CALIFORNIA STATE UNIVERSITY
FULLERTON™

Department of Computer Science

This project has been satisfactorily demonstrated and is of suitable form.

This project report is acceptable in partial completion of the requirements for the Master of Science degree in Computer Science.

AI Enhanced Recommender System for Movies

Project Title (type)

Shayan Darian

Student Name (type)

Rong Jin

Advisor's Name (type)

Advisor's signature	Date

Rong Jin

Reviewer's name

Reviewer's signature	Date

Abstract: In the era of rapidly expanding digital media libraries, personalized movie recommender systems have emerged as critical tools to enhance user experience and satisfaction. This report presents a comprehensive analysis of a movie recommender system project that implements a dual content-based filtering and collaborative-based filtering system, by leveraging TF-IDF Vectorization, User-Item Interaction Matrix, the K-Nearest Neighbors algorithm, and comparing and analyzing the results of different similarity/distance metrics, to deliver precise and engaging recommendations to a user, regardless of whether they're a returning user, or a brand-new user. The recommender system integrates user profiles and meta data, such as rating reviews, to predict user preferences effectively, and solves many of the common problems of recommender systems today, such as proper algorithm selection, data accuracy, cold start, and overspecialization. Experiments are conducted on publicly available movie datasets, evaluating the system's accuracy, scalability, and computational efficiency. The results demonstrate that the Cosine Similarity and the Correlation Distance similarity/distance metrics outperform other measurement metrics, such as the Euclidean Distance, Manhattan Distance, and the Bray-Curtis Distance, in terms of reliable and accurate results and user satisfaction. The research and results presented in this report highlights the potential of recommender systems in transforming the way audiences discover and engage with digital media.

Keyword List: Artificial Intelligence, Machine Learning, Recommender System, K-Nearest Neighbors, Collaborative-Based Filtering, Content-Based Filtering, Cold Start, Overspecialization, User-Item Interaction Matrix, TF-IDF Vectorization, Similarity Metrics

Table of Contents

List of figures

Introduction

Description of the Problem

As companies and businesses attempt to further increase and maximize profits, they have sought to develop a system that is able to directly recommend products and services to their customers. Systems such as these are referred to as Recommender Systems. They are built and designed with the goal of allowing the company or business with whose service the customer is using, to suggest new and additional products and items to the customer, which they will then likely be interested in purchasing or viewing, thereby improving and enhancing the user experience for the customer or user. Though the use of recommender systems has expanded beyond the use of commercial enterprise, and has practical applications for governments, universities, and other organizations. The goal of a recommender system is to solve a major problem, which is that of information overload for the user, which will then make the decision process easier and simpler for them, and will therefore enhance the user experience.

The first recommender systems came into existence a little more than 20 years ago, and were created through methods and techniques outside of the field of artificial intelligence (AI) and machine learning, especially in areas such as user profiling and preference discovery.

Over the last decade, we have seen a lot of progress and success made in successful AI-driven applications. Some examples of this include Deepmind's AlphaGo, the AI-driven program that was able to win the game of 'Go' when it played against a professional human player, in addition to the self-driving car, as well as others in the fields of computer vision and speech recognition. The advances that we have seen in AI, data analytics, and big data, present to us an amazing opportunity to embrace these impressive advancements in AI, and to see how we can use them to enhance recommender systems. A good summary regarding the history of recommender systems is discussed in [1].

Traditionally, the main goal of recommender systems has been to suggest new items to a user based on what they are most likely to want, through attempting to accurately recommend different items to a user that are similar to the ones that they have interacted with. The recommender system has done this through different methods, but in more recent times with advancements and further integration of AI and machine learning, this has been done by attempting to learn a user's behavior and then suggesting new items to the user based on predictions from that.

A major flaw of current recommender systems is that they are unable to give recommendations until they have collected enough data about a new user. This data is typically collected through your browsing history or through reviews that you have given. This issue is referred to as the Cold Start problem which will be discussed and expanded upon in greater detail later on in this report, and I will showcase my approach on properly addressing and solving this issue. This issue exists for new items as well. For example, at the time of writing this report, if you go to YouTube without any prior browsing history, instead of receiving any recommended videos to watch, you will get a message that says "Try searching to get started. Start watching videos to help us build a feed of videos you'll love". This is due to the fact that YouTube's recommender system has no data or information about you, and so it doesn't know what it should recommend to you.

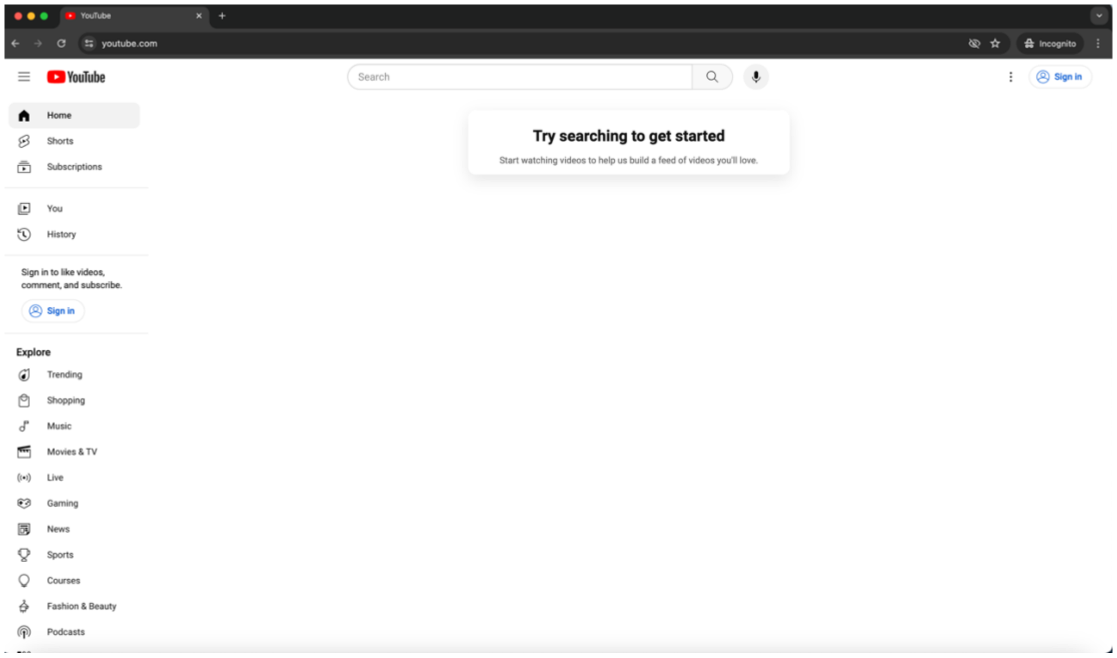


Figure 1 – YouTube's Cold Start Problem - youtube.com

For the purposes of this project, the recommender system that I built was a movie recommender system that can recommend movies to a user, and it is an improvement over many other current recommender systems, including those that leverage AI and machine learning, due to solving some key issues that modern recommender systems face, which include proper prediction algorithm selection, ensuring strong data accuracy, solving the issue of data sparsity, otherwise known as the cold start problem, and solving the issue of overspecialization. More details on these key issues will be addressed in the next subsection.

Key Issues

The first of these key issues that I will be addressing, is the issue of proper prediction algorithm selection. There are many different algorithms that can be implemented in creating a recommender system. This algorithm will need to implement the functionality of predicting the value of items for a particular user, which will vary according to the recommendation algorithm selected. Knowing which recommendation algorithm to select, is crucial in the accuracy and performance of a recommender system. From conducting thorough research, I have found that the K-Nearest Neighbors (K-NN) algorithm is the best machine learning algorithm to use for this task. It can be used to implement both Collaborative-Based Filtering and Content-Based Filtering, which will be discussed later on in this subsection.

The second of these key issues that I will be addressing, is the issue of ensuring strong data accuracy. Naturally, a database containing fewer movies in it, will have a higher accuracy in making correct recommendations. Whereas, a large database containing many movies in it, will have a lower accuracy due to the pool of information searched being too large. Therefore, being able to maintain a high level of accuracy regardless of the size of your database is crucial for a product that wants to ensure a good user experience. My solution was to use the K-Nearest Neighbors (K-NN) algorithm to also address this issue, while also experimenting with different types of distance metrics to see what produced the most accurate results. I found that Cosine similarity and Correlation distance both produced equal results that were also the most accurate. I used Cosine similarity as the final distance metric, due to it being a more popular and well-known distance metric, which would likely generalize better. I also considered going with a deep learning approach, where I would use an Artificial Neural Network (ANN) to generate the recommendations, but due to time constraints, I did not end up going with this approach. More information regarding the K-Nearest Neighbors algorithm, along with variations of this algorithm, can be found in [5].

The third of these key issues that I will be addressing, is the issue of data sparsity, also known as the Cold Start problem, which exists in the collaborative-based filtering approach, which is the standard approach that many modern AI based recommender systems use. When there is not enough data or information about an item (items in this project being movies) or a user, the system will not know how to categorize them relative to other items or users in the system's database. The cold start problem exists for both the scenarios when there a new user in the system, as well as when there is a new item in the system. In the case of a new user, when we do not have data or information about a particular user, it can make it very difficult to provide the user with relevant and accurate recommendations. Whereas in the case of a new item, not having enough reviews or other relevant information on said item, can make it difficult in determining how to categorize or rank its relevance, or it can make it difficult to determine how to accurately recommend the item to the correct demographic of users. My solution to implementing the collaborative-based filtering approach, while also solving the cold start problem, is to use a different filtering method, known as content-based filtering, when a new user or new movie has been introduced into the system, effectively giving the new user or movie a cold start. Once enough data and information has been gathered on the user or movie, to where they are no longer considered "new", then the system will switch to using collaborative-based filtering for said user or movie. Shown in figure 2 below is a diagram detailing how collaborative-based filtering works. For users, they are matched together based on their similarity to one another, which can be measured using cosine similarity, or some other distance metric, such as correlation distance. These similarities can be determined by collecting data on their browsing history, or even better on what they ratings they give to the movies that they watch. The more similar the ratings, the more similar the users. So, in figure 2 below, which can be found in [2], user 3 is determined to be similar to user 1, because they have both watched item 3 and item 4 in the list. Since user 1 has additionally watched item 1, the recommender system can recommend item 1 to user 3, because they are a similar user to user 1. A similar phenomenon is shown for similar items on the right-hand side of the diagram.

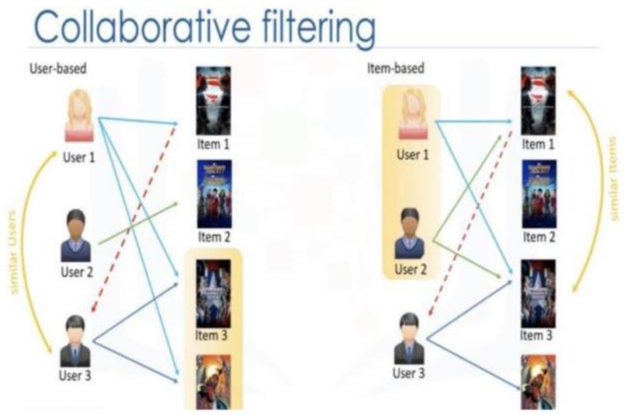


Figure 2 – Collaborative-Based Filtering, Malik, S. (2022) [2]

Shown in figure 3 below, which is also taken from [2], is a diagram detailing how content-based filtering works in comparison to collaborative-based filtering for a new user. Content-based filtering will recommend a new user items (in the case of this project those items are movies) that are similar to other movies based on the content features of the movies themselves, such as genre, cast, director, movie description, etc. If basic demographic information about a user is known, such as their age, gender, and other similar information, then they could be movies watched by users of a similar demographic, otherwise they can also be watched by the general audience or userbase population. This can be used to give either a new user or a new movie a cold start. From how a new user interacts with these initial recommendations, the system can start to profile the user and learn their behaviors, and thus learn to recommend them content that is more specific for them, and this can be done through collaborative-based filtering. Further reading on the Cold Start Problem can be found in [3].

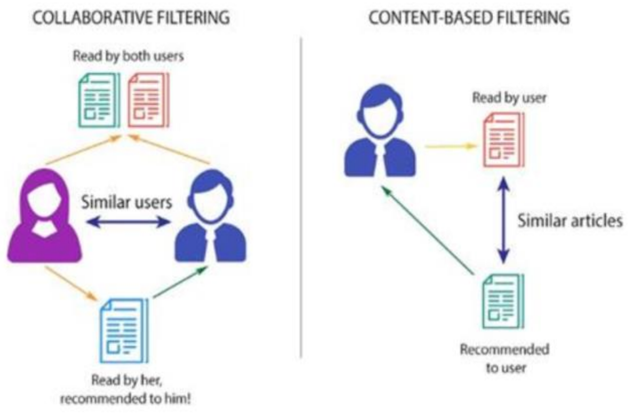


Figure 3 – Collaborative-Based vs Content-Based Filtering, Malik, S. (2022) [2]

The fourth of these key issues that I will be addressing, is the issue of Overspecialization. Once the system has profiled and learned the behaviors of a user very well, the user will start to always get similar types of recommendations, with no real variety or way to introduce them to different domains. This can cause the user to become bored and unable to discover new types of items through their recommendations. My solution to this, is to assign a low probability of sometimes recommending extra unrelated items to the user, unrelated to their other recommendations. This will allow them to occasionally see different types of movies in their recommendation list, perhaps introducing them to something that even they didn't know that they would like. Further reading on Overspecialization can be found in [1] and [4]

Project Objectives

The main objective of this project is to build an enhanced movie recommender system based on an AI and machine learning approach, which addresses and solves many of the key issues that modern recommender systems face, even those already leveraging AI and machine learning approaches. Breakthroughs in AI and machine learning during recent years have offered much insight and opportunity into integrating these techniques and methodologies into existing technologies, and recommender systems seem to be an area which could see significant advancements and improvements through this integration. But these advancements certainly have their problems as well, and my goal in this project is to solve some of the biggest problems that these advancements bring.

Using the K-NN (K-Nearest Neighbors) algorithm with a similarity measure such as cosine similarity or the correlation distance, is an effective and accurate way to determine the similarity between users, as well as movies. K-NN also yields high data accuracy, which solves two of the key issues presented in this report, with one method.

Implementing collaborative-based filtering works well for recommendations when there is already sufficient data and information that has been built for users and movies through their mutual interactions, but this filtering method has the flaw of not being able to suggest any recommendations to new users due to there being no sufficient data on the user yet, and likewise for new movies, it will not be able to accurately suggest these movies to users due to the lack of data on how users liked or interacted with the movie. This flaw is known as the Cold Start problem. Due to it being the most popular machine learning filtering implementation for modern recommender systems, collaborative-based filtering was still implemented for this project, but only for users who have already been given a cold start. Users and movies will be given a cold start through the implementation and use of content-based filtering, which was used for recommendation algorithm used when new users and new movies entered the system, for which the system had no information or data on. Thus, the main flaw of collaborative-based filtering does not exist in this hybrid filtering approach of using content-based filtering to give a new user or new movie a cold start, and then to use collaborative-based filtering on users or movies for whom there is already sufficient enough information.

The recommender system that was built for this project also had a method for dealing with the issue of Overspecialization. The solution to this overspecialization problem was to where a certain percentage of the time (this is an arbitrary percentage that can easily be changed), the recommender system will sometimes show the user unrelated or different types of movies from the ones normally in their recommendations, allowing them to explore different types of movies that they may end up enjoying. Though it is important to keep in mind that the percentage must be set to an amount to where these extra unrelated recommendations will not show up so frequently to where it will negatively impact their user experience. Instead, it will enhance their user experience by keeping them engaged and preventing them from getting bored with the same or similar recommendations.

Initially, I had also done research for and explored the idea of using a deep learning or Artificial Neural Network approach in creating the recommender system, but due to time constraints, I instead decided to go with a more traditional machine learning approach.

Development Environment

In terms of the hardware environment, the recommender system that was built for this project, was developed on computer hardware that had the specifications which met the requirements needed to implement the recommender system and handle the computationally heavy tasks required for a machine learning project of this caliber. Such specifications included a high-performing computing environment, with the use of Graphic Processing Units (GPUs), which could not only handle, but also speed up the computational time of the processes and calculations performed throughout this project tremendously. In order to do this from my own personal laptop, I had to connect to a Python kernel via Visual Studio Code. Running the project via Google Collab, could bypass their requirement however, and I used both throughout the development of this project.

In terms of the software environment, this project was built entirely using Python. Python has many libraries that are well suited for machine learning projects, such as NumPy, Pandas, TensorFlow, Keras, PyTorch, Scikit-Learn, and others. For this project, the Python libraries that I used are NumPy, Pandas, Scikit-Learn, and random. NumPy was used for some important calculations, Pandas was used to handle dataframes, Scikit-Learn was used to help implement the K-NN algorithm for the content-based filtering and collaborative-based filtering models, and random was used for the random number generation used in solving the overspecialization problem. I originally used Matplotlib and Seaborn as well, but I ended up discarding the code that I used them for, later on in the testing stage of development. If I had ended up going with a deep learning approach, I would have also used either Keras or PyTorch. Python allowed for a machine learning project, such as this recommender system project, to be built and implemented in a smooth and efficient environment. Git was also used to serve as the version control system, which allowed the tracking and management of the source code, and allowed for rollbacks in the code when needed.

In terms of the development environment, I used both Google Collab and Visual Studio Code as the Interactive Development Environment (IDE) when developing and building the recommender system, as they were both very helpful when it came to training and testing the system. Both had their advantages and disadvantages. I found Google Collab to be faster but more unstable, whereas Visual Studio Code was slower but more stable. Google Collab also had

Git built into it, so I mainly used this as my version control system, though I did also upload the project file and all the datasets used to GitHub.

Operational Environment

In terms of the hardware environment, the operational aspects of the hardware environment are similar to the development aspects of the hardware environment. While this project was not deployed to a platform such as Amazon Web Services (AWS) or Azure (though theoretically it could), its operational capabilities were thoroughly tested using a local personal computer. Though like with the development environment, the operational environment must be able to meet the requirements needed to implement the project's recommender system and to be able to handle the computationally heavy tasks required for a machine learning project of this caliber. Such specifications, such as with the development environment, include a high-performing computing environment, with the use of Graphic Processing Units (GPUs), which could not only handle, but also speed up the computational time of the processes and calculations performed throughout this project tremendously. Access to a Python kernel is also needed in the operational environment, but this was and can be easily set up through the use of Visual Studio Code.

In terms of the software environment, the operational aspects of the software environment are also similar to the development aspects of the software environment. The operational environment must be able to run Python and the required Python libraries, such as NumPy, Pandas, Scikit-Learn, and random. While Visual Studio Code is technically not required, as theoretically any IDE with access to Python and the functionality of connecting to a Python kernel would work, Visual Studio Code is what I used as the software environment for operating this recommender system project.

Requirements Description

Interfaces

The interface used for this project was the standard integrated terminal interface of Visual Studio Code. Both for asking the user for input, as well as outputting text back to the user, the standard integrated terminal interface was the only sort of interface that was used.

External Functions and Methods

In terms of the external functions and methods, the external functions and methods that I used mostly came from the Python libraries that I imported for this project, notably from the NumPy, Pandas, Scikit-Learn, and random libraries. It should be noted however, that some external functions and methods were default functions of Python not associated with any specific imported library. I will give an overview of these external functions and methods in this section, but I will also go over them when discussing the architecture and design of the project in the next section.

The first external function/method that is required, is Pandas "read_csv()" function, in order to import the needed datasets. The first parameter that was given to "read_csv()", was the name of the csv file that was being imported. Since the datasets in their original form, have every column entry fit into one column, I had to separate these into separate categories. Fortunately, the datasets used two colons ‘::’ as indicators between separate variables. So, the second parameter that was given to “read_csv()” was the “sep=’::’”, in order to separate each variable when a double colon ‘::’ was encountered. The third parameter that was given to “read_csv” was “engine=’python’”, indicating that the parsing engine that should be used to read the csv file, should be a python engine. The fourth parameter that was given to “read_csv” was “header=None”, specifying that no row should be treated as the header, and will instead auto-assign default numerical column names, starting from 0. Finally, the firth and last parameter that was given to “read_csv()”, was the names that I would like to assign to each column. I also called the Pandas “.dropna()” external function, in order to remove any rows or columns with missing values.

The second and third external functions/methods that are required, are Pandas “.shape” and “.head()” methods, respectively. The “.shape” method is used to get the number of rows and columns in the DataFrames. By default, the “.head()” method is used to output the first 5 rows in the given DataFrame, but you can pass a specific integer number as a parameter to the “.head()” method to output that same number of rows specified, but if no parameter is given, then by default it will out the first 5 rows.

The fourth external function/method that is required, is the Pandas “.astype” method. This was used to convert the entries of each column in the DataFrames, into a specific datatype, to ensure their compatibility with future operations, functions, and added DataFrame entries. For example, all IDs were converted to the 32-bit integer datatype “int32”, whereas column entries for the movie titles and genres were converted to the string datatype “str”, and all entries for the movie ratings, were converted to the 32-bit float datatype “float32”.

The fifth external function/method that is required, is the Pandas “.drop()” method, which allowed me to delete columns from the DataFrame, that were of no practical use to the purposes of this project. The columns that I wanted to drop, were passed as the parameter argument to the “.drop()” method. For example, “.drop(columns=’Timestamp’)” was used to drop the timestamp of the exact moment a rating for a movie was submitted, as this information served no practical purpose for the scope of this project.

The sixth external function/method that is required, is the Pandas “.merge()” function. Since I originally imported three datasets, which were “movies.csv”, “ratings.csv”, and “users.csv”, I used the merge function to merge their DataFrames into one big DataFrame, containing all the information of the three datasets. Since the “.merge()” function can only merge two DataFrames at a time, I first merged the DataFrames for “movies.csv” and “ratings.csv” together, and then merged that DataFrame, with the DataFrame for “users.csv”. I passed three parameters to the “.merge()” function, with the first two being the names of the DataFrames that I was merging, and with the third parameter being the name of the common column that I was merging them on, as all three of the datasets used for this project, came from the same source. For example, “movies.csv” had column entries for the movie ID, the movie title, and the movies genres, whereas “ratings.csv” had column entries for the user ID, the movie ID, the movie ratings, and the movie timestamp (which was first dropped from the ratings DataFrame using the ".drop()" function, so this column is not relevant). So, when I merged the DataFrames for these two datasets, I ended up with one merged DataFrame, containing column entries for the user ID, the movie ID, the movie titles, the movie genres, and the movie ratings. Similarly, "users.csv" contains column entries for user ID, the users gender, the users age, the users occupation, and the users zip-code (which was first dropped from the users DataFrame using the ".drop()" function, so this column is not relevant). So when I merged the previously merged DataFrame with the DataFrame for "users.csv", merging these on the user ID, resulted in a new merged DataFrame containing column entries for the user ID, the movie ID, the movie titles, the movie genres, the movie ratings, the users gender, the users age, and the users occupation.

The seventh, eighth, and ninth external functions/methods that are required, are the Pandas ".value_counts()", ".index", and ".isin()" methods respectively. The ".value_counts()" method was used to count the number of unique users (i.e. unique user IDs) in the final merged DataFrame. The ".index" method was used to return only the IDs of certain users, for instance when I was finding the active users in the DataFrame by isolating the users in the DataFrame who had submitted 700 or more ratings. And finally, the ".isin()" method was used to reduce the DataFrame to hold only the entries of users whose IDs matched those of the active users.

The tenth and eleventh external functions/methods that I will discuss, were not imported from any outside modules, but are rather default Python functions/methods. But since these are still external functions/methods in the sense that I did not define them, I will discuss them in this section as well. These functions are "dict()", and "zip()". The ".zip()" function was used to pack the movie IDs and their corresponding movie titles together into a pair, and then ".dict()" was used on that zipped pair in order to create a key, value mapping of movie IDs to their corresponding movie titles.

The twelfth external function/method that is required, is also a default Python method, which is the ".replace()" function. This was used for data preprocessing, specifically in regards to the genre entries. For movies with multiple genres, each genre was separated by a '|', but in order for the genre entries to be formatted properly when served as input to create a TF-IDF matrix which will be discussed later in this report, the genres had to be separated only by a single space ' ', so the ".replace()" method was used to replace '|' with ' ' for all genre entries.

The thirteenth external function/method that is required, is the Scikit-Learn "TfidfVectorizer" function. This function was used to create a TF-IDF vectorizer object imported from the Scikit-Learn library "sklearn.feature_extraction.text". TF-IDF stands for "Term Frequency-Inverse Document Frequency". The fourteenth external function/method that is required, is directly related to "TfidfVectorizer", and is the ".fit_transform()" method. This method converted each genre (which is of string datatype) into a sparse matrix, where the rows equal the movies, the columns equal the unique genre tokens, and the values equal the TF values, which are the frequency of each genre in that movie's genre list.

The fifteenth, sixteenth, and seventeenth external functions/methods that are required, which are still part of the TF-IDF Vectorizer section, are ".toarray()", "Dataframe()", and

“.get_feature_names_out()”, respectively. Though these functions and methods were used for the purposes of testing and debugging. The “.toarray()” method converts the sparse matrix into a complete NumPy array for better visualization. The “Dataframe()” function creates a readable DataFrame, where the rows are indexed by the movie titles, the columns are indexed by the genre tokens, and the values represent the term frequency of each genre for each movie. The “.get_feature_names_out()” method retrieves the list of genre terms that are used as column headers.

The eighteenth external function/method that is required, is the Scikit-Learn “cosine_similarity()” function. As the name suggests, this function calculates the cosine similarity between the rows of a given matrix, in this case the matrix being the TF-IDF matrix. Cosine similarity is a measurement metric that measures the angle between two vectors in a multi-dimensional space. In the case of implementing content-based filtering, that angle determines how similar two movies are based on their genre vectors.

The nineteenth, twentieth, and twenty-first external functions/methods that are required, are Pythons “input()”, “.strip()”, and “.lower()” functions/methods. The “input()” function prompts the user to enter an input through the terminal. The “.strip()” method is used to remove any leading or trailing whitespace from the input string, ensuring that extra spaces will not affect how the answer is read and interpreted. The “.lower()” method converts the entire input string to lower case, eliminating any concern for upper or lower case letters being interpreted differently.

The twenty-second external function/method that is required, is Pythons “.values” method. This is used to retrieve the values from a dictionaries key-value pairs. The twenty-third external function/method that is required, is the Python “float()” function, which is used to ensure that the value that it is taking as input, is converted into a floating-point datatype.

The twenty-fourth external function/method that is required, is the Pandas “.loc()” method. This method can be used to access rows and columns by their names, rather than by their indices or positions. This is a very useful tool in order to select entries from a DataFrame when you only know the labels or names of what you’re looking for, so you can search by their labels or names only, without having to know or worry about their indices or positions.

The twenty-fifth external function/method that is required, is the Pandas concat() function. This is used to append new data to an existing DataFrame. In the case of this project, it is used to add new entries to the merged DataFrame, when a user submits a new movie review. To be more precise, it adds an entry to the merged DataFrame containing the user’s UserID, the MovieID of the movie that they watched, that movie’s corresponding title, the genre or genres of the movie, and the rating that the user gave to the movie.

The twenty-sixth external function/method that is required, is the Python “print()” function. This is a straightforward enough function. It prints or produces an output via the terminal, of whatever we want the output to say. This can be a static message, or it can contain certain information from variables or other forms of information that is available. This can be used both for testing/debugging, as well as giving messages to the user themselves.

The twenty-seventh and twenty-eighth external functions/methods that are required, are the Python “list()” and “enumerate()” functions. The “enumerate()” function takes an iterable (such as a list) as its parameter input, and returns (key, value) pairs. The “list()” function can the be used to convert the enumerate object into a list of tuples. More on the exact implementation of this within the scope of this project will be discussed in the next section.

The twenty-ninth and thirtieth external functions/methods that are required, are the Python “sorted()” function, and the Pandas “.iloc()” method. The “sorted()” function returns a new sorted list from the items of any iterable, such as a list or tuple. It is not modify the original list, unlike the “.sort()” method which works in-place. The “.iloc()” method is used to access a DataFrames rows and columns by their index or position, not by their name or label. So in essence, “.iloc()” works in the opposite way of how “.loc()”, which was discussed earlier works.

The thirty-first external function/method that is required, is the Python “.format()” method. This is used to insert variables into a string. For example, when outputting a string, you can insert placeholder variables using curly braces, and then the parameters passed as input to “.format()” will be what replaces those placeholder variables.

The thirty-second, thirty-third, and thirty-fourth external functions/methods that are required, is the Python “.random()” method imported from Python’s “random” module, the Python “set()” function, and the Python “.choice()” method, also imported from Python’s “random” module. The “.random()” method generates a random floating point number between 0.0 and 1.0. The “set()” function is used to remove duplicate items and to make set operations easier. The “.choice()” method is used to pick one random item from a list.

The thirty-fifth external function/method that is required, is the Pandas “.pivot()” method. The”.pivot()” method is used to reshape a given DataFrame by transforming the data into a matrix, where one column becomes the matrix rows (i.e. the indices), another column becomes the matrix columns, and a third column provides the matrix values. More detail on the exact implementation in regards to this project, will be provided in the next section.

The thirty-sixth, thirty-seventh, and thirty-eighth external functions/methods that are required, are the NumPy “.mean()” method, the NumPy “.sub()” method, and the Pandas “.fillna()” method. The “.mean()” method is used to calculate the average or the mean along a specific axis of a given array or matrix. The “.sub()” method is used to perform a broadcasted subtraction between two elements across two arrays or matrices. The “.fillna()” method is used to replacing missing values with whatever value is passed as a parameter input into the method itself.

The thirty-ninth and fortieth external functions/methods that are required, are the Scikit-Learn “NearestNeighbors()” function, and the Scikit-Learn “.fit()” method. The “NearestNeighbors()” function is used to implement the K-Nearest Neighbors (KNN) algorithm for the collaborative-based filtering model. A distance or similarity metric, as well as a specific algorithm approach can be passed as parameters into the “NearestNeighbors()” function for its exact implementation. The “.fit()” method is then used to train the model on the given input data, which in the case of this project was all stored in the normalized user-item matrix, which will be discussed more in detail in the next section. This model doesn't learn weights like other models, such as linear regression. Instead, it simply stores the data so that it can be later used to compute distances between any input vector and all the rows in the given dataset. Internally, it builds a structure to allow for a fast lookup of the k-nearest neighbors when another method is called later on.

The forty-first, forty-second, and forty-third external functions/methods that are required, are the Numpy ".reshape()" method, the Scikit-Learn ".kneighbors()" method, and the NumPy ".flatten()" method. The ".reshape()" method is used to modify the shape of an array or matrix, without changing the actual data itself. So this allows for example a 1-dimensional array to be reshaped into a 2-dimensional array or matrix. This is a very useful tool when preprocessing data, so that it is of acceptable format to serve as input for another function or method. The ".kneighbors()" method calculates the k-nearest neighbors to the given input vector, using the cosine similarity distance metric. The ".flatten()" method is used to turn a 2-dimensional array or matrix back into a 1-dimensional array.

The forty-fourth, forty-fifth, and forty-sixth external functions/methods that are required, are the Pandas ".groupby()" method, the Pandas ".sort_values()" method, and the Python ".get()" method. The ".groupby()" method allows for the grouping of rows in a given DataFrame, based on the labels of the column or columns passed as parameter inputs. The ".sort_values()" method is used to sort the given values of the DataFrame type object that it is operating on. The sorting can be in either ascending or descending order, and this can be specified by passing in a parameter which will indicate it. The ".get()" method is used to retrieve the value associated with a given key, and if the key is not found, then it allows for a default value specified by the second parameter to be returned instead.

The forty-seventh, forty-eighth, and forty-ninth external functions/methods that are required, are the Python "len()" function, the Python "int()" function, and the Pandas ".empty()" method. The "len()" function will return the size of the object that is passed to it as input, or in other words, it will return the number of items or elements that the object contains. This is useful, for example, in determining how many elements are in a list. The "int()" function is used to ensure that the value that it is taking as input, is converted into a integer datatype. The ".empty()" method simply checks if the Series or DataFrame that it is operating on is empty or not, and it returns true if it is empty, and false if it is not empty.

The fiftieth and fifty-first external functions/methods that are required, are the Pandas ".tail()" method, and the Pandas ".copy()" method. The ".tail()" method is the opposite of the ".head()" method discussed earlier in this report. Whereas the ".head()" method by default shows the first 10 rows in a given DataFrame, the ".tails()" method by default shows the last 10 rows in a given DataFrame. Though as with the ".heads()" method, the ".tails()" method can show any specific number of rows, if that specific number is passed as an input parameter into the method. The ".copy()" method is used to create a duplicate of the DataFrame or Series that it is operating on. This is useful when wanting to modify or manipulate data, without modifying or manipulating the original DataFrame or Series.

The fifty-second and fifty-third, and fifty-fourth external functions/methods that are required, are the Pandas ".map()" method, the Pandas ".rename()" method, and the Pandas ".max()" method. The ".map()" method is used to replace each value in a given Series or DataFrame, with another value. The ".rename()" method is used to rename specific rows or columns in a given DataFrame. The ".max()" method returns a Series with the maximum value obtained from each column.

Design Description

Internal Functions

The first of the internal functions that I created for this project, is the function to run the interface shown to the user or tester. This function is called "get_recommendations_for_user", and takes as parameters the ID of the current user, which has already been assigned to the variable "user_id" and is denoted by the same variable within the functions parameters, the current version of the merged DataFrame, denoted by "datasetReduced", the function name for the content-based filtering model, denoted by "content_based_model", and the function name for the collaborative-based filtering model, denoted by "collaborative_based_model". The first thing that this function does, is checking how many movie rating reviews that the current user has submitted, and it checks this by looking up their user ID within the merged DataFrame "datasetReduced", to see how many entries are recorded for this user. As each time a user submits a rating, a new entry containing their user ID is concatenated to "datasetReduced". The function then creates a local variable called "num_reviews", which holds the length or the size of the number of entries in "datasetReduced" that are contained for that specific user. To rephrase this in perhaps more simple terms, "num_reviews" holds an integer value corresponding to the number of times that the user has submitted a review. After this, an if statement checks the number of reviews that the user has submitted, and if it is below a certain threshold, then the content-based filtering model is used to generate recommendations for the user, and if the number of reviews that the user has submitted is at or above that certain threshold, then the collaborative-based filtering model is used to generate recommendations for the user. This specific threshold is completely arbitrary, and can be adjusted as a developer sees fit.

The second of the internal functions that I created for this project, is the function to create the content-based filtering model, which implements content-based filtering. This function is denoted is "content_based_model" and takes as parameters the users ID, denoted by "user_id", the original DataFrame created for "movies.csv", denoted by "datasetMovies", the final merged DataFrame, denoted by "datasetReduced", the calculated cosine similarity for the TF-IDF vectorization matrix, denoted by "cosine_sim", and the number of reviews submitted by the current user, denoted by "num_reviews". This function also has a default parameter denoted by "top_n=10", which is used to specify the number of nearest neighbors for the K-Nearest Neighbors algorithm.

The "content_based_model" function starts by checking if the user is a completely new user by checking if the number of movie reviews that they have submitted is 0. If this check evaluates to true, meaning that this is indeed a brand-new user with zero prior history in the database, they will be prompted with a message stating: “You haven’t reviewed any movies yet. Would you like to search for a movie to watch? (yes/no):”. If the user selects “yes”, then the system will prompt the user for the name of the movie that they would like to watch, by outputting: “Enter the name of the movie that you’d like to watch:”. In case that movie is not in the “datasetMovies” DataFrame, the program will simply output: “Movie not found in the dataset.” and the function will return. If the user instead selects “no”, then the system will randomly choose a movie from the “datasetMovies” DataFrame for the user to watch. This “yes” or “no” prompt, ensures that no matter what, the user will be given a cold start. After the user watches their first movie, whether by them choosing the movie, or the program selecting a random one for them, they will then be asked to rate the movie, on a scale from 0.0 – 5.0. If they give a rating outside of this range, a message will be outputted saying “Invalid rating. Please enter a number between 0.0 and 5.0” and the function will return. In the case of the input not being a number, a ValueError will be thrown, with a message outputting: “Invalid input. Please enter a number.”. If the rating is valid, the MovieID for the selected movie will be retrieved from “datasetReduced”, a check will occur to see if the user has already rated the movie previously or not, and if they have, instead of creating a new entry for this movie, their previous rating will instead be overwritten with the new rating, and a message will be outputted indicating this, by stating: “Your rating for {movie_title} has been updated to {rating}/5.0”. If the user had not previously rated this movie, a new DataFrame entry will be created, with the user’s ID, the movie’s ID, the movie’s title, the genre or genres of the movie, and the rating that the user has just submitted. This new entry will then be appended to the merged DataFrame “datasetReduced”. A message will then be output to the user stating: “Your rating has been recorded successfully! User {user_id} rated ‘{movie_title}’ with a score of {rating}/5.0”.

In the case where the user had already submitted at least 1 review when starting the function, the else clause will instead execute, where the user will instead be prompted to enter the name of a movie from their previous set of recommendations. A prompt will be outputted to the user for this, stating: “Enter the name of the movie that you’d like to watch:”. Once the user as input the movie name, a check will first happen to ensure that the movie exists in the “datasetMovies” DataFrame. In case it does not exist, a message will be outputted to the user stating: “Movie not found in the dataset” and the function will return. If the movie does exist, then instead the user will be prompted to rate the movie, with an output stating: “Enter your rating for ‘{movie_title}’ (0.0-5.0):”. In case the rating is outside of this range, a message will be outputted stating: “Invalid rating. Please enter a number between 0.0 and 5.0”, and the function will return. If the rating is valid, then as with before, the MovieID for the selected movie will be retrieved from “datasetReduced”, a check will occur to see if the user has already rated the movie previously or not, and if they have, instead of creating a new entry for this movie, their previous rating will instead be overwritten with the new rating, and a message will be outputted indicating this, by stating: “Your rating for {movie_title} has been updated to {rating}/5.0”. If the user had not previously rated this movie, a new DataFrame entry will be created, with the user's ID, the movie's ID, the movie's title, the genre or genres of the movie, and the rating that the user has just submitted. This new entry will then be appended to the merged DataFrame "datasetReduced". A message will then be output to the user stating: "Your rating has been recorded successfully! User {user_id} rated '{movie_title}' with a score of {rating}/5.0".

Once the users rating has been successfully recorded and appended to the merged DataFrame "datasetReduced", their recommendations will then generate. Content-based filtering generates recommendations based on the similarity in the features or content of the movies themselves, which in the case of this project are the genres. This section of the function code starts by retrieving the ID or index in "datasetMovies" of the movie that the user had just watched and rated. The cosine similarity function from before will then take the index of that movie, and compute the similarity scores. The similarity scores will then be sorted in descending or non-increasing order. The target movie itself will then be excluded from the recommendations, as obviously the movie most similar to a movie is itself, and the goal is to show and recommend to the user new movies, not the same one that they had just watched and rated. From here, the function will only take the "top_n" recommendations, which was specified in the default parameter of this function by "top_n=10", meaning that only the top 10 recommendations will be shown to the user. But of course, this is an arbitrary number that can be changed or modified as a developer sees fit. A list of these top 10 recommendations will then be created, as well as a list of their similarity scores. These top 10 recommendations, along with their similarity scores, will then be outputted and displayed to the user, with a message stating: "Here are the top {top_n} recommendations for {movie_title}:", with "movie_title" being the name of the movie that the user had just watched and rated. Lastly, a random number between 0.0 and 1.0 will be generated, and if it less than a certain threshold, for example 0.1 (meaning 10% of the time), then an extra random and unrelated movie will be recommended to the user, for the purpose of overcoming the overspecialization problem. If the randomly generated number is within the threshold, the function will then retrieve a set of all the movie titles in the database, will exclude from this list, the movies that were just recommended, in addition to the movie that was just watched and rated, and then out of the list containing the remaining eligible movie titles, will recommend to the user a random movie from amongst that list. This movie will be outputted with a message stating: "Bonus Recommendation (Random pick): {bonus_movie}". The function will finally then return.

The third of the internal functions that I created for this project, is the function to create the collaborative-based filtering model, which implements collaborative-based filtering. This function is denoted by "collaborative_based_model" and takes as parameters the users ID, denoted by "user_id", the final merged DataFrame, denoted by "datasetReduced", and the dictionary mapping of movie IDs to movie titles, denoted by "movie_mapping". This function has a default parameter denoted by "n_neighbors=10", which is used to specify the number of nearest neighbors for the K-Nearest Neighbors algorithm.

The "collaborative_based_model" function starts by creating a User-Item Interaction Matrix, with the rows being the user IDs, the columns being the movie IDs, and the values being the ratings that user X has given to movie Y. The mean or average of a user's ratings is then calculated, followed by a normalization of this data, in order to minimize the effects of outlier entries. Once the normalized user-item interaction matrix has been created, the model for collaborative-based filtering using K-Nearest Neighbors is then created. More information regarding the User-Item Interaction Matrix can be found in [8], and the specific case of a User-Movie Rating Matrix, can be found in [7]. The model is then fitted or trained on the data from the normalized user-item interaction matrix. In case the ID of the current user is not in the user-item interaction matrix, a message will be outputted to the user, stating: "User {user_id} is not found in the dataset.". The function will then return.

Next, in order to find the most similar users to the current user, an input vector of the current user is created from the normalized user-item interaction matrix, which is then fed into the K-Nearest Neighbors model, along with desired the n neighbors, which was defined as a default parameter to the function as "n_neighbors=10", meaning that here, only the top 10 most similar users to the current user will be found. But of course, this is an arbitrary number that can be changed or modified as a developer sees fit. The IDs of these most similar users is then shown to the current user, along with their corresponding similarity distance, which was found using cosine similarity.

Once the most similar users have been found and displayed to the current user, it is then time to find the movies that will be recommended to the current user. This is done by filtering through the movie ratings that the most similar users to the current user have made, and aggregating the average rating per movie among them. Since the point of collaborative-based filtering is to recommend new movies to the user, we don't want to recommend movies that the user has already watched, so these already watched movies need to be excluded first. This is done by getting the movie IDs of the movies that the current user has already watched and submitted ratings for, filtering them out to where only the unseen and unrated movies are left, and to then select the top 10 movies from amongst this list, based on the average ratings of the movies. The "movie_mapping" dictionary from earlier will then be used to convert the movie IDs to the corresponding movie titles, and a list of the top 10 recommendations is built. These recommendations are then printed or outputted to the user, with a message stating: "Recommendations for User {user_id} based on similar users:", followed by the list of recommendations with the movie title followed by the average rating, which is in the following format: "{i}: {movie_title} (Predicted rating: {rating:.2f})".

Similar to what was implemented for content-based filtering, a random number between 0.0 and 1.0 will be generated, and if it less than a certain threshold, for example 0.1 (meaning 10% of the time), then an extra random and unrelated movie will be recommended to the user, for the purpose of overcoming the overspecialization problem. If the randomly generated number is within the threshold, the function will then retrieve a set of all the movie IDs in the database, will exclude from this list, the movies that were just recommended, in addition to the movie that was just watched and rated, and previous movies that the current user has watched and rated, and then out of the list containing the remaining eligible movie titles, will recommend to the user a random movie from amongst that list, first by working with the movie IDs, and then using the "movie_mapping" dictionary to convert this movie ID to the corresponding movie title. This movie will be outputted with a message stating: "Bonus Recommendation (Random pick): {bonus_movie_title}", followed by another output with a message in the format of: "{len(recommendations)+1}: {bonus_movie_title} (Bonus)".

Once the list of recommendations has been generated and shown to the user, it is then time for the user to select the recommended movie that they would like to watch and rate. The user now selects the movie that they would like to watch and rate, and this functionality supports both cases to where a bonus movie is recommended and to where a bonus movie is not recommended. In case the user tries to select an invalid movie, a message will be outputted to the user stating "Invalid choice. Please select a valid number", indicating the number associated with the movie (i.e 1-10 in the case of no bonus recommendation or 1-11 in the case of a bonus recommendation) in the generated list of recommendations. If this error message is shown, the function will simply return. If the movie choice was a valid selection from the generated recommendation list, the user will be prompted to enter the rating that they want to give the movie, with a message being outputted stating: "Enter your rating for '{selected_movie_title}' (0.0 – 5.0):". If the rating is outside of this range, an error message will be outputted to the user stating: "Invalid rating. Please enter a number between 0.0 and 5.0". The function will then return. If the entered rating is valid, the UserID and MovieID for the current user and selected movie will be retrieved from "datasetReduced", a check will occur to see if the user has already rated the movie previously or not, and if they have, instead of creating a new entry for this movie, their previous rating will instead be overwritten with the new rating. If the user had not previously rated this movie, a new DataFrame entry will be created, with the user's ID, the movie's ID, the movie's title, the genre or genres of the movie, and the rating that the user has just submitted. This new entry will then be appended to the merged DataFrame "datasetReduced". Once the previous rating has been updated or a new rating has been appended, a message will then be output to the user stating: "Your rating has been recorded successfully! User {user_id} rated '{selected_movie_title}' with a score of {rating}/5.0". The function will finally then return.

The fourth of the internal functions that I created for this project, was a function to help with testing, debugging, and measuring model accuracy, which was called "get_user_ratings". This is a simple enough function, to where its purpose is to retrieve all the entries in the merged DataFrame "datasetReduced" of a given user. This function will pull all entries of a user's ID, and return the movie ID and movie rating associated with each of that particular user's entries. This was used to look at the movies that a user's most similar users had watched and rated, and to see how well they matched up with the recommendations generated by collaborative-based filtering.

The fifth of the internal functions that I created for this project, was a function that had the same functionality as "get_user_ratings" that was just mentioned, but this function, called "get_user_ratings_with_titles" returns the movie titles instead of the movie IDs. The purpose of this function, was to make debugging quicker and easier, due to not having to manually look up the movie titles corresponding to the movie IDs each time.

The sixth of the internal functions that I created for this project, was a function called "get_common_rated_movies", which takes two users, and outputs all the common movies that the two have watched and rated. This was used when analyzing the accuracy of the collaborative-based filtering model, by comparing the current user with the nearest neighbors generated by the collaborative-based filtering model, and to see how similar they truly were, by looking at the number of common movies that they had watched and rated, and to see how close their ratings for these common movies actually were.

Architecture

To start the project, the required libraries are first imported. Once this has been done, the project then imports the needed datasets in csv format by using the Pandas "read_csv()" external function, which contains all the initial data and information used for the project. The imported csv files were "movies.csv", "ratings.csv", and "users.csv". In the datasets, since what should be separate column entries was compressed into one column, with a double colon "::" separating what should be each category, The "sep" parameter of "read_csv()" is used and set equal to "::", in order to create separate column entries, wherever "::" was seen. And then the "names" parameter of "read_csv" is used to create the appropriate column names for the DataFrame of each imported dataset. Additionally, the "engine" parameter of "read_csv()" is set equal to "python", as this was the programming language that is used for this project. And lastly, ".dropna()" is used to remove any rows or columns with missing values.

Once the datasets were successfully imported, a check is done to make sure that the columns were separated and named correctly. To do this, the Pandas ".shape" method is first used, to print the number of row and columns of "datasetMovies", followed by using the Pandas ".head()" method on "datasetMovies", to output the first 5 rows of the DataFrame. Similarily, the same ".shape" and ".head()" methods are used on the DataFrames for "datasetRatings" and "datasetUsers", to verify that their columns were also separated and named correctly.

Once the formatting of these initial three DataFrames has been verified, the datatypes of each column in the three DataFrames is set to a specific datatype, to ensure full compatibility and intended functionality with the rest of the project. In "datasetMovies", 'MovieID' is set to be 'int32', 'Title' is set to be 'str', and 'Genres' is set to be 'str'. In "datasetRatings", 'UserID' is set to be 'int32', 'MovieID' is set to be 'int32', and 'Rating' is set to be 'float32'. Since 'Timestamp' is not needed for the purposes of this project, this column was removed from the DataFrame entirely, using the Pandas ".drop()" method. In "datasetUsers", 'UserID' is set to be 'int32', 'Gender' is set to be 'str', 'Age' is set to be 'int32', and 'Occupation' is set to be 'int32'. Since 'Zip-code' is not needed for the purposes of this project, this column was removed from the DataFrame entirely, using the Pandas ".drop()" method. While with the exception of 'UserID', I did not end up using any of the other columns of "datasetUsers", I decided to still keep all of them except for 'Zip-code' in the DataFrame, as I believe that they will be useful when making improvements to this project in the future.

Once the specified datatypes have been set, it is time to merge the three separate DataFrames into one big DataFrame containing all the relevant information of the initial three datasets. This is done using the Pandas ".merge()" method in two steps. Since ".merge()" can only merge two DataFrames at a time, "datasetMovies" and "datasetRatings" are first merged together, creating "movieRatingsMerged", and this is then merged with "datasetUsers", creating "datasetMerged". After each step, ".shape" and ".head()" is used to verify that the data entries are as expected.

Since the size of "datasetMerged" is too large, to the point that the entire system would crash when trying to run certain operations on the DataFrame, reducing the size of "datasetMerged" was required. A few different approaches were considered, such as filtering by keeping only the entries of the most popular movies, removing some entries at random, etc. The approach that I ultimately decided on, was filtering the dataset by keeping only the entries of the most active users, which after some experimentation, had to be users with at least 700 reviews submitted. If the threshold was lower, crashes would still occur. So a new DataFrame was created, only containing the entries of users who had submitted at least 700 reviews, with this DataFrame being "datasetReduced". In order to verify that the format and structure of the DataFrame was still intact, ".shape" and ".head()" were used.

Once the merged DataFrame is finally ready, a dictionary mapping is then created, mapping movie IDs to their corresponding movie titles. This is useful in later sections of the project, where certain operations are much more easily implemented using the movie IDs, but other operations, the outputting of information to the user, and testing and debugging, are more easily done when you have the movie titles instead, and so it is very helpful to have this dictionary mapping for easy conversion between the two.

In order to implement the content-based filtering model, the first thing that needs to be done is to complete any final data preprocessing that is still needed. Since the content-based filtering model is working with the genres of the movies, the formatting of the movie genres will need some slight modification from their current form. Since this model is using TF-IDF vectorization, the movies with multiple genres, will need their genres separated by a space ' ' before being served as input to the TF-IDF model. At this point, movies with multiple genres, have their genres separated by a vertical line '|' , which is how they were formatted in the original datasets. The ".replace()" method is used here to replace each '|' with a space ' ' in each genres entry. The ".head()" method is then used to check and ensure that the replacement was successful in the DataFrame. It should be noted that "datasetMovies" is being used here, rather than "datasetReduced". This is because in "datasetReduced", the same movies will show up repeatedly, due to the multiple ratings for those movies existing. Whereas in "datasetMovies", each movie shows up only once, and so this design decision was decided in order to resolve a bug that was occurring in when debugging the function for the content-based filtering model, where the same movie was showing up repeatedly in the generated recommendations.

The next step involves creating the TF-IDF vectorization matrix. This approach will convert the movie genres into a numeric matrix by using a bag-of-words approach, which will then be used to implement content-based filtering. A "TfidfVectorizer" object is created, followed by a model that learns the genre vocabulary from the DataFrame "datasetMovies", to where then a TF-IDF matrix is created, with the rows representing the movies, the columns representing the genres, and the values representing how often each genre appears for each movie, which in other words, is the term frequency. Each genre string here is transformed into a one-hot encoding styled vector. For easier analysis, a print statement then outputs the matrix in tabular form. The pairwise cosine similarity between all movies is then computed based on their genre vectors. The "cosine_similarity" function takes the TF-IDF matrix and creates a square matrix, to where each row i and column j gives the cosine similarity between movie i and movie j. More information regarding TF-IDF vectorization and its applicability to content-based filtering can be found in [6].

The function for the content-based filtering model is then created and implemented. A detailed description of this function has been given in the "Internal Functions" subsection of this same section.

The function for the content-based filtering model is then created and implemented. A detailed description of this function has also been given in the "Internal Functions" subsection of this same section.

A few functions for testing and debugging were then created and implemented, to where detailed descriptions of these functions have also been given in the "Internal Functions" subsection of this same section.

The function that serves as the main user interface is then created and implemented. The function, which is called "get_recommendations_for_user", also has a detailed description of it that is available in the "Internal Functions" subsection of this same section.

Once all these functions have been loaded, the program will start by prompting the user to enter their user ID, with a message being outputted stating: "Enter your User ID (or press Enter if new:)". If the user is a returning user, then they can simply input their user ID, and a message will be outputted to them in the format: "Welcome back, User {user_id}!". If the user is a new user, they can simply press the "enter" button on their keyboard, and a new user ID will be generated and assigned to them, with a message being outputted of the format: "New user detected. Assigning User ID: {user_id}". Finally, "get_recommendations_for_user" will be called, which handles the operations for the entire system.

Implementation

Organization of Source File Structure

The organization of the source file structure is pretty straight forward. Everything is contained within one ".ipynb" file, with the different segments of the project being next to each other. For example, the first section of the project contains the library imports, with the second section of the project containing all the data preprocessing that is required. The file structure then transitions into the third section of the project, which is for content-based filtering. The fourth section of the project is for collaborative-based filtering. The fifth section of the project is for testing and debugging. Finally, the sixth section of the project is for the user interface and the actual launch of the system from the user-end.

Reference List of Files

The list of files for this project consists of four files, which are: the source code file, denoted by “CPSC_597_Recommender_System.ipynb”, and the three data files, which are from the GroupLens dataset, are denoted by “movies.csv”, “ratings.csv”, and “users.csv”, respectively. The GitHub repository can be found at the following link: https://github.com/shayandarian/CPSC-597-Project.

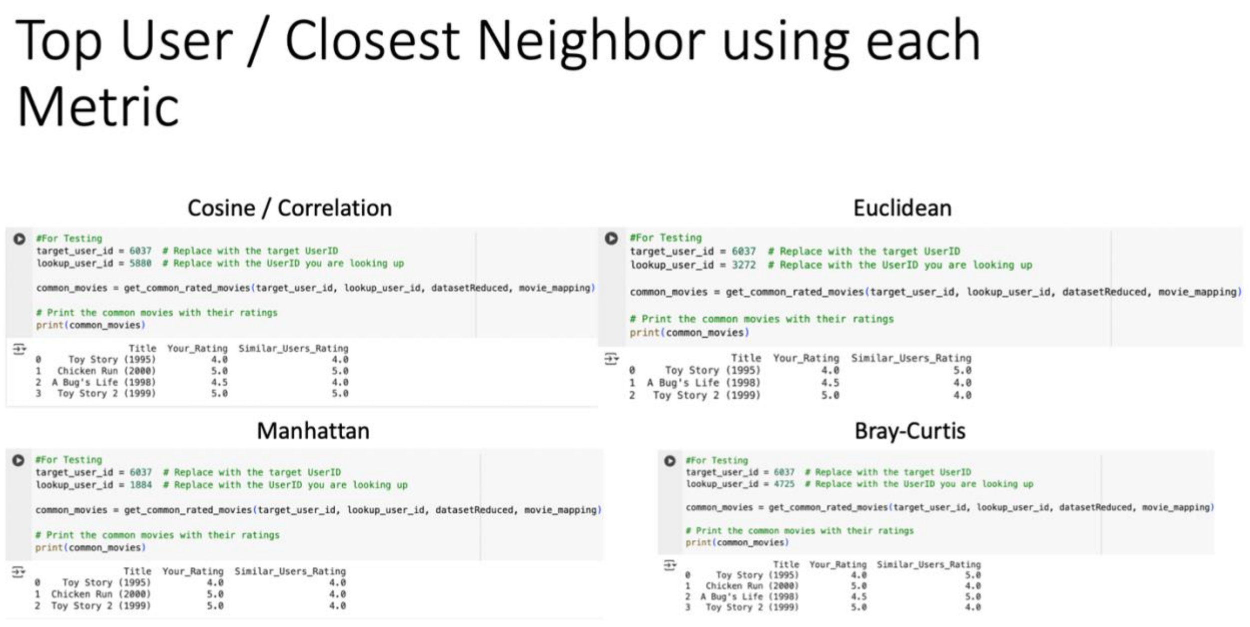
Test and Integration

Testing

The testing for this project was done in two different stages. The first stage, was to create testing functions, such as “get_user_ratings” and “get_user_ratings_with_titles”, which allowed me to see all the movies that a user has rated. This was useful when testing to make sure that the ratings that a user gives, were successfully added to the merged DataFrame. I also used some external functions, such as Pandas “.tail()” to see the latest entries in the DataFrame, making sure that the formatting and structure of the newly added entries matched all the other DataFrame entries.

The second stage, was to create the testing function “get_common_rated_movies”, and to experiment with different measurement metrics when implementing K-Nearest Neighbors. The function “get_common_rated_movies” allowed me to look at all the movies that two different users had commonly watched and rated, in addition to what ratings that they gave the users. This was essential in being able to measure the accuracy of the collaborative-based filtering model, in terms of how similar the current user actually was in regards to their nearest neighbors, as well as in comparing and contrasting the results of the different similarity metrics that I experimented with.

After experimenting with Cosine Similarity, Euclidean distance, Manhattan Distance, Bray-Curtis Distance, and Correlation Distance, I found that Cosine Similarity and Correlation Distance both produced identical results, which were the most accurate results. A table showcasing the number of common movies between the current user and their nearest neighbor, in addition to the similarity in their ratings, can be seen in the figure below. As you can see, Cosine Similarity and Correlation Distance both produced a closest neighbor to the current user who had rated four of the same movies as the user. In addition, the ratings of each of the four movies were all identical, except for “A Bug’s Life (1998)”, where the difference between the two ratings was only off by 0.5. Euclidean Distance and Manhattan Distance each produced a closest neighbor with whom the current user had only three rated movies in common, and the differences in ratings between these movies was much more varied. The Bray-Curtis Distance did also produce a closest neighbor to the current user to where they had four rated movies in common, but the ratings of each of these movies was much more varied, in comparison to the results produced by the Cosine Similarity and Correlation Distance metrics.



Top User / Closest Neighbor using each Metric

Cosine / Correlation

```
#For Testing
target_user_id = 6037  # Replace with the target UserID
lookup_user_id = 5880  # Replace with the UserID you are looking up

common_movies = get_common_rated_movies(target_user_id, lookup_user_id, datasetReduced, movie_mapping)

# Print the common movies with their ratings
print(common_movies)
```

```
                Title  Your_Rating  Similar_Users_Rating
0    Toy Story (1995)          4.0                   4.0
1   Chicken Run (2000)          5.0                   5.0
2  A Bug's Life (1998)          4.5                   4.0
3   Toy Story 2 (1999)          5.0                   5.0
```

Euclidean

```
#For Testing
target_user_id = 6037  # Replace with the target UserID
lookup_user_id = 3272  # Replace with the UserID you are looking up

common_movies = get_common_rated_movies(target_user_id, lookup_user_id, datasetReduced, movie_mapping)

# Print the common movies with their ratings
print(common_movies)
```

```
                Title  Your_Rating  Similar_Users_Rating
0    Toy Story (1995)          4.0                   5.0
1  A Bug's Life (1998)          4.5                   4.0
2   Toy Story 2 (1999)          5.0                   4.0
```

Manhattan

```
#For Testing
target_user_id = 6037  # Replace with the target UserID
lookup_user_id = 1884  # Replace with the UserID you are looking up

common_movies = get_common_rated_movies(target_user_id, lookup_user_id, datasetReduced, movie_mapping)

# Print the common movies with their ratings
print(common_movies)
```

```
               Title  Your_Rating  Similar_Users_Rating
0   Toy Story (1995)          4.0                   4.0
1  Chicken Run (2000)          5.0                   4.0
2  Toy Story 2 (1999)          5.0                   4.0
```

Bray-Curtis

```
#For Testing
target_user_id = 6037  # Replace with the target UserID
lookup_user_id = 4725  # Replace with the UserID you are looking up

common_movies = get_common_rated_movies(target_user_id, lookup_user_id, datasetReduced, movie_mapping)

# Print the common movies with their ratings
print(common_movies)
```

```
                Title  Your_Rating  Similar_Users_Rating
0    Toy Story (1995)          4.0                   5.0
1   Chicken Run (2000)          5.0                   4.0
2  A Bug's Life (1998)          4.5                   5.0
3   Toy Story 2 (1999)          5.0                   4.0
```

Figure 4 – Comparison of Similarity Metrics

Integration

Integrating the different components of the system, particularly the two different recommendation models, i.e. the content-based filtering model and collaborative-based filtering model, was handled by the "get_recommendations_for_user" function. All reviews that the user submitted, whether it was from the content-based filtering model or collaborative-based filtering model, were appended to the "datasetReduced" DataFrame, ensuring that all the data remained consistent and up to date, regardless of which recommendation model was currently being used. Other smaller segments of the project were integrated into the content-based filtering and collaborative-based filtering models themselves. This resulted in a smoothly transitioning recommender system able to handle all the various components of the project.

Installation Instructions

The installation instructions for this project are pretty straightforward. From the GitHub link in the "Reference List of Files" subsection of the previous section, you will find the code for the project itself, as well as the relevant datasets. You can download all of these files onto your desktop. Next, you should make sure that Visual Studio Code and Python3 are installed on your computer. You will also need to install the needed Python libraries if you haven't already, which are NumPy, Pandas, and Scikit-Learn. If you wish to install these via terminal commands, the exact terminal commands may vary depending on your operating system. If you do not wish to or are not able to successfully install what is required, then the project can also be ran through Google Collab. Go to Google Collab, open the project file via Google Collab, upload the datasets, and then you will be able to run the program without needing to install anything.

Operating Instructions

The operating instructions are also pretty straightforward. Launch Visual Studio Code, and open the file for this project, named "CPSC_597_Recommender_System.ipynb". Since the testing functions and commands are also in this file, each code block will need to be ran one at a time. Start by running the code block for the library imports. You will be prompted to select a Python environment / kernel for this project. Any Python3 kernel will be fine. Next run all the code blocks in order from top to bottom for the data preprocessing section. Do the same for the content-based filtering section, and then the collaborative-based filtering section. Lastly, do the same for the user interface section. The second code block in the user interface section will ask the user to input their user ID, or simply have them press the "enter" button on their keyboard if they are a new user and thus do not have a user ID. From here, everything has been set up for the current session. Whenever the user wants to interact with the system, they simply execute the code block in the "Get Recommendations" section.

Recommendations for Enhancement

For possible future enhancements to this project, here are some that I recommend. The first enhancement, would be modifying the function implementations of the content-based model and the collaborative-based model to support handling the entry of new movies into the system, as the current implementation was built to handle the entry of new users into the system. The concepts, theories, and solutions would be the same, the only difference would be to modify these functions to accommodate this new scenario.

The second enhancement, would be for the implementation of the content-based model. Currently, the content-based model works by analyzing the genres of the movies. But adding additional content features, such as cast, director, movie description, etc. could lead to more complex and stronger results. Though this would require different datasets, as the datasets used for this project lacks these additional content features.

The third enhancement, would be for the implementation of the collaborative-based model. In addition to the ratings that users have given, their personal information, such as age, gender, occupation, etc. could be used as additional parameters when training the model to create the list of the most similar users. A fourth enhancement, that is also for the collaborative-based model, would be to use a deep learning approach, in other words to implement collaborative-based filtering using an Artificial Neural Network (ANN). Discussions revolving around the topic of Deep Learning or Artificial Neural Network implementations for the models of a recommender system can be found in [9] and [10].

A fifth enhancement, would be separating different sections of the code into separate files. This would be beneficial if this project were to actually be deployed, and would allow the code blocks relevant to the user to automatically be ran at once, rather than manually doing each code block one at a time. This would also allow the testing functions to be separate and hidden from the user. A sixth enhancement, related to the user as well, would be to build a front-end web interface, which would keep the back-end hidden from the user completely, and would also enhance the user experience with a much better user interface.

Bibliography

[1] Q. Zhang, J. Lu, and Y. Jin, "Artificial intelligence in recommender systems," *Complex & Intelligent Systems*, vol. 7, no. 1, pp. 439–457, Nov. 2020, doi: https://doi.org/10.1007/s40747-020-00212-w.
[2] S. Malik, "Movie Recommender System using Machine Learning," *EAI Endorsed Transactions on Creative Technologies*, vol. 9, no. 3, p. e3, Oct. 2022, doi: https://doi.org/10.4108/eetct.v9i3.2712.
[3] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-Based Systems*, vol. 46, pp. 109–132, Jul. 2013, doi: https://doi.org/10.1016/j.knosys.2013.03.012.
[4] L.-C. Zhao *et al.*, "Machine Learning Techniques for Recommender Systems -A Comparative Case Analysis," *IOP Conference Series: Materials Science and Engineering*, no. 1085, 2020, doi: https://doi.org/10.1088/1757-899X/1085/1/012011.
[5] S. Airen and J. Agrawal, "Movie Recommender System Using K-Nearest Neighbors Variants," *National Academy Science Letters*, vol. 45, no. 1, pp. 75–82, May 2021, doi: https://doi.org/10.1007/s40009-021-01051-0.
[6] J. Beel, B. Gipp, S. Langer, and C. Breitinger, "Research-paper recommender systems: a literature survey," *International Journal on Digital Libraries*, vol. 17, no. 4, pp. 305–338, Jul. 2015, doi: https://doi.org/10.1007/s00799-015-0156-0.
[7] S. Airen and J. Agrawal, "Movie Recommender System Using Parameter Tuning of User and Movie Neighbourhood via Co-Clustering," *Procedia Computer Science*, no. 218, pp. 1176–1183, 2023, doi: https://doi.org/10.1016/j.procs.2023.01.096.
[8] X. Yang, Y. Guo, Y. Liu, and H. Steck, "A survey of collaborative filtering based social recommender systems," *Computer Communications*, vol. 41, pp. 1–10, Mar. 2014, doi: https://doi.org/10.1016/j.comcom.2013.06.009.
[9] J. Bobadilla, F. Ortega, A. Gutiérrez, and S. Alonso, "Classification-based Deep Neural Network Architecture for Collaborative Filtering Recommender Systems," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 6, no. 1, p. 68, 2020, doi: https://doi.org/10.9781/ijimai.2020.02.006.
[10] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep Learning Based Recommender System," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–38, Feb. 2019, doi: https://doi.org/10.1145/3285029.