query is not available in our collection, it is discarded. If the token is present in the collection, its TF is taken as '1' irrespective of how often it appears in the query. IDF is calculated using the formula discussed in section 3.4. The query vector is thereby generated by calculating the TF-IDF weights for each term. This vector is then normalized to a unit vector.

3.7. Cosine similarity for ranking

Now that we have the documents and queries represented as vectors in the vector space, we used the concept of cosine similarity for ranking. The similarity between two vectors is measured by the angle between them. Cosine similarity is calculated by taking the dot product of the unit vectors. The resumes are ranked in the decreasing order of the cosine of the angle between the query vector and a resume. The vector with the highest cosine similarity value is closest to the query vector. Hence, the corresponding resume is the most similar to the job description given. This way, our program ranks the resumes against the input job description and displays the list of top resumes in the UI.

3.8. User Interface

We have built a simple and interactive user interface in Python using the 'Tkinter' library. Our UI page consists of a search box that accepts input queries from the user and a click button to start processing the input query. It also has a results box that displays the top resumes and the corresponding similarity values generated based on the input query. The UI page takes the user input, and once the user clicks on the "Find Resumes" button, it passes the input query to the backend program. Finally, it fetches the output from the backend and displays the results to the user. Below is the snapshot of our UI displaying the results.

Figure 2 – User Interface

IV. EVALUATION AND RESULTS

4.1. Model validation

To validate our model, we have generated a test data sample of four documents, each consisting of a single sentence. A query consisting of words from these documents is passed, and the program is run. The index, document vectors, normalized document vectors, and the similarity values obtained have been compared with the manual calculations. We have found an exact match between our calculations and the values generated by our program. Through this simple yet powerful test, we ensured that our model was working as we intended.

4.2. Performance Evaluation

To evaluate the performance of our model, we have designed a test with a sample size of 500 resumes. These documents produced 9565 unique terms after tokenization and the removal of stopwords. Our model took 162.25 seconds in the first run to generate the results. This includes reading the content of the documents, processing the data, and generating and storing the index, document vectors, and normalized document vectors. We performed a second run for the same test dataset, and it took 2.78 seconds for our model to display the results. The pie charts in Figure 3 and Figure 4 show the percentage of processing time for various tasks in both the first and second runs.

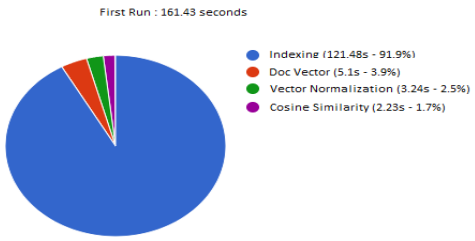

Figure 3 – First Run Results

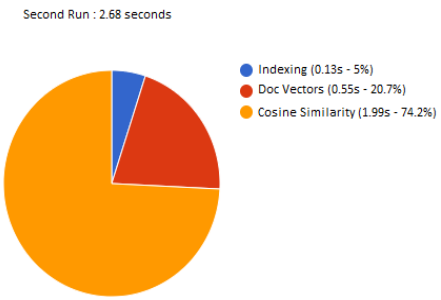

Figure 4 - Second Run Results

It can be observed that the run time is significantly cut down by storing the index and the normalized document vectors. Moreover, in the second run, the program does not have to reread the documents (as the index and document vectors are already available), which cuts the majority of run time.

4.2. Accuracy Evaluation

The accuracy of our model has been evaluated by testing it with four different input queries. Each query is a job description from the following domains: Information Technology, Human Resources, Automobiles, and Business Development. Each of these queries perfectly fetched the top resumes from the corresponding domains. Our dataset has resumes from 24 domains, and our model successfully retrieved the resumes that matched the job description. The snapshots from each of our tests are shown in figures 5-8.

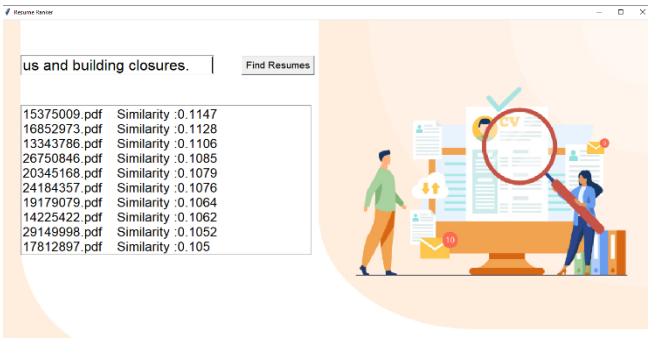

Figure 5 – HR Domain Results

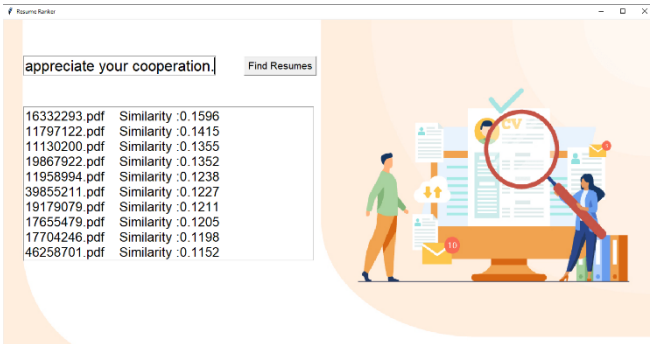

Figure 6 – Automobile Domain Results

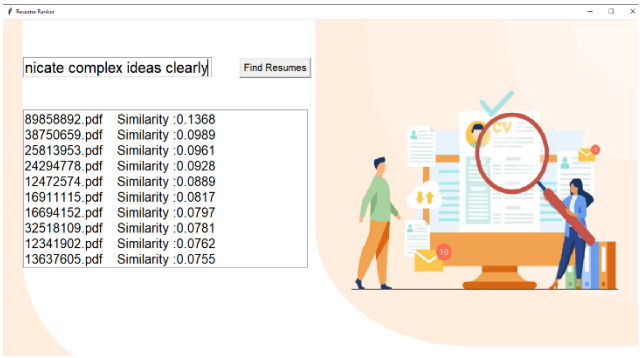

Figure 7 – Information Technology Domain Results

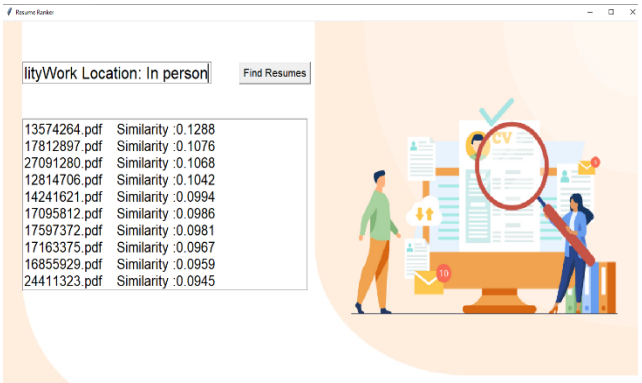

Figure 8 – Business Development Domain Results

The similarity values for the top documents in each test are in the order of 0.1. This might look higher for a dataset with 2484 resumes. Based on our analysis, this is mainly because of the diversity in the dataset. As the resumes are from 24 domains, each domain holds around 100 resumes. This results in a comparatively higher IDF for a frequent term like 'programming' in the Information Technology domain. This term does not even exist in most resumes from other domains. Hence, for any given job description, most of its terms exist in only around 100 resumes which is a comparatively smaller number. A higher IDF value increases the TF-IDF weights of these terms. This, in turn, results in a comparatively higher similarity value than what we usually expect for this database.

V. LIMITATIONS & FUTURE WORK

Our model serves as a primary screening system for recruiters. Our model has certain limitations, and hence there is room for improvement.

The current version of our model is designed to read only the resumes in PDF format. As a future work, we plan to upgrade the current version to make it versatile for reading resumes in other formats like HTML, JSON, and DOCX.

We also plan to build a dynamic web crawler that automatically crawls and collects resumes from various online job search portals. Then we will host our program on a server

to keep it live all the time and update the index and document vectors as and when a new resume is added to the database.

Another limitation of our model is that it does not consider factors like a candidate's experience, educational qualifications, or personal attributes. We plan to revise our model by calculating a score that accounts for the above attributes. This score can be combined with the cosine similarity score, and resume ranking can be done based on the combined score.

We further plan to extend our model by implementing a clustering mechanism using K-means clustering. This will enable the recruiters to segregate the resumes based on their domain.

VI. Conclusion

Our project aims to simplify the current employee recruitment process and cut down a significant part of the manual efforts for the recruiters. This saves time and reduces the costs incurred by an organization in the hiring process. In this project, we presented a resume screening model developed by applying information retrieval techniques. Our model can retrieve the required number of resumes that match the given job description from a large pool of resumes. If this needs to be done manually, it requires efforts from many people, and also, there is a high chance that a qualified resume is not evaluated due to time constraints on the hiring process. Using our model, the recruiters can screen a large volume of resumes within seconds and generate a subset of resumes with suitable candidate profiles. This avoids going through many unwanted profiles and ensures that suitable candidates' profiles are not missed from being evaluated.

Our model reads all the resumes from the database, pre-processes the text, and generates tokens. Stopwords are removed from these tokens to improve model efficiency. Tokens are stemmed to their roots to maintain a single token for similar words. The vector space model has been incorporated to represent the documents and the query as vectors, and the concept of cosine similarity is applied to retrieve and rank qualified resumes. Our model serves as a basic version, and there is scope for improvement. We plan to incorporate various aspects discussed in section V in the coming days and develop a more powerful version of our model.

Our source code is available at [srijanyarao/Information-retrival-project-group15-KU (github.com)]

VII. References

[1]C. Daryani, G. S. Chhabra, H. Patel, I. K. Chhabra, and R. Patel, "AN AUTOMATED RESUME SCREENING SYSTEM USING NATURAL LANGUAGE PROCESSING AND SIMILARITY," *ETHICS AND INFORMATION TECHNOLOGY*, Jan. 2020, doi: https://doi.org/10.26480/etit.02.2020.99.103.

[2]"Resume Dataset," *www.kaggle.com*. https://www.kaggle.com/datasets/snehaanbhawal/resume-dataset (accessed May 12, 2023).

[3]S. Sarica and J. Luo, "Stopwords in Technical Language Processing," *PLOS ONE*, vol. 16, no. 8, p. e0254937, Aug. 2021, doi: https://doi.org/10.1371/journal.pone.0254937.

[4]"Porter Stemming Algorithm," *tartarus.org*. https://tartarus.org/martin/PorterStemmer/

[5]"Contents," *nlp.stanford.edu*. https://nlp.stanford.edu/IR-book/html/htmledition/contents-1.html (accessed May 12, 2023).