A picture containing graphical user interface

Description automatically generated

ISSS609 Project Report

**Text Summarization and**

**Similarity Matching of Plots**

**For Movie Recommendations**

***Group 2***

*Gerry CHNG Kian Woon*

*Karishma YADAV*

*Mayurapriyann ARULMOZHI BASKARAN*

*SYED Ahmad Zaki Bin Syed Sakaf Al-Attas*

*Vertika PODDAR*

***8th November 2021***

Contents

[1. Introduction 3](#_Toc87305669)

[2. Background and Motivation 3](#_Toc87305670)

[3. Understanding The Data Used 4](#_Toc87305671)

[4. Data Cleaning Considerations 4](#_Toc87305672)

[5. Overall Flow 5](#_Toc87305673)

[6. Solution Overview 6](#_Toc87305674)

[7. Solution Details 8](#_Toc87305675)

[8. Results and Analyses 9](#_Toc87305676)

[9. Discussions and Gap Analysis 14](#_Toc87305677)

[10. Conclusion 16](#_Toc87305678)

[11. Project Experiences/Reflections 17](#_Toc87305679)

[12. References 19](#_Toc87305680)

# Introduction

We are living in an era of information overload that is surfacing in all aspects of our lives. With the quantum of data at our disposal, the challenge is to identify the specific information / topic that is of relevance to us at that point in time. Paradoxically, the less we need to read, the more we can read.

For a world that is leaping towards automation, the need is to have models that can assist in search, selection, and summarization without compromising on quantity or quality.

Our project is an attempt to contribute towards simplifying this process of search, selection, and summarization. This study aims to deploy models that can facilitate:

1. Text Summarization
2. Generate Recommendations based on known preferences

To give context to these models, a database of movie plots was targeted. The advent of over-the-top (OTT) platforms has eased and enhanced movie-watching activities. However, in the absence of such recommendation, summarization, and ranking algorithms, viewers will struggle to find movies that they *‘Like’* due to the sheer volume of information.

This study provides a solution to this challenge by designing an independent summarization and recommendation engine.

# Background and Motivation

We are often presented with a wide range of information that can be consumed if we have the time to sift through them. While basic search functions can help us zoom and narrow down to key words and specific metadata that is of relevance to us, it takes a very narrow focus as the ranking is simply based on the presence of such metadata (for binary cases) or by the number of occurrences of such terms.

On a popular platform, we are still left with a very lengthy search result to sift through. This might be the case for researchers who struggle to keep up with new journals published daily. For example, the Computer Science Repository in arXiv averages around 400 submissions of new and updated journals daily.

For this project, the use case of Movie Plots was considered, though it can be easily extended to other OTT platforms providing similar structures of documents structured by preferences (journal sites, news sites, online chats, and even movie channels by running another layer of speech-to-text conversions).

Reading a lengthy movie summary can be a very time consuming and tedious task before deciding on a movie to watch from a streaming service. Viewers would rather prefer to skip the summary if it were too lengthy and pick the movie solely based on the movie titles. A movie title gives the first impression of the movie but can only hint on what the movie is about. So, including a concise summary of the movie along with its title provides a gist of the movie’s storyline. This would reduce the workload of the reader and speed up the decision-making process.

The decision-making process would be sped up even further, if there exist an algorithm that keeps track of our known preferences, pre-read these summaries for the viewer, and recommend a ranked list of movies based on movies that these same viewers have liked before.

This study thus seeks to create the scaffold of this algorithm in making recommendations based on user preferences, and further summarizing the returned results for quick manual scans by the user.

# Understanding The Data Used

This study notes the difficulty in obtaining readily available and detailed plot summaries, without major defects and in a working format. Thus, to simplify the data gathering process, this study pulls from Carnegie Mellon University’s Movie Summary Corpus (<http://www.cs.cmu.edu/~ark/personas/>). This data volume was extracted from Freebase, and compiled by Bamman, O’Connor, and Smith (2013) as part of a study titled “Learning latent personas of film characters.” While this corpus contains various datasets relating to movies, this study utilizes only the movie metadata and the plot summaries dataset. The movie metadata contain 81,741 movie details such as its title, release dates, language, genre, and others, while the plot summaries contain 42,306 detailed movie summaries written in English.

While the plot summaries remain central to this study, not all fields of the movie metadata would be included. However, fields such as movie release date, movie languages, movie genre and a few others were kept as part of subsequent data cleaning and evaluation process to support future work.

The plot summaries themselves do not have any inherent defects such as misspellings or non-English words that may impact our analysis. On the other hand, this study notes the sheer size of plot summaries within this dataset, both in terms of how detailed and lengthy each plot summary is as well as the number of plot summaries, may have an impact on overall computational speed. To allow this study to run in real-time for demonstration purposes, static information such as similarity scores and generated summaries offline were pre-computed.

While this recommendation engine study has potential for future deployment, its computational speed and generated search results would need to be on par with those of Netflix, Disney Plus and Apple TV+. This study addresses this computational speed issue in detail in subsequent sections (both within the data cleaning as well as gap analysis.)

# Data Cleaning Considerations

Merging and Filtering Process

To generate a final meaningful dataset suitable for analysis, an initial merging of separate data files into a single Pandas Dataframe was performed. After this initial merging step, the data was filtered on three levels: first, to exclude movies without a plot summary, secondly to include movies where English is one of the languages, as well as limit the year of release to be on or after 1980. At this stage, the dataset had approximately 14,000 records, and included the following columns: movie name, year of release, revenue, movie language, country of movie produced, genre, plot summary and plot word count.

As a sanity check on whether this study could have utilized alternative datasets for the purpose of plot summaries, a web scrapper was built to scrape plot summaries from IMDb, a well-respected movie database. After filtering out movies without any plot synopsis, the resultant dataset had 14,425 records, which was not significantly different from what we had. Thus, this study continued with the dataset from CMU for the reasons cited above.

Merging with Pegasus Abstractive Summary

This dataset was then merged with the text abstractive summarization output of plot summaries. While the detailed plot summaries remain crucial for the recommendation engine, this study incorporates abstractive summarization to generate a much more concise plot summary. It is hoped that the shorter summary allows the user to quickly skim through the various movies recommended as a first glance. The full detailed plot summary would still be available to him/her, should s/he require a final read on the movie before deciding to view it. Further details on this abstractive summarization are covered in the solution section below.

Pre-Processing using spaCy

spaCy was selected as the Python package of choice for this pre-processing section as well as the solution sections, given its comprehensive processing pipeline and use of word-embedding vectors for cosine similarity calculations (the latter would be elaborated further in the solution section below). Since word-embedded vectors need not be called upon at the pre-processing stage, spaCy’s small pipeline package was used in this section. In this pre-processing step, this study made use of spaCy to perform lemmatization before basic cleanup such as removing additional white spaces and stop-word removal. By visual inspection, there were no other unnecessary artefacts that needed special cleaning (e.g., HTML tags, mathematical equations etc.). These additional requirements could be easily added on, as necessary.

For our study, lemmatization was intentionally performed first before removing stop-words as spaCy allows the use of POS tags to decide whether to lemmatize the text. This was observed to result in better results due to the semantic relationship. The stop words were then subsequently removed after the lemmatization process.

# Overall Flow

Though this study may not mimic a real-world deployment due to a lack of the entire front-end interfaces for the customer, the solution below attempts to closely mimic the full user interaction in relation to the recommendation engine.

Diagram

Description automatically generated

*Figure 1. User Interaction Process*

As represented by Figure 1, the four-step interaction process begins with the user entering a few key words or phrases that best describe the movies that s/he is keen on viewing. The recommendation engine shows an initial list of top 20 movies that best fit the phrases entered based on document similarity. The user is then expected to *‘Like’* movies within this top 20 list that closely resembles the movies that s/he is keen on viewing. A final list of ranked movies is then generated for the user, equipped with its abstractive summaries.

The primary objective of entering the keywords is to perform an initial seeding to return a list of movies the user would add to their *‘Liked’* list. This study sees instances of such approaches in some online applications where upon sign-up, one is asked to choose n-topics that s/he is interested in to start off the recommendation with a list closer to the user’s preference. This approach also helps if the user knows some key phrases that s/he likes from past experiences to create this initial seeded recommendation.

It is also envisioned that as the user adds more *‘Liked’* movies to the list, the recommendation would be trained further in taking these changing preferences into account.

# Solution Overview

To create the overall flow above, the solution revolves around deciding on the optimal use of cosine similarity, evaluating the best text summarization method, creating a small-scale user-friendly interface to mimic the front-end development as well as exploring evaluation methods relating to a highly qualitative dataset. This section provides an overview of the solutions, while a detailed exploration of the solutions is further elaborated in the next section.

Choosing Between Term Frequency – Inverse Document Frequency (TFIDF) vectors and spaCy’s word-embedded vectors in Cosine Similarity Calculations

While it is natural to use TFIDF when calculating the cosine similarity between initial keywords and plot summaries, this may result in a less effective filter. TFIDF excels when such keywords appear in the plot summaries but generates a null value when these same keywords are absent in the plot summaries. Such binary matching may miss identifying relevant movies, despite the absence of such words. Thus, adopting the use of word-embedded vectors allows for a closer matching, where words that may be absent in the plot summaries are replaced with similar words along the same vector. Relevant movies would then have a greater chance of being identified.

Use of Max and Total in Cosine Similarity

Table

Description automatically generated

*Figure 2. Sample Word-Embedded Vectors Cosine Similarity Calculations Between Keywords and Plot Summaries*

Figure 2 illustrates the word-embedded vector cosine similarity calculations of four keywords i.e., cartoon, babies, unicorn, and fantasy, against all the plot summaries in the corpus. To rank the plot summaries, two columns of maximum and total were used. The maximum column pulls the maximum cosine similarity score from the various keywords for each plot summary, whereas the total column sums all the cosine similarity scores across all the keywords for each plot summary.

Different thresholds were used in either filter stages. In the initial filter stage using keywords, a threshold of 0.6 was used on the max column. The plot summaries were then sorted in descending order of the total column scores. A top 20 plot summaries were then generated as an initial result.

Table

Description automatically generated

*Figure 3. Sample Word-Embedded Vectors Cosine Similarity Calculations Between 3 Plot Summaries and The Rest of the Corpus*

Figure 3 illustrates the word-embedded vector cosine similarity calculations of three *‘Liked’* movies i.e., 4397117, 3843721 and 600094 against all the plot summaries in the corpus. In this subsequent filter stage using *‘Liked’* movies, a higher threshold of 0.8 was used on the max column. The plot summaries were again sorted in descending order of the total column scores. An eventual top 20 plot summaries were then generated as a concluding result.

Text Summarization: Evaluating Extractive vs Abstractive

Generating a concise summary is always non-trivial. Given an abundance of available summarizers, picking a suitable summarizer for the dataset was challenging as well as time consuming, owing to a large dataset containing 42,000 movie summaries. In evaluating the most suitable technique, a quality summary should be short, concise and at the same time reflect the core idea of the input content. Further, justifying the final selection also imposed a challenge given the subjectivity of the activity. This was resolved with the help of independent evaluation done by all the team members, who acted as ‘human annotators’ for the task.

R Shiny Application as a Front-End Development Proof-of-Concept

A Proof-of-Concept was developed to give a face to the models designed. After exploring several packages in Python, R Shiny was selected basis on the quality of output it generates. The idea was to develop an interface that would showcase the output of text summarization and recommendation model based on the similarity matrix. The ‘reticulate’ package was used to knit the python codes in R. Due to limitations in time and scope of the module, a minimalist structure was designed, and the creation of a full-fledged UI was left to be an area for future work.

# Solution Details

Text Summarization: Evaluating Extractive and Abstractive Models

The primary aim of the project was to highlight the most apt summary of the movie plots. The theme was to be conveyed in short, succinct sentences which would ease decision making for the user without taking up much time. Both the categories of the summarization technique i.e., extractive, and abstractive was explored. First, two extractive summarization methods were analyzed using the spaCy library and TextRank algorithm, and then three abstractive summarization methods were analyzed using pre-trained models such as T5, BART and Pegasus, before deciding on the best summarizer for the dataset. As the overall summarization process was time consuming, short summaries for the same set of 50 random movies was generated using all five methods, rather than generating short summaries over the entire dataset of 42,000 records.

As a recap, abstractive summarization generates a new unique summary of text given a context whereas the extractive summarization “quotes” and concatenates relevant portions into a summary. Eliminating the extractive summarization methods was straightforward as it concatenated only the important sentences from the original text without retaining the overall meaning. From the remaining abstractive summarization methods, Pegasus model trained on cnn\_dailymail dataset was chosen as the most suitable summarization technique. It performed comparatively well on our dataset with minimal hallucination.

Choosing Between TFIDF and spaCy in Cosine Similarity Calculations

Choosing spaCy over TFIDF allows for the search of a wider range of keywords. Using the large spaCy model, each word token is loaded within a 300-dimension vector space that had been pre-trained on Wikipedia, Common Crawl and Twitter. This means that though certain search words may otherwise be missing within the vocabulary or the plot summaries, such pre-trained word-embedded vectors return similar words that are close to these search words within the vector space. These word-embedded vectors apply not just to the search words, but also to the plot summaries. Thus, each plot summary is converted to a document vector, where it is the average of all its token vectors. Thus, the cosine similarity is calculated on two parts: initially, the word-embedded vectors of the search words are calculated against the document vectors, and for the later part, the document vectors are calculated against each other.

This study notes the use of vector space and its related challenges. First, the length of the plot summaries may have an impact on the document vectors. Since the document vector space is the average of the token vector space, the averages of long documents may regress to the mean. Secondly, it is computationally more intensive to store the vectors and its corresponding similarity matrices. Based on our final dataset containing approximately 14,000 plot summaries, this meant storing a 14,000 by 14,000 similarity matrix which was pre-computed. This study notes a solution to this would have been to store only half of the similarity matrices instead, but this methodology adds another layer of coding complexity.

R Shiny Application as A Front-End Development Proof-of-Concept

To facilitate user interaction with our recommendation model, a proof of concept was designed using R Shiny. The interface project was built in R Markdown file format in RStudio, the file format allows easy integration of the following pieces into one file:

1. Text, to display information about the interface
2. Images, to enhance the design of the interface
3. R code chunks for reading the data to be used for the Shiny app.
4. Python code chunks, taken from the movie recommender model and the movie summarization model and deployed in R using the reticulate library suitably. Reticulate library in R facilitates embedding of a python session within an R session.
5. Shiny UI and Server code chunks, that enabled the creation of the interface/application.

The app **homepage** displays the title of the app with an image and contains 2 tabs explained as follows:

1. **Movie Summary tab:** Allows users to select a movie of their liking and see the attributes such as release year, country, genres, and a short plot summary generated using the Pegasus model. This tab is designed for instances where users would want to simply view the mentioned attributes for a movie. Thus, this summary saves them time and facilitates quicker decision making.
2. **Movie Recommender tab:** Allows users to select multiple movies and obtain movie recommendations based on them. The 14,000 X 14,000 similarity matrix built from the recommender model forms the basis of this tab. Top 20 movie recommendations based on the maximum and total similarity scores along with their attributes are displayed as shown in Figure 4. The recommendations are updated as and when the user *‘Liked’* more movies.

Graphical user interface, text, application

Description automatically generated

Figure 4: Movie Summary and Recommender Interface

# Results and Analyses

Text Summarization: Evaluating the best-fit model

As detailed above, to generate summaries that forms the crux of the movie plots with minimal words, this study adopted a trial-and-error approach and experimented with various techniques to identify the best fit option. Below is a snapshot of a sample plot summary (shown in Figure 5) and the output of all the extractive and abstractive techniques (shown in Figure 6) explored to obtain the most apt summary.

Text, letter

Description automatically generated

*Figure 5. Example Of a Plot Summary*

Graphical user interface, text, application

Description automatically generated

*Figure 6. Comparison of Various Text Summarization Models*

The extractive summarization technique, using spaCy and TextRank, was easily eliminated, considering these techniques only reproduce the key sentences of the main text, at times without retaining the main meaning. The challenge was to evaluate the four techniques under abstractive summarization method. Among the models tested, the pre-trained Pegasus model trained on cnn\_dailymail dataset returned the most meaningful summary compared to the other models. This conclusion was based on the independent evaluations of the team where each of the team member had sampled selected movie plots and assessed the results within each model. As per the unanimous consensus of the team, the pre-trained Pegasus model trained on cnn\_dailymail dataset appeared to be the best fit under the circumstances.

Results from the Movie Recommender:

To solve the common issue of a ‘cold start’ in most recommender systems, the movie recommender allows users to enter keywords that are on their minds. These keywords are used to retrieve relevant movies. E.g., if a user enters the search query as per Figure 7 –



Figure 7: Search Query to Get Movie Recommendations

The initial output will be a top 20 movie recommendations based on the queried word-embedded vector, 5 of the 20 movie recommendations for the example query are listed in Figure 8:

Table

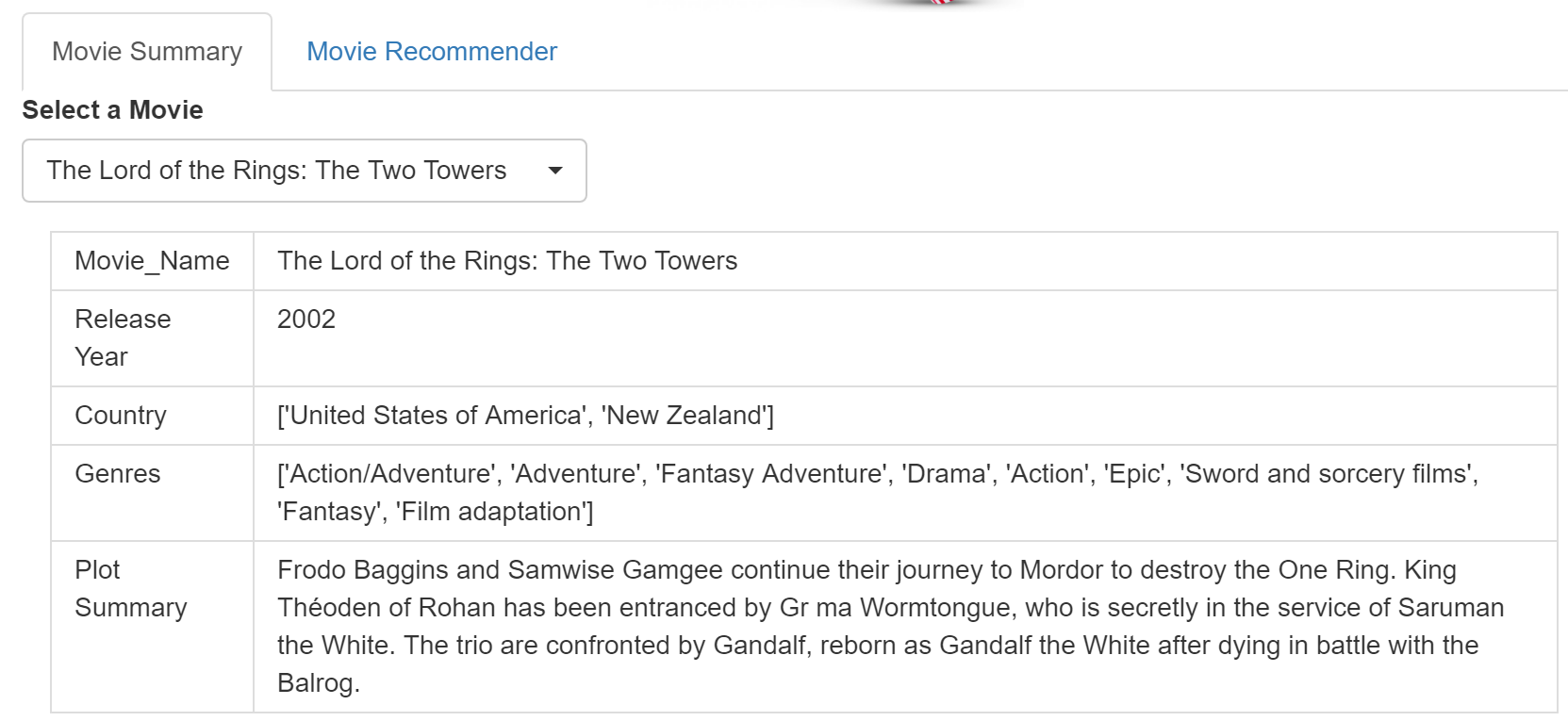
Description automatically generated

Figure 8: Showing 5 Of 20 Movie Recommendations for The User Entered Keywords

From Figure 8, the recommended movie names and genres relate to the sample query words e.g., military, war, world war etc. suggesting that the results are relevant to the keywords in the search query.

Another instance when movies will be recommended to the user is when they will *‘Like’* specific movies from the list of all movies or the movies that are displayed on the screen.

Figure 9 displays the details of a sample movie *'Liked'* by the user. Figure 10 displays 5 out of 20 movie recommendations based on the user *'Liked'* movies, the theme of the '*Liked'* movie closely match the themes of the recommended movies e.g., the genres, which are indicators of the theme of the movie plots, are same among the *'Liked'* and the recommended movies.



*Figure 9. Sample Movie 'Liked' By The User*

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

*Figure 10. Showing 5 Of 20 Movie Recommendations Based On The User 'Liked' Movies*

First Evaluation – Using Gensim’s TFIDF

To measure the accuracy of the recommendation engine, two separate evaluations were performed. First, a comparison is done against Gensim’s TFIDF. The main assumption when comparing the accuracy of spaCy’s word-embedded vectors against Gensim’s TFIDF is that the initial query words are within the vocabulary. The same pre-processing steps in the above Data Cleaning Considerations section were used to ensure the same dataset was used for Gensim’s TFIDF. This dataset was then converted into a dictionary, and its similarity matrix based on TFIDF was calculated as per Figure 11.

Table

Description automatically generated

*Figure 11. Sample TFIDF Cosine Similarity Calculations Between 3 Plot Summaries and The Rest of the Corpus*

A comparison of the same three *‘Liked’* movies were used for this TFIDF calculation. Given that the TFIDF cosine similarity results are lower than the word-embedded vector cosine similarity results, a lower threshold of 0.1 was used on the max column. In similar fashion, the plot summaries were then sorted in descending order of the total column scores. The eventual top 20 plot summaries were then generated as a concluding result.

|  |  |  |
| --- | --- | --- |
| Ranked In Terms of Best Match | TFIDF | spaCy’s word-embedded vectors |
| Top Match | Day One | Deterrence |
| 2nd | W. | The Sum of All Fears |
| 3rd | Red Corner | Red Dawn |
| 4th | Capitalism: A Love Story | Countdown to Looking Glass |
| 5th | DC 9/11: Time of Crisis | Mother Night |
| 6th | Bush Family Fortunes: The Best Democracy Money Can Buy | Megiddo: The Omega Code 2 |
| 7th | Roger & Me | Day One |
| 8th | Sicko | Why We Fight |
| 9th | Footloose | The Last Days of Patton |
| 10th | Best Defense | 5 Days of War |

*Figure 12. Comparison of Top 10 ‘Liked’ War-Related Movies Between TFIDF and Word Vector*

A comparison was made on the top 10 plot summaries highlighted by both Gensim’s TFIDF and spaCy’s Word Vector within Figure 12. A cursory subjective look suggests that spaCy’s Word Vector highlighted more war-like movies than Gensim’s TFIDF.

Second Evaluation – Genre Comparison

To build on the earlier movie comparison, the genres of the top 100 movies was extracted from both methods.

|  |  |  |
| --- | --- | --- |
|  | TFIDF | spaCy’s Word-Embedded Vector |
| War-Related Genre | 22 | 47 |

*Figure 13. Extracting Genres from The Top 100 ‘Liked’ War-Related Movies Between TFIDF and Word-Embedded Vectors*

In Figure 13, this study finds that 47 of the top 100 movies recommended by spaCy’s word-embedded vectors had a war-related genre, compared to just 22 recommended by Gensim’s TFIDF.

# Discussions and Gap Analysis

Computationally Intensive Processing / Scalability

In our project, two different mechanisms were used to deal with typical CPU-bound and Memory-bound challenges often encountered for Big Data analysis such as NLP and Computer Vision.

The immediate solution that benefited the project was the use of Python’s Multiprocessing library. Due to the Global Interpreter Lock (GIL) in Python, where only one thread is allowed to hold the Python interpreter, it is not natively possible to run processes to utilize the entire CPU cores available. This study had access to a server with a 32-core CPU but was only able to properly utilize one core while the other 31 cores were idling.

When the 14k x 14k similarity matrix was being calculated, this would have needed approximately 8 hours to complete based on initial test runs. However, given the availability of CPU cores and the fact that this is a typical *embarrassingly parallel* problem where no state information needs to be exchanged, the Python Multiprocessing library could have been easily used to split the load.

By creating such logic to split and rejoin back the results, the computation of the similarity matrix was completed in 19 minutes using 24 CPU cores.

Another external library used was Dask[[1]](#footnote-2) , which allowed us to split a Pandas Dataframe into smaller chunks of Pandas Data frames. Combined with its lazy evaluation methods, one can build a series of steps which are then internally optimized when a compute call is made to execute the stream of activities. This study used this in our lemmatization processes as the corpus was loaded into data frames which could be automatically handled by Dask in the parallel processing. This study managed to cut down the lemmatization process across the entire dataset from about 20 minutes to complete in 5 minutes.

While this study could have made use of the Python Multiprocessing logic, the data frames would have needed to be broken up and recombined manually. In addition, Dask supports distributed compute over a local or network cluster, and this scaffold allows us to scale the implementation for even larger datasets and more complex dataframe operations in the future.

Thirdly, this study also noted that the spaCy algorithms also support parallelized methods to convert the texts to the spaCy Doc objects. This was found when reading textbooks and API documentations, and the results are much faster than building Python loops as spaCy distributes the workload to multiple CPU cores. This study used this during the conversion of the movie plots into the spaCy Doc objects in our reference dictionary used by the core engine (which needed access to all word vectors and text).

Further Enhancing Recommendation Engine by Incorporating Additional Attributes Such as Cast, Character Names and Sentiments of Movie Reviews

The recommendations that are currently only based on the plot summaries of the movies can be improved by including other attributes such as movie cast, directors, revenue (a symbol to ascertain popularity of a movie), genres, movie reviews and aspect-based sentiment analysis.

Movie plots could match a user’s liking but still the overall delivery of the movie including the acting skills of the actors, cinematography, etc. might not match the user’s interests. Hence, by taking the sentiments of the movie reviews into account, the recommendations can be improved. Aspect based sentiment analysis can further leverage insights on a particular user’s preferences- whether they like to watch movies based on the cast, popularity, or the plot. In addition, movies released within the same time could also receive a boost in likelihood for recommendation. However, the above ideas may lead to overfitting, given that a user’s taste can be highly varied.

Additional Evaluation Based on Human Annotators and Publicly Accessible Movie Databases (IMDb Movie Recommendations)

The most appropriate model for generating text summaries of movie plots was selected on a basis of uniform consensus of the team. This internal evaluation was restricted to some random samples selected by the team members. There lies further scope on model evaluation by introducing Human Annotators who can provide independent assurance on the functionality and efficiency of the model.

Further, to assess the quality of recommendations given by the recommendation engine, a comparison could be made with publicly accessible movie databases (IMDb Movie Recommendations). This comparison currently was left unexplored due to limitations on the dataset. For movies which are not part of the base data, it would naturally not be recommended by the engine, thereby creating a biased view when compared.

Adapting Recommendation Model on Academic Journals with Formulae and Equations

Given that this recommendation engine builds off word vectors, it is natural to consider other documents that contain large tracts of words beyond plot summaries. A noted consideration would be to implement this recommendation model on academic journals. The added benefit is that the double filtering mechanism within the recommendation engine would identify word vectors that closely match the query words, leading to a more robust search result. This recommendation engine could also be added to return journals that match certain formulas or equations entered by the user.

A Fully Functional User Interface to Deploy Models Built and Facilitate Easy User Interaction

While as of now, the Proof-of-concept designed effectively highlights the summarization work but only a portion of the recommendation model has been deployed. The interface to allow users to enter keywords for an effective search is yet to be designed. A fully functional User Interface (UI) to deploy both the models and facilitate easy user interaction can solve the widespread problem of finding *‘Like’* movies.

Incorporating Rocchio Algorithm in Recommendation Engine

Given that the latter half of the double filtering mechanism involves the use of *‘Liked’* movies, tweaking this to include the Rocchio algorithm seems a progressive step. The Rocchio algorithm assigns positive weights to documents that match the original query and negative weights to non-related documents. This is akin to the user “disliking” movies that does not match his preference, on top of currently “liking” movies that do match his preference. Though it is advantageous to rank movies as non-relevant, the algorithm places a greater weight on relevant matches, based on its traditional values. The Rocchio algorithm does allow for tweaking of the weights between relevant and non-relevant matches. On this basis, further study is required to compare the increase in accuracy from the recommendations made from the current engine against one that utilizes the traditional Rocchio algorithm as well as its various changes in weights.

# Conclusion

Recommendation engines enable users to search an endless amount of data and retrieve only relevant information that fits the user’s search queries in a timely manner. This study has demonstrated that the use of a double filter i.e., spaCy’s word-embedded vector algorithm and *‘Liked’* movies, generated a more relevant search result than just solely relying on TFIDF. Additionally, this study also presents users with shortened summaries of movie plots to help them choose a movie based on the plots they *'Like'* instead of wasting time going through often mixed and subjective online reviews or reading a long Wikipedia page about the movie.

Having said that, further studies need to be conducted on how to effectively incorporate other attributes mentioned in the gap analysis section. Furthermore, the similarity matrices and the summaries can be improved in terms of accuracy by considering other models and perhaps creating an ensemble of relevant models and techniques.

# Project Experiences/Reflections

**Gerry CHNG Kian Woon:** There are multiple libraries available which can be confusing for adopters (e.g., NLTK, CoreNLP, spaCy, gensim, Huggingface, etc.). Both NLTK and CoreNLP are useful for academic uses, and some of the others are more popularly used in production environments. It is also important to go through the API documents to understand more about how the algorithms are implemented, and whether there are methods created to facilitate common tasks.

One of the other key learning is that tasks in NLP will very quickly hit CPU-bound and Memory-bound challenges due to the large corpus size, or the vocabulary size. In these situations, design choices will have to be made about filtering to the necessary components, using distributed and/or parallel computing (e.g., using Dask or Multiprocessing as discussed in this paper), or dimension reduction through PCA or SVD to satisfy practical time and space constraints.

In considering multi-processing, the solution architecture and algorithm design choices are also important to ensure that the overheads in multiprocessing (due to state information exchange) is minimized as it can significantly compromise the benefits of parallel computing.

Additionally, I also learnt that web-scrapping and data cleansing skills are particularly important in NLP as not all the datasets may be available in the formats that is needed. In our project, we used scrapy to build the IMDb web-scrapper for initial evaluation to compare whether an alternative dataset can be better.

**Karishma YADAV:** Through this project, I have learnt the application of text mining models studied in class, specifically the analytical tasks performed in this project – document representation and retrieval and text summarization. One of the key learnings is how to handle unstructured data and cleaning it to transform as inputs for our models. I have also learnt about the various techniques available for the analytical tasks we performed in this study and how to choose the most appropriate models by evaluating and comparing them against one another. Additionally, I understood the intricacies involved in the deployment of models built in python into R shiny through the integrated file format – R Markdown and 'reticulate' package in R that facilitates embedding of a python session inside an R session. I also learnt how RShiny can be used to create an effective interface and how to troubleshoot some of the common issues that one may encounter while building an interface. Another key learning is the limitations regarding the data set, the big data challenges that textual data might face and how to overcome them via use of multithreading or Dask.

**Mayurapriyann ARULMOZHI BASKARAN:** With this project, I gained a much better understanding of how the content-based recommendation system works. The concepts such as text pre-processing, document similarity, text retrieval, and summarization techniques discussed in our coursework were helpful during the project phase. I was able to learn about the limitations of the TF-IDF model in our use case and how the use of the spaCy model over the TF-IDF model helped to overcome these limitations. As part of this project, I had the opportunity to learn and explore the internal workings of various extractive and abstractive summarization models. The optimal model for our dataset was selected manually through trial-and-error approach. But the evaluation of the model's performance on our dataset is still remains challenging. Another key learning was speeding up the computations with Dask, a parallel computing framework that helped to cut down our processing time of our dataset while performing computation intensive tasks.

**SYED Ahmad Zaki Bin Syed Sakaf Al-Attas:** I did not realise how deep and wide the field of text analytics was til I embarked on the course as well as this project! With many of the text analytics packages residing in a Python environment, one is expected to have a working knowledge of this programming language. While there are other software and programming languages that cover text analytics, neither are as trailblazing in terms of both research and application as these Python packages.

Another reflection from this study was on evaluating the accuracy and usefulness of the summarization model. While there are computational evaluation models such as ROUGE to measure the accuracy of such summaries, it does not provide a good gauge on either the fluency or how close it mimics a human annotator. Text summarisations, done by humans, have an element of language competency, background, cultural nuances and style of writing. Conversely, while the abstractive summarisations such as PEGASUS was pre-trained on a wide variety of corpus, this meant that the abstractive summarizations may ‘regress to the mean’ from all its pre-trainings. As such, PEGASUS may not be able to tweak its summarization output according to its target reader.

**Vertika PODDAR:** The objective of taking the module was to gain a better understanding of the subjective world of Text Analytics. Text Summarization and Document Similarity were the two main themes of our project. Given the sensitivity of the deadline, we decided to adopt the approach of divide and conquer and put the best use of our skills. I had teamed up with another team member to design an User Interface (UI) to showcase the output of our project idea. UI execution demanded rewriting the python codes using the ‘reticulate’ package in R. This activity empowered me to refine and solidify my understanding of the models designed by my fellow team members. The trial-and-error method used to explore various techniques, helped me appreciate the peculiarities of text pre-processing, extractive and abstractive text summarization, and document similarity techniques. I understood the limitations poised by TF-IDF models. Further, another major learning is parallel computing with the use of Dask to solve the issue associated with processing time. Having worked on the UI, not only I have gained an in-depth understanding of the models built and the practical challenges encountered but I am now better equipped in designing an interface. All this is a product of peer learning. I am grateful to my team members who discussed and explained the challenges they encountered while writing the codes for the models. My learnings and takeaways from the project would not be so rich if not for the diverse and experienced background of all my team members.

# References

Bamman, D., O’Connor, B., & Smith, N. A. (2013, August). Learning latent personas of film characters. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 352-361) [Data set].

Shinywidgets overview. shinyWidgets. (n.d.). Retrieved from <http://shinyapps.dreamrs.fr/shinyWidgets/>

Welcome to shiny. Shiny. (n.d.). Retrieved from <https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/>.

Ushey, K., & Allaire, J. J. Interface to python. Interface to Python • reticulate. Retrieved from <https://rstudio.github.io/reticulate/>

1. <https://dask.org> [↑](#footnote-ref-2)