Machine Learning and Recommender Systems

Marcus Fitzsimons
N00164071

Supervisor: John Montayne
Second Reader: Cyril Connolly

Year 4 2022-23
DL836 BSc (Hons) in Creative Computing

Abstract

As the field of AI continues to evolve and advance, it becomes an ever more intertwined part of people's daily lives, from browsing the web, to self-driving cars, to recommendation systems and everything in between. The aim of this thesis is to research machine learning and recommender systems, how they can be applied to a movie recommendation application and then document the development such an application.

Recommender system filtering and methodology options were researched to determine the most appropriate models for a movie recommendation system, while similar applications were studied and interview were carried out to help define what kind of functionality people want from a movie recommendation system.

The developed application allows users to search and browse through a collection of movies, view extensive information on each movie and provides them with personalised, AI generated movie recommendations based on collaborative filtering methodology.

Acknowledgements

First and foremost, I need to thank my project supervisor, John Montayne, without whose help this project would have been significantly more challenging. John always made himself available for any problems that could occur, his advice was consistently invaluable and helped me to get over a few hurdles over the course of this project.

I'd also like to thank Mohammed Cherbatji, one of my lecturers at IADT. Mohammed was always there to offer guidance and support in any web development areas.

Finally, I'd like to thank all of the lecturers I've had at IADT throughout this course, in particular Louise Glynn and Anne Wright, without all of their help and support I would not have gotten far enough in this course to even attempt this project.

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student: Marcus Fitzsimons

Signed: M Fitzsimons

Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

1 Chapter One: Research

1.1 Introduction

It can be a challenging process for a business to offer products and services that appeal to individual potential customers, but with the development of machine learning based recommender systems over the last few decades, and the ever-increasing amount of data available, it has never been more achievable than it is now. Over the last few years, AI-driven applications have made huge strides in many fields, such as Deepmind's AI AlphaGo, the first AI program to defeat a professional Go player, the development of self-driving cars and the dramatic improvement of computer vision and speech recognition systems. These advances in the field of AI, along with the ever-growing, massive amounts of people's data being recorded offer significant opportunities to machine learning-based recommender systems to make use of these advancements.

1.1.1 What is machine learning?

Machine learning is a field of artificial intelligence that aims to create systems that use data to automatically improve their performance on specific tasks through experience. Machine learning algorithms use neural networks to build models based on a set of training data to learn how to make the best decisions on future data without being specifically programmed to.

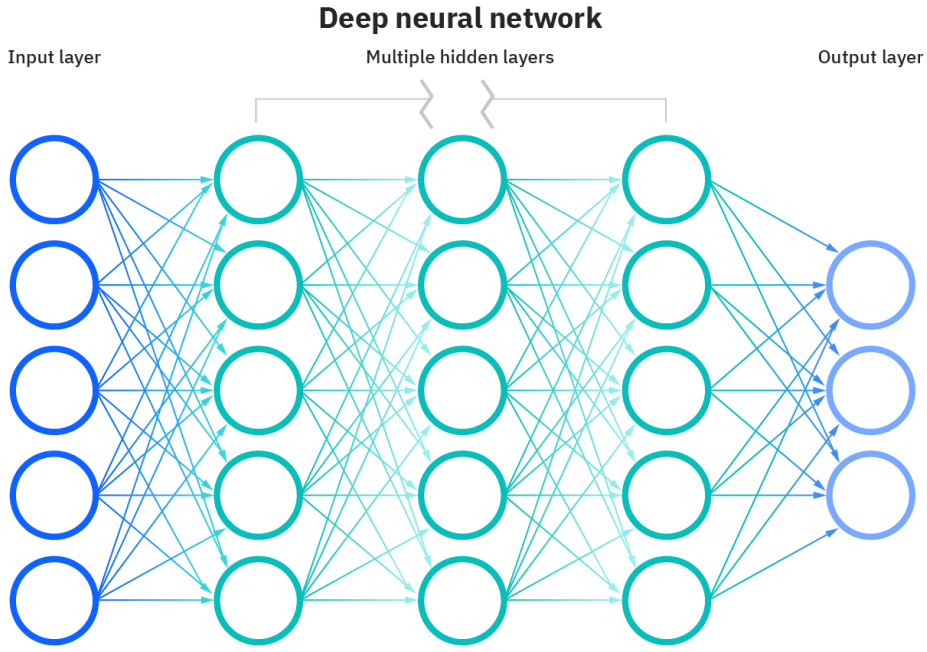


Figure 1: A basic example of the structure of a neural network

1.1.2 What is a recommender system

A recommender system uses a person's interests and preferences in an attempt to recommend something that the system believes would appeal to their preferences. It is not exclusively used by machine learning based systems, but as technology advanced and the field of machine learning along with it, it became clear that AI based recommender systems could achieve heights that would otherwise have been highly impractical, if not impossible to achieve otherwise. With the rapid growth of information that goes through the World Wide Web and the increased globalisation of the world, people are presented with an ever-increasing number of choices, which often results in more complex decision making. Recommender systems can assist people with making decisions in areas where they have little knowledge or information.

1.1.3 How recommender systems work

Recommender systems have primarily evolved down two distinct paths: collaborative filtering and content-based filtering. Collaborative filtering attempts to discern a user's preferences based on the similar preferences of other users in the system, presuming that if many other people with similar preferences liked something, the targeted user likely will also. Content-based filtering is based on knowing as many details or characteristics as possible on the entity being recommended, as well as knowing each user's affinity for each of these details or characteristics, the more information you have on the entity being recommended and the user it's being recommended to, the more likely it is that the recommendation is appropriate.

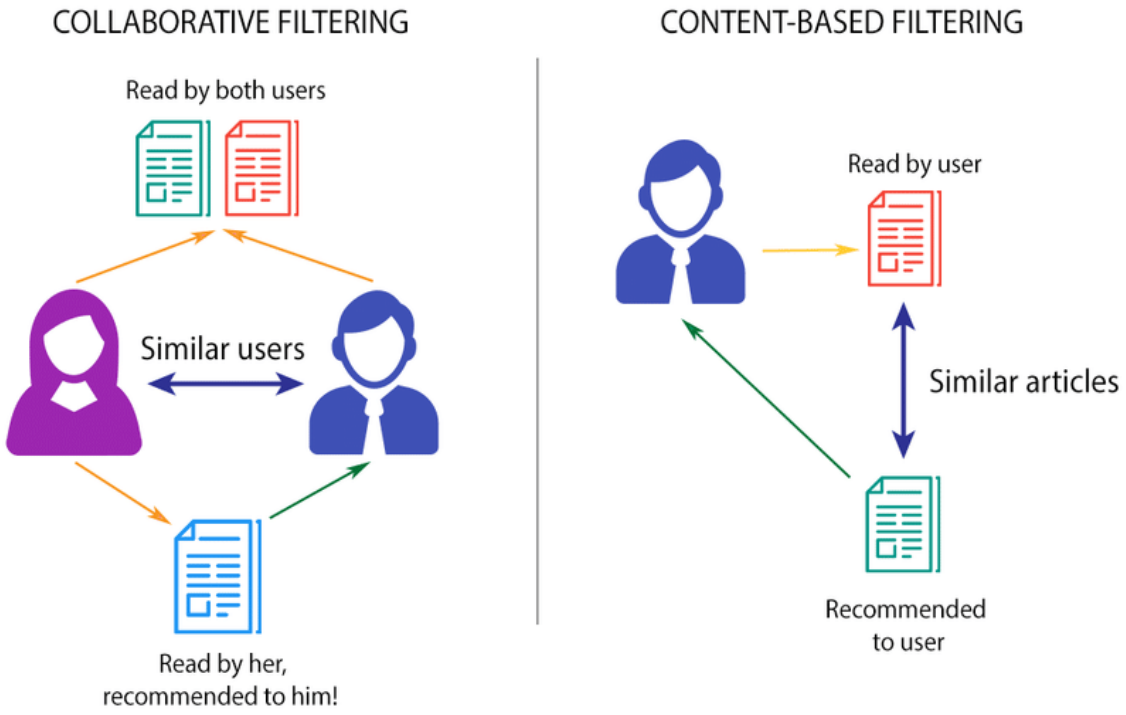


Figure 2: Basic examples of collaborative filtering and content-based filtering logic

1.1.4 A brief history of recommender systems

Humans are social animals, so much of what people do and think is based on other people around them. Recommender systems development began after the simple realisations that people often seek and use recommendations provided by others to help them to make their decisions, and that computer systems could help with generating recommendations. For example, if a person is looking for a movie to watch, or a book to read, it is common for them to look to people they know or experts/people of authority in the sought field for recommendations.

The first known computer-based recommender system was created in 1979 by Elaine Rich, who at the time was an assistant professor at the University of Texas. She created a book recommendation system and dubbed it 'Grundy'. As Rich describes in her personal blog, Grundy used models based on the books and the users, each individual book's information was created and added to the system by hand, while each user was sorted into specific 'stereotypes' based on sets of predefined words provided by the user as a simple self-description. Grundy would then generate book recommendations by comparing its model of the user to the models of the books it knew about, choosing its determined best match and generating a brief description of the book, which included the reasons the system thought it was a good recommendation to the user.

In 1992, developers at the Xerox Palo Alto Research Centre created an experimental mail system to handle large amounts of emails and messages posted to newsgroups in an attempt to allow users to find documents they're interested in. The paper published alongside this system, 'Using Collaborative Filtering to Weave and Information Tapestry', contained the first use of and popularised the term 'collaborative filtering'. They cited the effectiveness of filtering systems of past mail systems, and determined that involving humans in the filtering process created a more effective system. They involved humans in the process by recording their reactions to documents they read and applying their reactions to other user's filters, personalising them (Goldberg, Nichols, Oki & Terry, 1992, pp. 1-2).

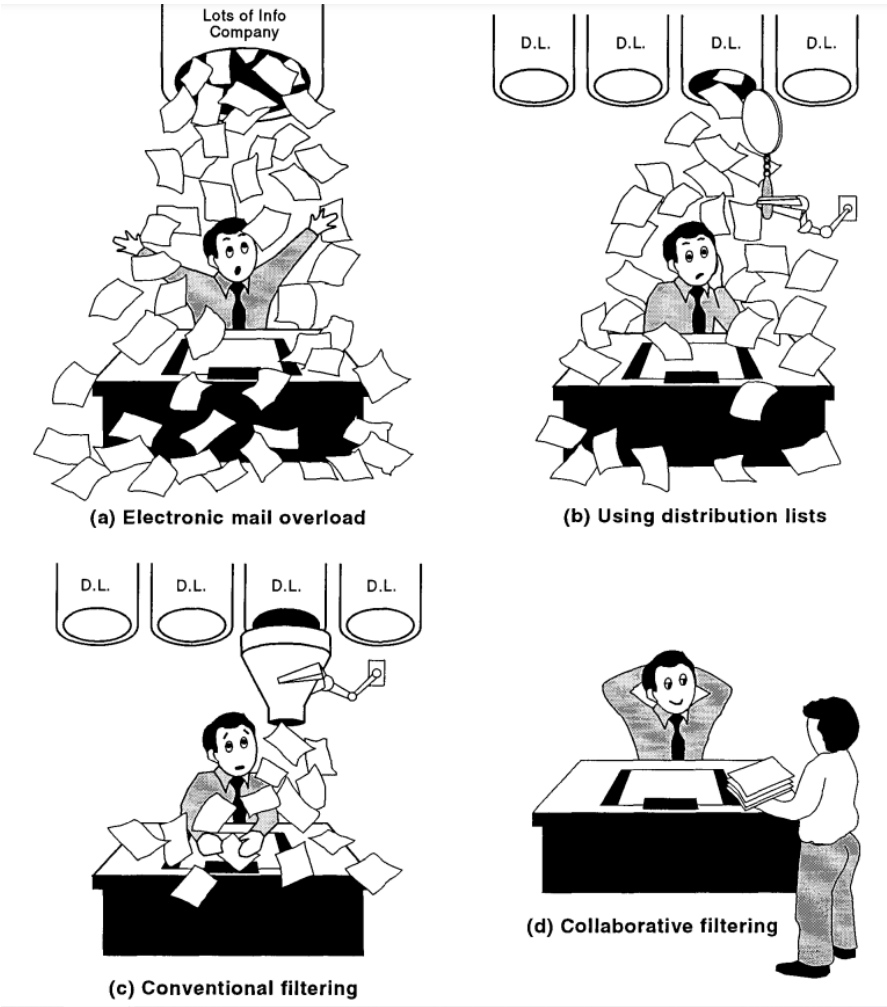


Figure 3: Diagram from 'Using Collaborative Filtering to Weave and Information Tapestry showing the effectiveness of different filtering methods of the time (Goldberg, Nichols, Oki & Terry, 1992, p. 2).

With the rise of the world wide web and e-commerce, the tech industry was beginning to realise the value of recommendations. The first company that focused on offering marketing recommendation systems to other business', Net Perceptions, was founded in 1996 and their customers included the likes of Amazon, Best Buy and JC Penny. These systems helped E-commerce websites to increase sales by analysing aspects such as user interfaces, recommendation models and user inputs. In 1997 the GroupLens research lab developed MovieLens, a movie recommendation system. The early versions of the recommender models were trained using the EachMovie dataset, a dataset containing millions of movie ratings by thousands of users for over a thousand movies. Since then, MovieLens datasets have been continuously released and is one of the more popular datasets used for studying recommendations (Dong, Wang, Xu, Tang & Wen, 2022, p. 1).

Taking the rapid development of the research and practical applications of recommender systems into consideration, the community decided to hold the first Association for Computing Machinery Recommender Systems Conference (abbreviated to ACM RecSys) at the University of Minnesota in 2007. Since then, ACM RecSys has become annual academic conferences focused on the study and development of recommender systems. Around the same time, many studies into the user experience of recommender systems were being published and given more attention (Dong, Wang, Xu, Tang & Wen, 2022, pp. 1-2). Pearl Pu et al. proposed a more user-centric evaluation framework

for recommender systems, arguing that existing applications have suggested a set of criteria that details characteristics that constitute an effective and satisfying recommender system from a user's perspective, and that these criteria can be used to determine perceived qualities of recommender systems. Pu et al. developed a model called 'Recommender systems' Quality of user experience', otherwise known as ResQue. ResQue aimed to assess the perceived qualities of recommender systems such as their usability, usefulness, interface interactions, user satisfaction and the influence of these qualities on user behaviours such as intention to purchase products, intention to recommend products to others and to return to the system in the future.

1.2 Filtering

1.2.1 Collaborative-based filtering

Collaborative-based filtering techniques aim to make recommendations to the active user based on items that other users with similar preferences like or have liked in the past. Similar preferences between users are calculated based on things such as rating, purchase and/or interaction history (Ricci, Rokach & Shapira, 2022, p. 13).

1.2.2 Content-based filtering

A content-based recommender system tracks item data and recommends items that are similar to any items that a user has purchased, liked or even just interacted with in the past. Item similarity is calculated based on the features shared or associated with the compared items (Ricci, Rokach & Shapira, 2022, pp. 12-13). For example, if a user has purchased baby food, the system might recommend other baby related items, such as diapers or baby clothes.

1.2.3 Hybrid-based filtering

A hybrid-based recommender system aims to combine two or more approaches to building recommender systems and draw on the advantages of each approach in use. A hybrid system combining collaborative-based filtering and content-based filtering will try to use the advantages of collaborative filtering to make up for the disadvantages of content-based filtering and vice versa. For example, collaborative filtering systems don't deal with new items very well, they cannot accurately recommend items that have no previous ratings. This is not a problem for content-based systems, as the predictions for new items are based on their typically easily available features rather than interactions with users (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 5-6).

1.3 Methodology

1.3.1 K-Nearest Neighbour (KNN) methodology

K-Nearest Neighbour (KNN) is one of the more basic algorithms and is used in both regression and classification problems (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, p. 6). KNN-based methods focus on relationships between items or between users. An item-based approach determines a user's preferences towards an item based on ratings of similar items by that same user. A user-based approach determines a user's preferences towards an item based on similar user's ratings of that same item.

The algorithm works by assigning a value to K which will define how many neighbours are to be checked to determine the classification of a specific data point. For example, if K=1, a data point will be assigned to the same class as its single nearest neighbour, while if K=5, a data point will be assigned to whichever class most of its nearest neighbours are. A data points nearest neighbours are determined by plotting the data points on a graph and calculating the distance (primarily the Euclidean distance, but other methods for calculating distance, such as Minkowski distance, can be successfully applied also (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 11, 99)) between each data point.

KNN-based methods have gained considerable popularity because of their accuracy, efficiency, simplicity and personalised recommendations. Whether an item-based approach or a user-based approach is more appropriate for a specific system largely depends on the quantity the different areas of information the system has access to. A system that has significantly more users than available items should generally be expected to use an item-based approach to provide the most accurate recommendations, while also being more computationally efficient and requiring less frequent updates. If a system has more items than users, desires more original recommendations or is not as concerned by efficiency, a user-based approach can be used to offer users a more satisfying experience.

User-item relation data to make recommendations can be obtained explicitly, for example, through review and rating systems such as a numerical (e.g., 1-5 stars, 0-100%), ordinal values (e.g., strongly agree, agree, neutral, disagree, strongly disagree) or binary values (e.g. like/dislike, interested/uninterested), but the user-item relation data can also be obtained implicitly through information such as purchase history, or the patterns of how individual users have interacted with the system (Ricci, Rokach, Shapira, 2022, p. 10).

KNN is a robust algorithm that works well with noisy training data and is very effective when used for cases with a large training dataset.

1.3.2 Support Vector Machine (SVM) methodology

Support Vector Machine (SVM) is a supervised machine learning algorithm that analyses data for classification and regression analysis. In SVM, the primary goal is to find the hyperplane (A hyperplane is a decision boundary that differentiates two classes in SVM, a data point landing on either side of hyperplane can be attributed to a different class, the goal is to achieve the maximum

sized margin between the two classes) in the N-dimensional space with the largest margin (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 8-9).

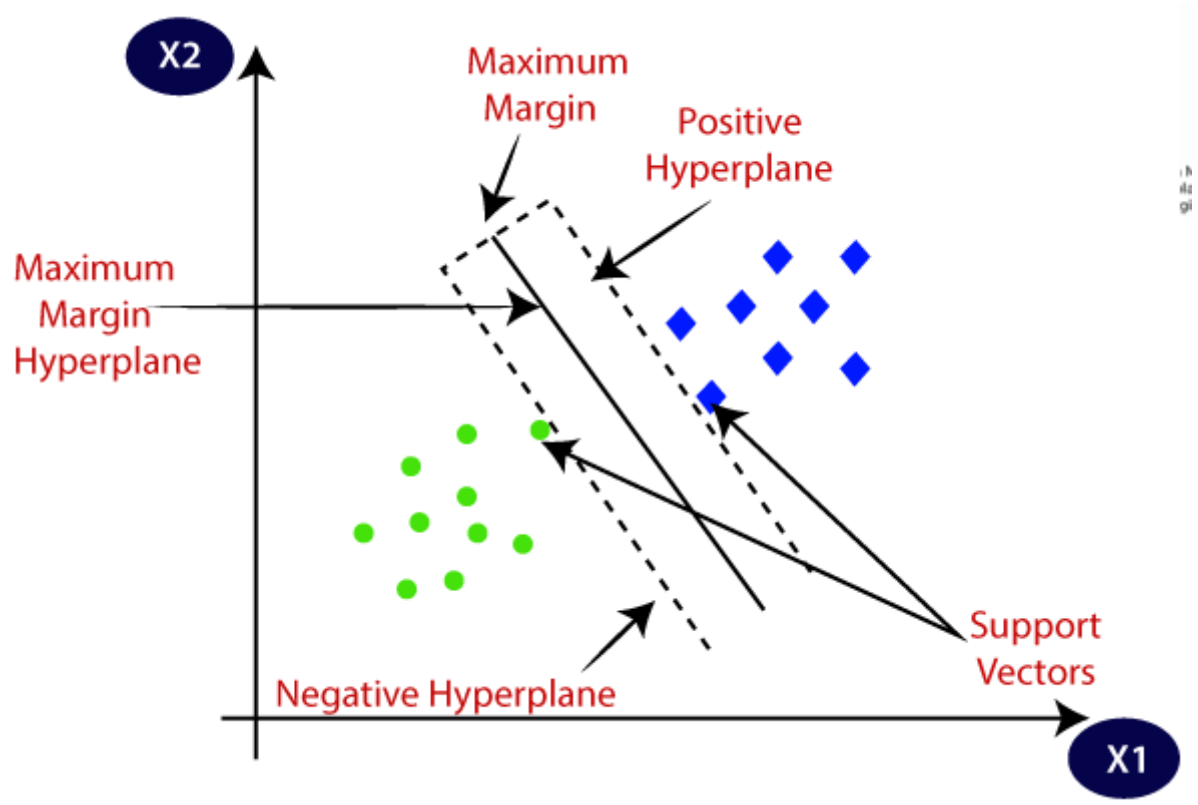


Figure 4: Visual representation of a hyperplane optimally dividing data

In situations where the training data set is much larger than the quantity of features, KNN algorithms outperform SVM algorithms. SVM requires a lot of time for training, therefore in situations where there are large amounts of features compared to the amount of training data available to the system, SVM outperforms KNN.

1.3.3 Random decision tree (Random Forrest) methodology

The Random Forest algorithm is a group learning technique for both classification and regression and is based on a 'divide and conquer' principle (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 7, 11-13). The approach combines multiple decision trees and combines each tree's prediction into one overall, averaged prediction. This method shows excellent performance in systems where the number of variables is significantly larger than the number of observations on those variables. Such systems are prone to overfitting (training a model on specific data too well, so that it becomes excellent at predicting the specific data it's testing, but comes with a significant drop in accuracy when making predictions based on brand new data), but applying random tree methodology with its varied way of making predictions can dramatically limit any possible overfitting.

A decision tree generally consists of a root node, internal node, branches, and leaf nodes. The root node is the topmost level node and contains the entire dataset being analysed, the internal node contains a test on a feature, each branch connects nodes to each other and contains the result of a test carried out, and each leaf node contains a class label.

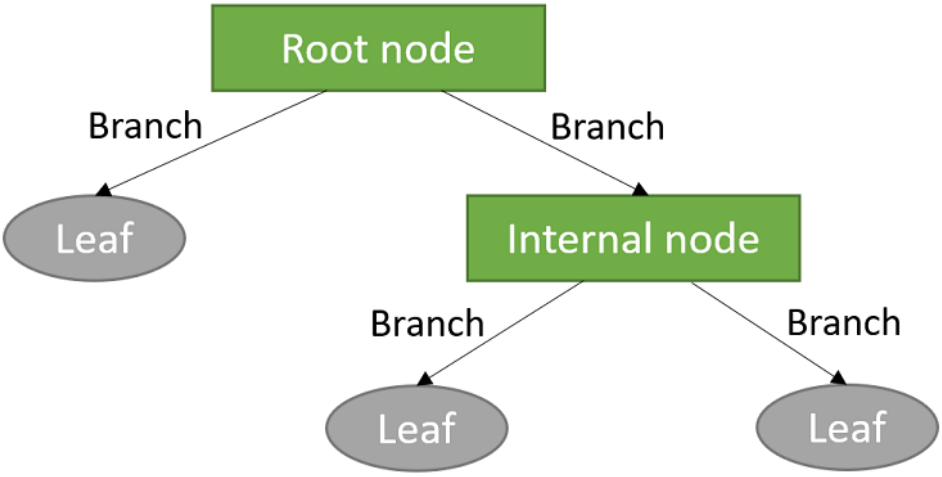


Figure 5: Visual representation of a simple decision tree structure.

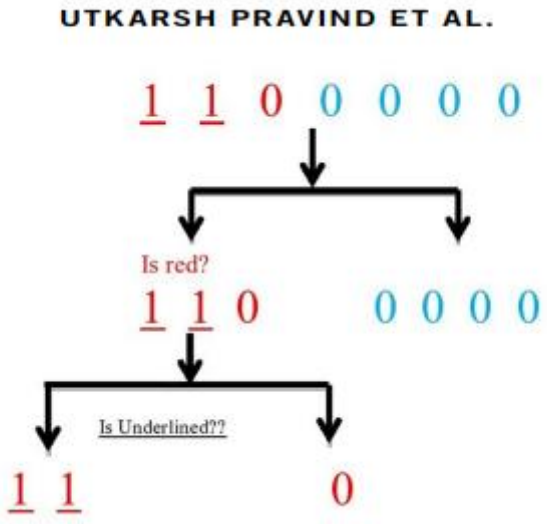


Figure 6: A simple decision tree in action. (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, p. 12).

As seen in the figure above, the root node contains the dataset: numbers of different colours with some being underlined, branches connect the root node to the internal nodes containing the first test parameters which check if the colour of the numbers is red, branches carry those test results to the next layer of internal nodes which is checking whether the numbers are underlined or not. The end result is three separate classes or 'leaf nodes', one for underlined red numbers, one for not underlined red numbers, and one for not underlined blue numbers.

Random Forest methodology is based on combining the predictions of several decision trees, with each tree being trained separately.

1.3.4 Cross-Domain Recommendation Systems (CDRS)

A CDRS can improve recommendations by extending their recommendation requests from one domain to multiple domains. By making use of correlations between two or more domains, a CDRS

can improve their recommendation outputs. For example, if there's a large amount of data on a person's movie preferences, but not much data on the same person's book or music preferences, the data from their movie preferences can be used to try and predict their book or music preferences (Ricci, Rokach, Shapira, 2022, pp. 485-486).

Brand new domains and domains with too little data to make accurate recommendations can make use of data from similar or related domains with vast amounts of data.

1.4 Current Applications of Recommender Systems

The modern giants of the tech industry have pioneered the advancement of recommendation systems, companies such as Google, Amazon and Netflix have all built their empires on the back of machine learning based recommendation systems.

1.4.1 Amazon's recommender system

Since 2012, approximately 35% of Amazon's total revenue is generated by its recommended purchases (MacKensie, Meyer & Noble, 2013). Amazon's approach to their recommendation system was dubbed 'item-to-item collaborative filtering', an algorithm that placed more emphasis on product data, which was in contrast to user based collaborative filtering, the standard of the day.

Through item-to-item collaborative filtering, the recommendation system uses a user's purchase, view and search history and, for each item, create a list of related items. Items that show up multiple times on the lists are then weighted, sorted and recommended based on how related they are to the user's histories.

The weight of an item's relation is determined by how often a user who buys one item buys another. Item A's relation to item B increases based on the percentage of people who bought item B who also bought item A, the higher the percentage, the higher the relation.

Since embracing item based collaborative filtering, Amazon have tested and implemented a variety of improvements to make customer recommendations more accurate, such as factoring in personal preferences for things like brands and fashion styles or different kind of recommendations based on time of day (Hardesty, 2019).

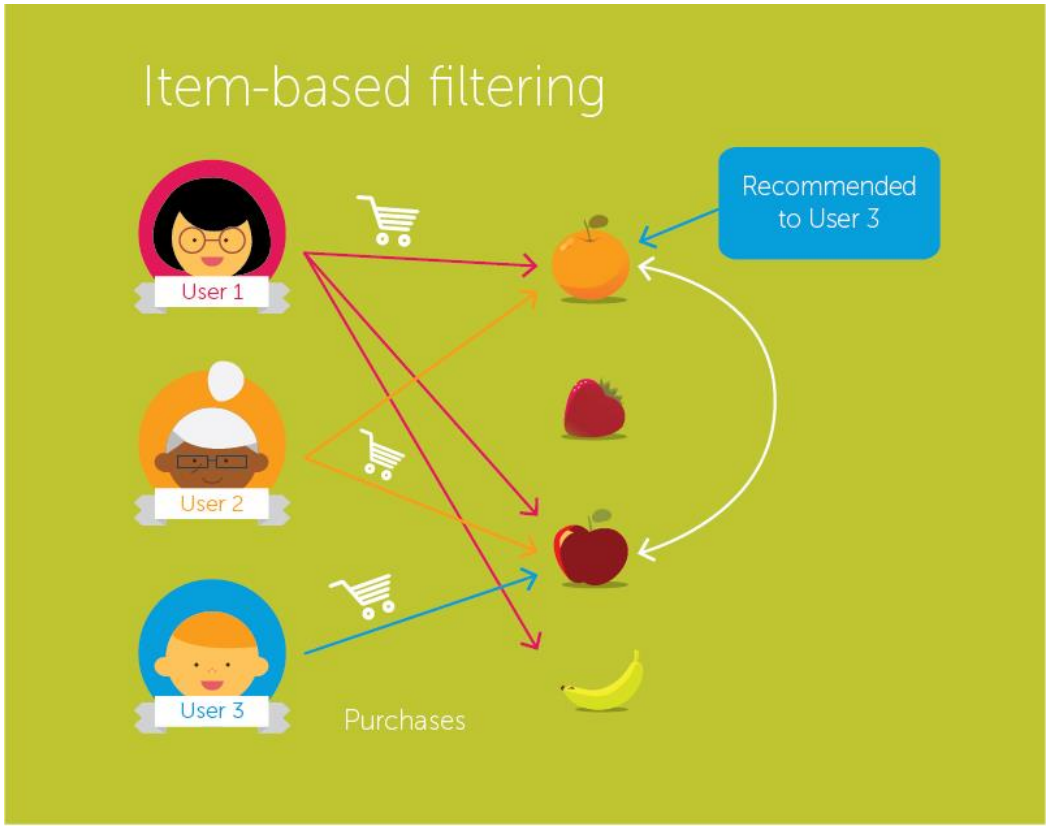


Figure 7: Item-to-item based collaborative filtering

1.4.2 YouTube's recommender system

YouTube is the world's largest content sharing platform, with over 500 hours of video being uploaded every minute, accurately recommending videos users would be interested in poses three main challenges:

1. Scale: The sheer scale of YouTube's massive user base and content library requires highly specialised learning algorithms to efficiently sort and determine the most appropriate recommendations.
2. Freshness: Recommendations must maintain a balance between freshly uploaded content and well-established videos.
3. Noise: Some important data is difficult to determine due to unobservable external factors related to the type of content YouTube serves. There is no reliable way to determine a user's level of satisfaction taken from a video they've watched. It's also challenging to define and sort many videos from their associated metadata, which can only describe a video in limited ways.

The YouTube recommendation system is comprised of two neural networks, one for candidate generation and one for ranking the candidates. The candidate generation neural network takes in a user's activity history as the input layer, and outputs a few hundred videos that are determined relevant from a large pool of videos. The ranking neural network then assigns a score to each video using an expansive set of features describing the video and the user (Covington, Adams & Sargin, 2016, pp. 1-2).

This two-stage format of recommendation system allows YouTube to make recommendations from a massive pool of videos while maintaining a high degree of certainty that the few videos appearing to the user are engaging and personalised.

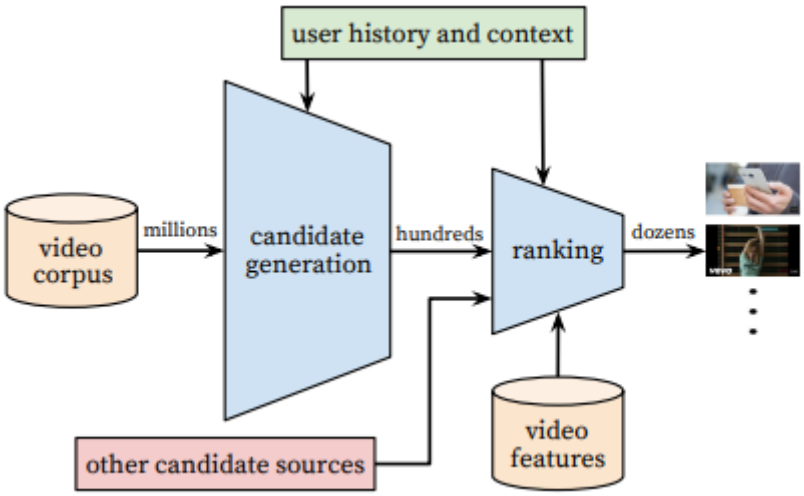


Figure 8: YouTube's recommendation system architecture, highlighting how possible video candidates for recommendation are narrowed down from millions to dozens before being recommended to the user (Covington, Adams & Sargin, 2016, p. 2).

1.5 User Experience and Recommender Systems

Recommender systems exist to help users make better choices, often from large content catalogues containing distinct items such as movies, books, food, laptops, jokes or even insurance policies (Knijnenburg, Willemsen, Ganter, Soncu & Newell, 2012, p. 442). Accuracy is generally the primary focus when it comes to building a good recommender system, the quality of a user's options to choose from is largely based on the accuracy of the systems recommendation. The consensus is that better systems lead to better recommendations, which in turn lead to a better user experience. However, researchers have argued that there are other factors that should be taken into consideration in regard to improving user experience (Ricci, Rokach & Shapira, 2022, pp. 21-23).

One aspect other than accuracy that can influence user satisfaction with the system is diversification of predictions. For example, a user who enjoys mostly action movies may be recommended the ten movies the system has determined as the top ten recommendations for this user, all of which happen to be action movies. This narrow selection of exclusively action movies may satisfy the user most times, but be lacking on the occasion that the user isn't in the mood to watch an action movie.

There are also situational or personal aspects that can apply to a user and change how a recommender system is perceived. For example, a user may have expertise in the area that the system is working in, which can significantly change how they view the area and recommendations in it. Another potential aspect is that some users may have privacy concerns that can clash with the data collection necessary to create predictions (Knijnenburg, Willemsen, Ganter, Soncu & Newell, 2012, pp. 452-453).

1.6 Limitations and Challenges of Recommender Systems

1.6.1 Scalability

With an ever-increasing number of users and items being kept track of in recommender systems, scalability becomes a serious problem. A huge number of users along with a huge number of items makes the collaborative matrix more and more complex, which leads to a loss in performance quality of the recommender system (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 17, 187).

1.6.2 Data sparsity

Many modern commercial recommender systems use a large quantity of datasets, which can hugely complicate the user item matrix used in collaborative filtering and consequently reducing the performance of the recommender system. This problem often leads to the cold start problem (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 17, 180, 187).

1.6.3 Cold starts

Cold starts are one of the most familiar problems experienced by recommender systems. A cold start occurs when there is a lack of information about the users or items in the system, consequently limiting the systems prediction accuracy. One solution to cases with cold start problems is to use a content-based system, which can provide accurate predictions for new items without the need of any user information (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, p. 17). Another possible solution is to make use of information from a similar domain with a cross domain recommender system (CDRS) to help extrapolate useful data on items or users based on the other domains similarities to the intended for use domain (Ricci, Rokach & Shapira, 2022, p. 28).

1.6.4 Long tails

A long-tail item or service is one that has historically been unpopular or rarely noticed by people. Long-tail items often pass under the radar of recommendation systems specifically because there's less data available on them compared to other items and services, which can result in these items and services being forgotten or unnoticed by people. When given equal attention however, long-tail items and services can achieve great success with both customers and businesses (Falk, 2021, pp. 5-6).

Cross-domain recommendation systems can offer potential solutions to the long-tail problem with their ability to make use of data from related items and services.

1.6.5 Privacy in recommendation systems

As recommendation systems become more and more widespread and integrated into daily life, it can cause people to be concerned about their privacy. People can be apprehensive about providing authentic personal information, which can decrease the accuracy of the recommendation system (Ricci, Rokach & Shapira, 2022, p. 40). One possible way to secure people's personal information in the system is to encrypt people's personal data. The problem with this method, however, is that it comes with a high computational cost. Recommendation systems in domains where privacy is paramount, such as healthcare or banking depend on more cross-platform systems to be developed with a primary goal of privacy preservation, while keeping the system as efficient and accurate as possible.

1.7 The Future of Recommender Systems

1.7.1 Concept drift

Recommendation systems are great at sifting through large amounts of collective data, however, their performances are not as good when dealing with the complex characteristics of 'Big Data' over longer periods of time. Modern recommendation systems usually work under the assumption that people's preferences remain relatively static over time, resulting in historical records being weighed equally. In reality though, people's preferences change as the people themselves do. As a person's historical records increase, older records may be inconsistent with the persons current preferences. This problem with accurately weighing a person's data from historical records is known as 'concept drift'. Recommendation systems with methods of time awareness were developed to address this issue, though these methods have failed to accurately detect the concept drift. Systems that can manage and describe changes with the complex dynamics of Big Data are required (Pavan Kumar, Vairachilai, Potluri & Nandan Mohanty, 2021, pp. 200-201).

1.8 Conclusion

AI, Machine Learning and the area of recommender systems are still in their infancy, they show significant potential to perform a wide variety of tasks that could be hugely beneficial for people and the world, but also has the potential to be hugely harmful. As the development of AI, and consequently recommender systems, continues to accelerate, new recommendation methodologies are developed and current methodologies improved upon to provide better recommendations across a plethora of areas, from movie recommendations to healthcare to marketing, and everything in between. For businesses in modern times, especially any kind of online businesses, it's become almost a necessity to use AI tools to improve their capabilities, or risk being outcompeted.

2 Chapter Two: Requirements

2.1 Introduction

In this chapter, the requirements for the intended application are researched and defined. The intended application is a movie recommendation system with a primary purpose of giving users AI generated, personalised movie recommendations based on a variety of factors. Users should also be able to search and browse through movies, provide ratings and reviews, and be able to interact with their fellow users.

To help in the development of this application, similar applications were researched to determine their advantages, disadvantages and how they could relate to the intended application. Interviews and surveys were conducted to gauge potential users interest in a movie recommendation system and features and qualities that they valued and would want to see from such an application. From this research, a list of requirements for the application can be defined.

Once a list of application requirements has been defined, the requirements to develop that application can be determined and explained.

2.2 Requirements gathering

2.2.1 Similar applications

Three different applications with different similarities were researched for the development of this application: MovieLens.org, IMDb.com and MyAnimeList.net. MovieLens.org was chosen to be researched because of their committed recommendation focused approach and variety of recommendation options offered to users. IMDb.com was researched because of their management of extensive information on movies and people. MyAnimeList.net was researched because of their community driven approach to recommendations and content management.

2.2.1.1 MovieLens.org

MovieLens is a research website run by GroupLens Research at the University of Minnesota, it is a web-based recommender system that uses collaborative filtering to recommend movies that users might enjoy and avoid ones they won't. Personalised recommendations of movies the user hasn't seen yet are based on user ratings.

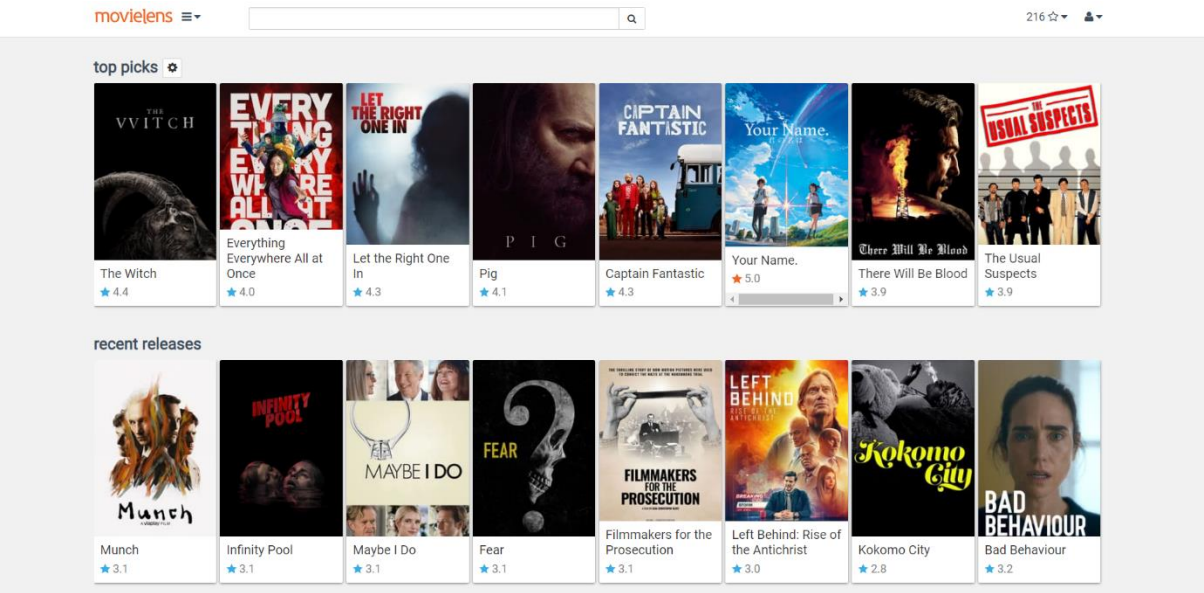


Figure 9: MovieLens Homepage.

The MovieLens Homepage, as shown above in figure 9, has a simplistic design. It has a navigation menu and different categorised rows of movie cards showing minimal movie information, that's it.

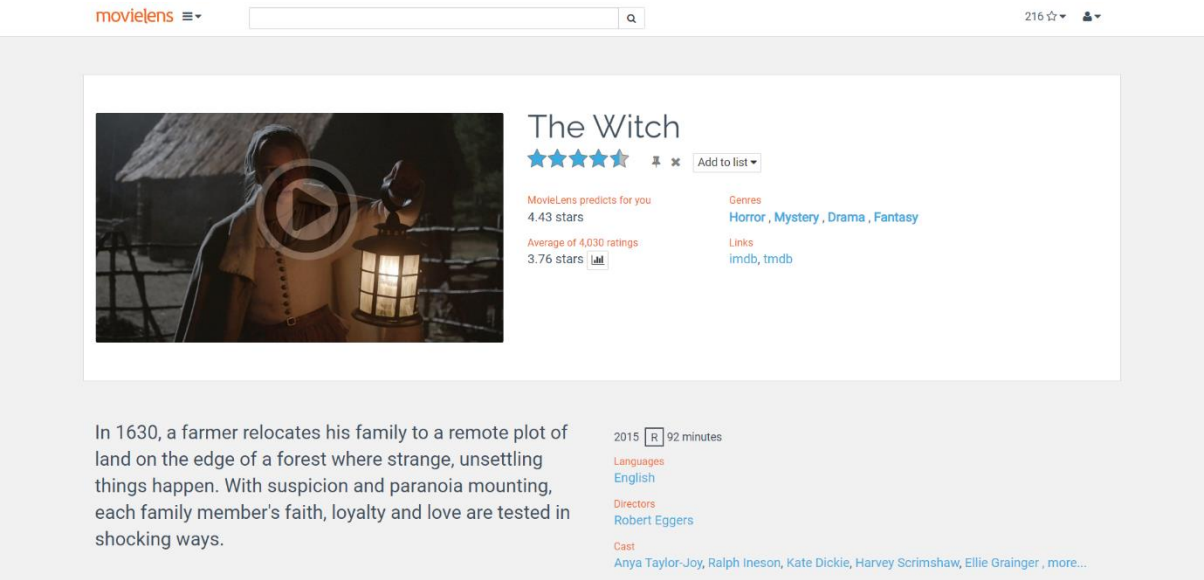


Figure 10: MovieLens individual item page #1.

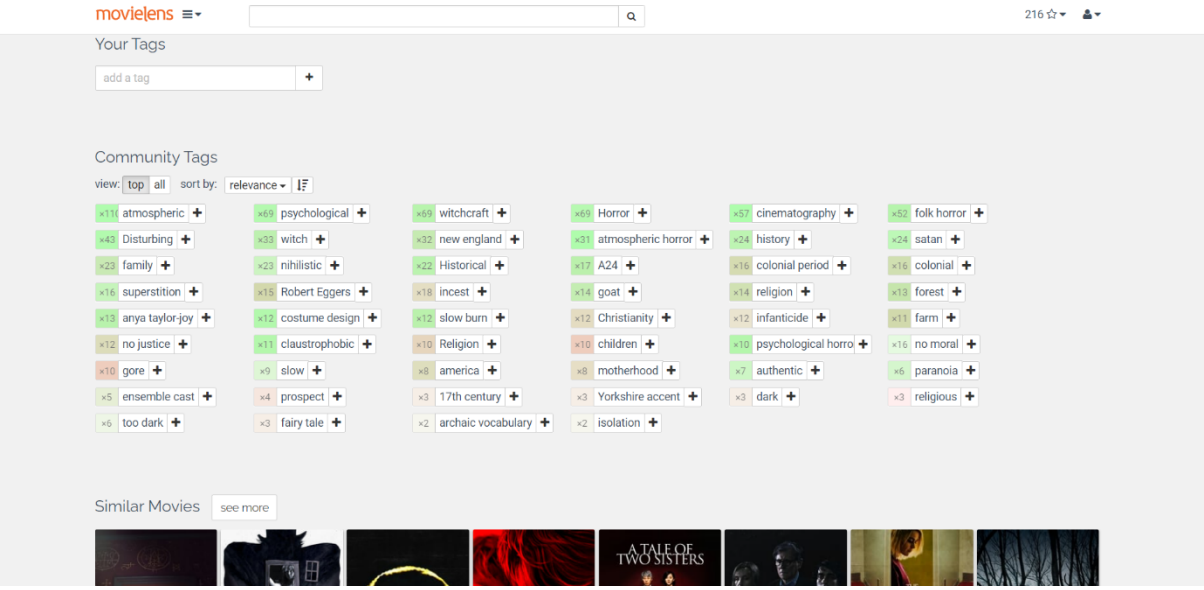


Figure 11: MovieLens individual item page #2.

Each movie directs to its own individual movie page, as depicted above in figures 10 and 11. This page contains more information on the movie, contains a list of community assigned tags, and shows a row of similar movies.

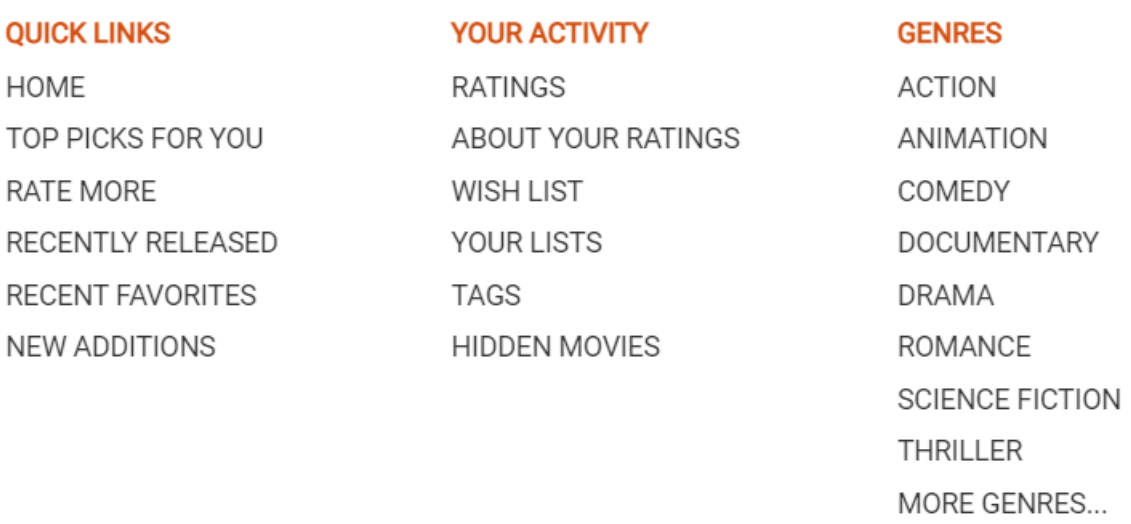


Figure 12: MovieLens navbar dropdown.

MovieLens' navigation menu drops down and provides various links throughout their website. As can be seen in figure 12, it follows the pattern of the website and is very simplistic.

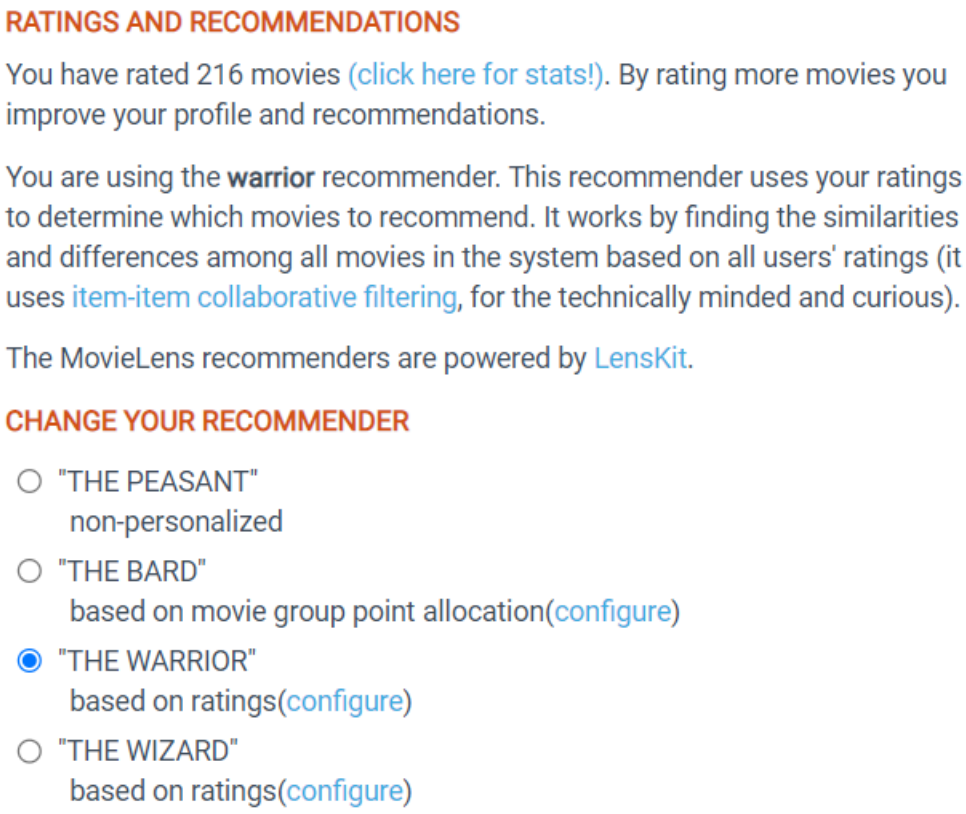


Figure 13: MovieLens recommendation options.

As shown in figure 13, MovieLens offers users the option to change how their recommendations are generated.

2.2.1.1.1 Advantages

- Simplistic user interface.
- Customisable recommendation methods.
- Is the largest database of movie ratings in the world, offering the greatest potential for the most accurate recommendations presently possible.
- Can hide individual films from recommendations.

2.2.1.1.2 Disadvantages

- Cannot hide groups (genres, movies by director, movies with actor etc.).
- Community tags, as can be seen in fig. 11, stand out from the rest of the aesthetics.
- Can be quite static when a user reaches a point of having rated all the movies they've seen that the system has given them, and are not very interested in what's left of what the system has given them. Without an option of generating an entirely (or at least mostly) new set of recommendations, the user would have to individually hide every movie or else see the exact same recommendations they aren't very interested in every time they enter the website.

2.2.1.2 IMDb.com

IMDb (The Internet Movie Database) is the world's largest and most popular database of movies, television and celebrity information and is designed to help users browse movies and TV series and decide what to watch. IMDb also offers users local movie showtimes, ticketing information, trailers, critic and user reviews, personalised recommendations (based on user ratings, watchlist activity and users with similar tastes), photo galleries, entertainment news, quotes, trivia and box-office data.

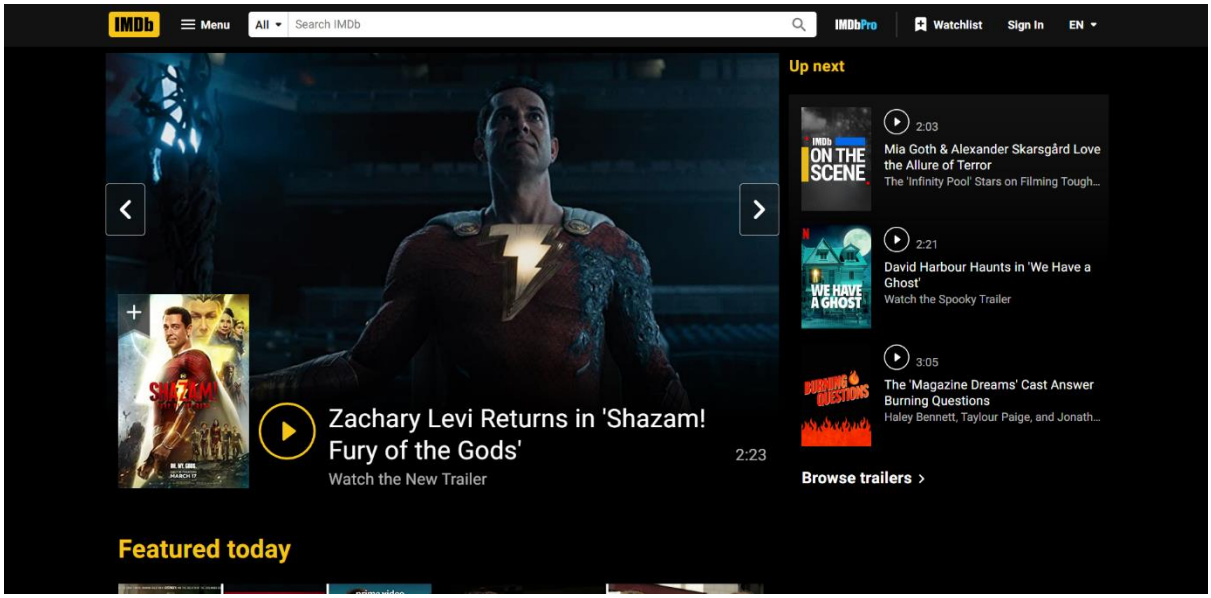


Figure 14: IMDb homepage #1.

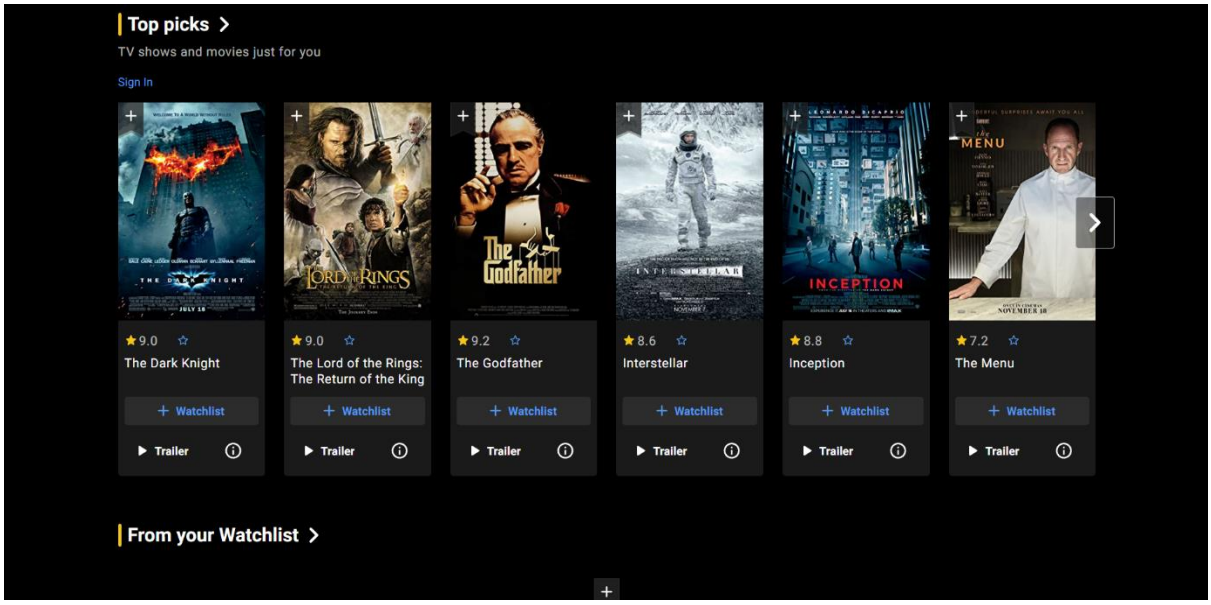


Figure 15: IMDb homepage #2.

As can be seen above in figures 14 and 15, IMDb's homepage showcases the newest popular releases, and the highest rated movies from their database. The homepage scrolls down for a long time, containing a lot of various categorised movie lists and other information.

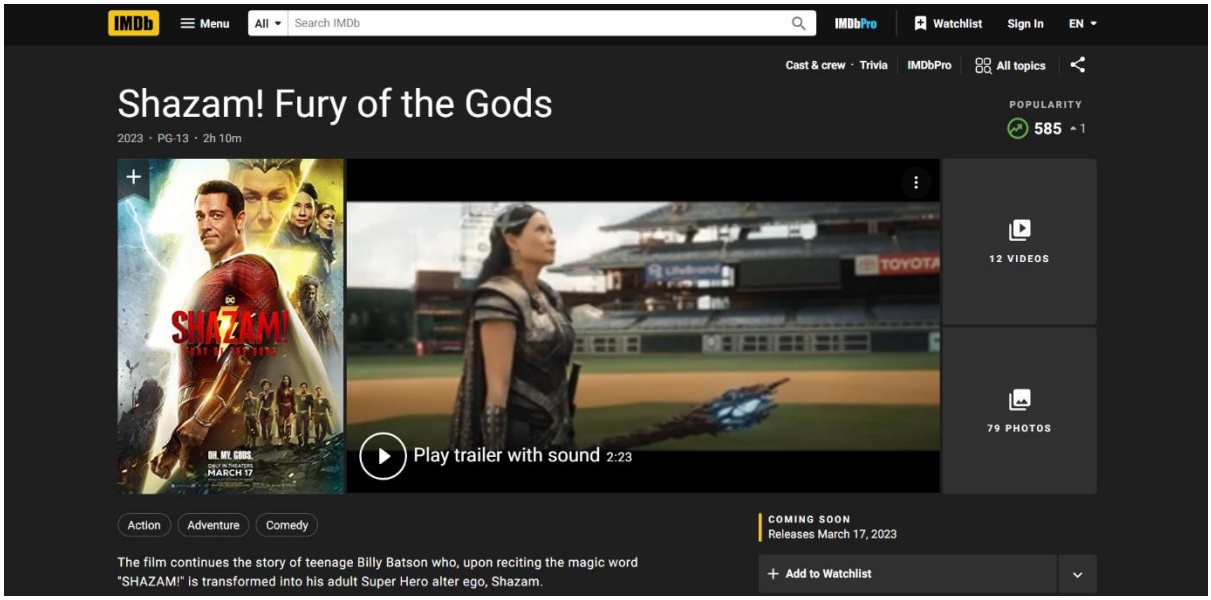


Figure 16: IMDb individual item page #1.

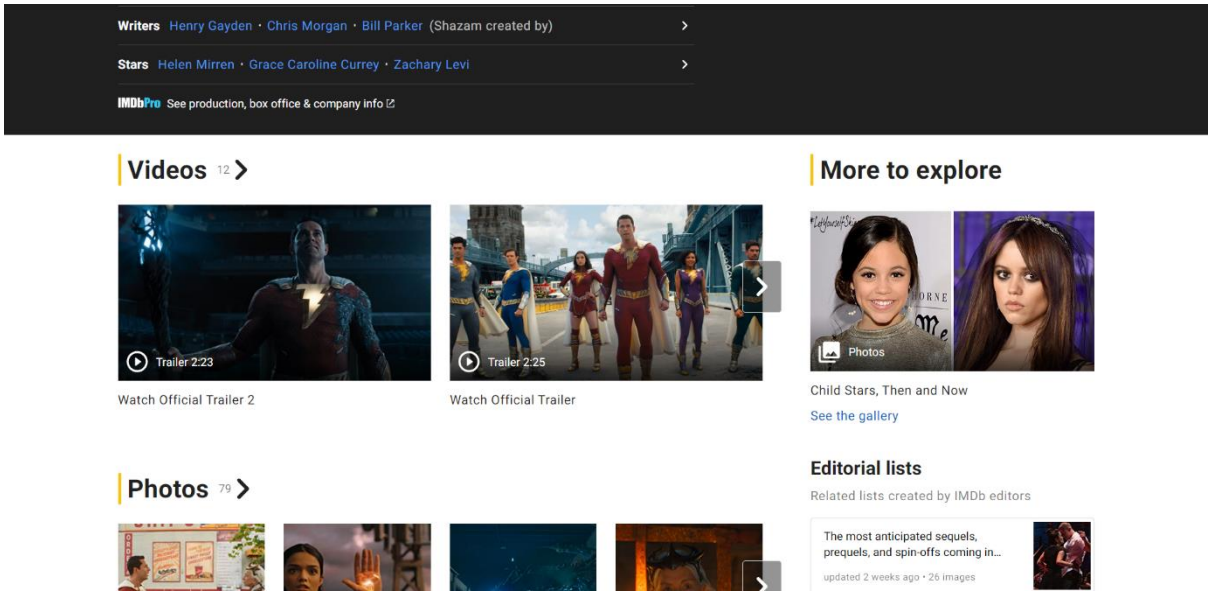


Figure 17: IMDb individual item page #2.

The IMDb individual item pages contain extensive information for that item, from trailers, to cast, to similar movies, to synopsis, to trivia, to user reviews, to frequently asked questions, to awards and box office results and everything in between.

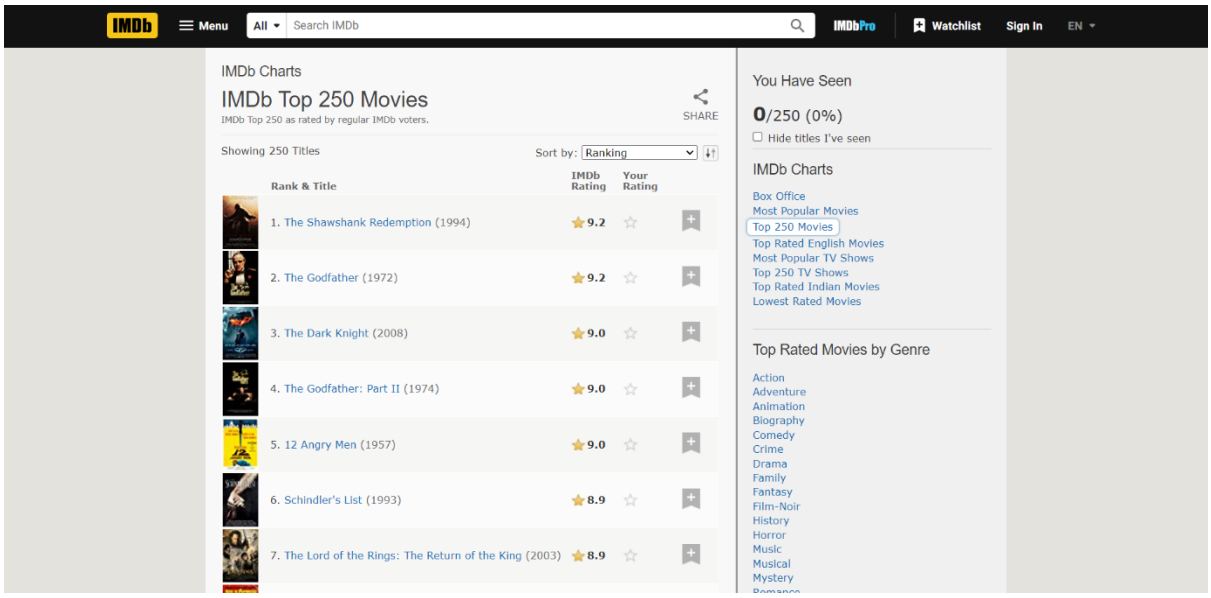


Figure 18: IMDb list page.

As can be seen in figure 18 above, IMDB's list pages aren't as cluttered as their other pages, it shows minimal movie information and has a side bar of useful quick links to other list pages.

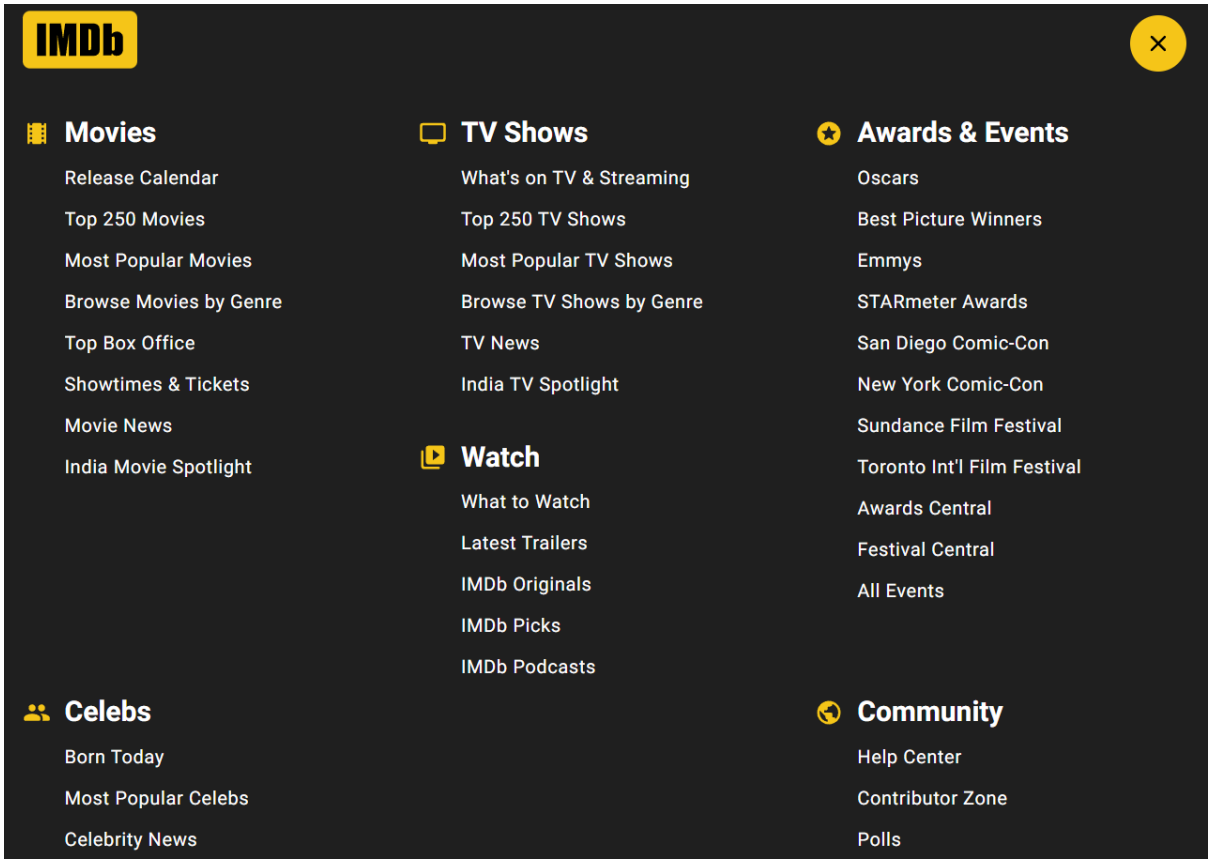


Figure 19: IMDb navbar dropdown.

IMDB's navigation menu drops down and encompasses the entire browser window, as can be seen above in figure 19, there are a lot of options.

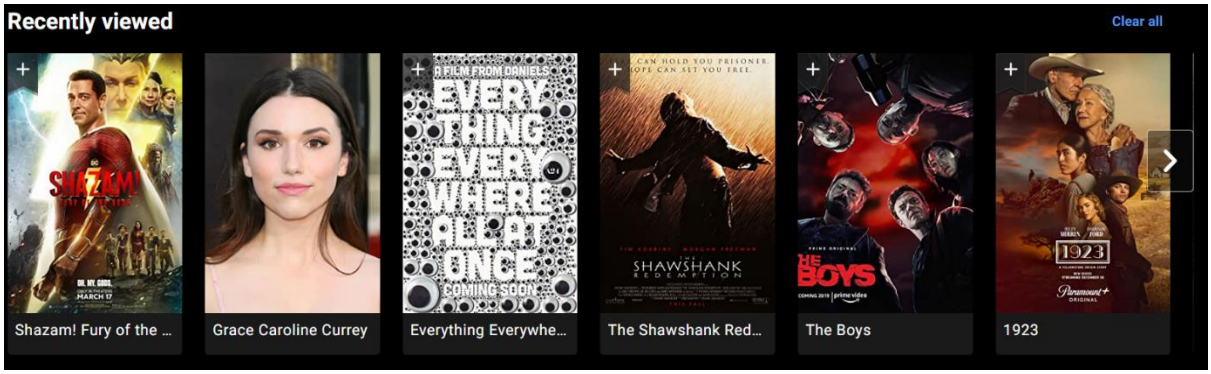


Figure 20: IMDb shows the users most recently viewed pages at the bottom of almost every page on the website, as a footer of sorts.

2.2.1.2.1 Advantages

- Largest database of movies, TV Series and video games in the world.
- Extensive information on movies, TV Series, video games and everyone and everything that went in to making them.

- Rating information on over 10 million titles (including individual episodes) by over 83 million registered users.

2.2.1.2.2 Disadvantages

- Information overload, users are given so much information on movies, series, people etc. that it may take them longer than they expect to find a piece of information they were looking for, or users unfamiliar with the website might not find what they were looking for at all.
- Most users tend to only leave reviews if they either loved or hated the content, which skews the results in one direction or the other.
- Navigation drop down menu takes over entire screen when expanded

2.2.1.3 MyAnimeList.net

MyAnimeList (MAL) is an online, community driven and community focused database of manga and anime content, it is the world's largest anime and manga database and community and is run by volunteers. MAL contains information on over 5 million Japanese, Korean and Chinese animation and comic book entries and receives over 120 million visitors a month. Users can create lists that they seek to complete, rate, submit reviews, write recommendations, blogs, use the site's forum, create clubs/groups with users similar who have similar interests.

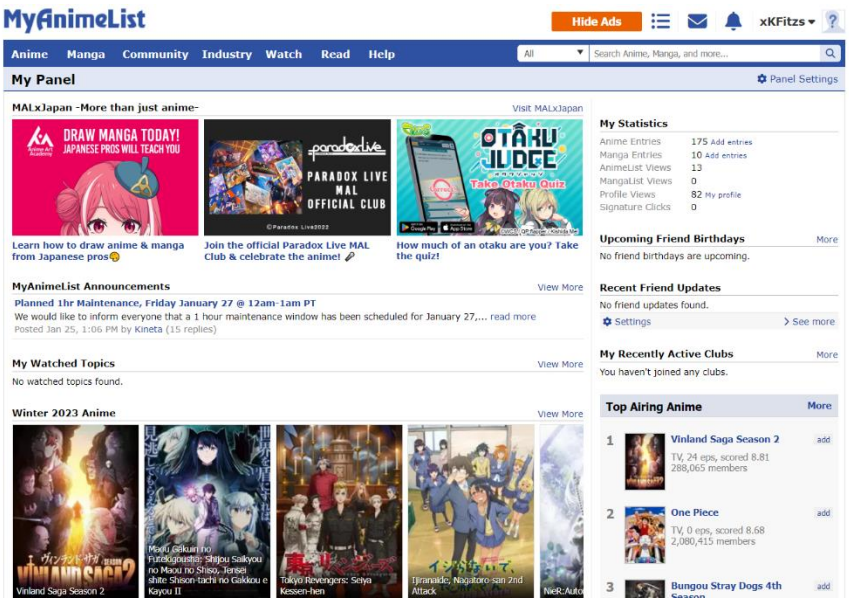


Figure 21: MyAnimeList homepage #1.

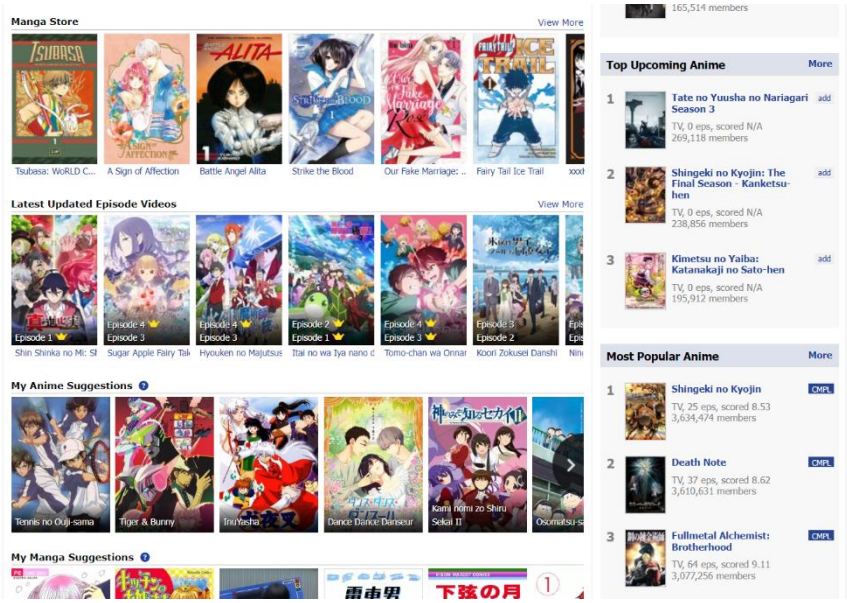


Figure 22: MyAnimeList homepage #2.

As can be seen above in figures 21 and 22, the MyAnimeList homepage has a lot going on. It seems that they've tried to pack as much information as possible into as small a space as possible. They showcase the newest popular content, advertise their store, show profile options, show upcoming content, promote user submitted content, highlight features articles and much, much more.

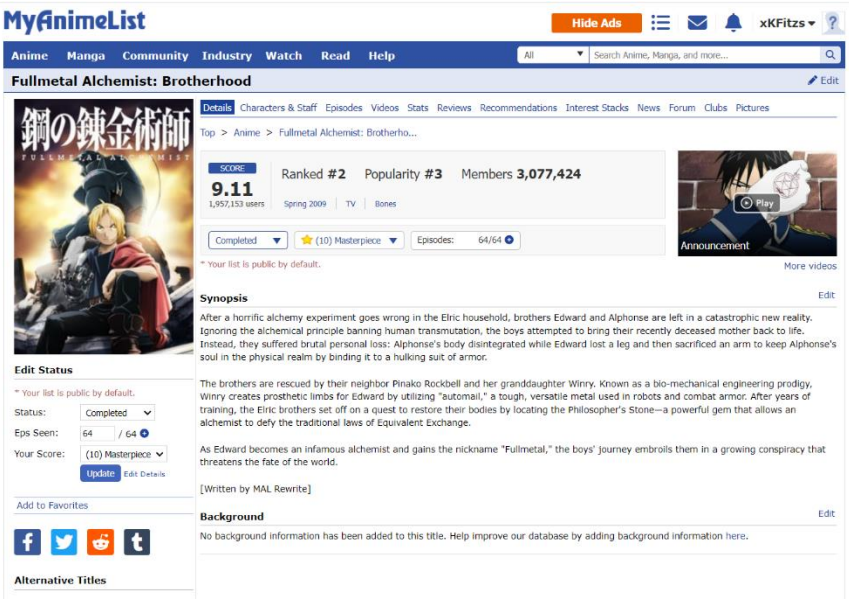


Figure 23: MyAnimeList individual item page #1.

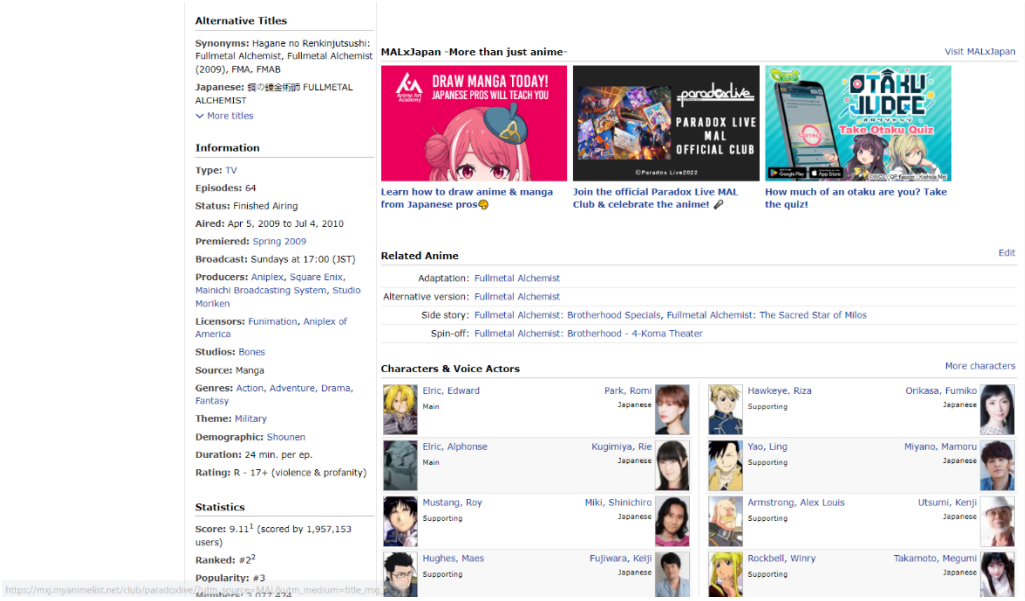


Figure 24: MyAnimeList individual item page #2.

The individual item pages show more information on its item, from content details to people involved. It shows the most recent news on the content, more related content, music if it exists, user reviews and user submitted recommendations.

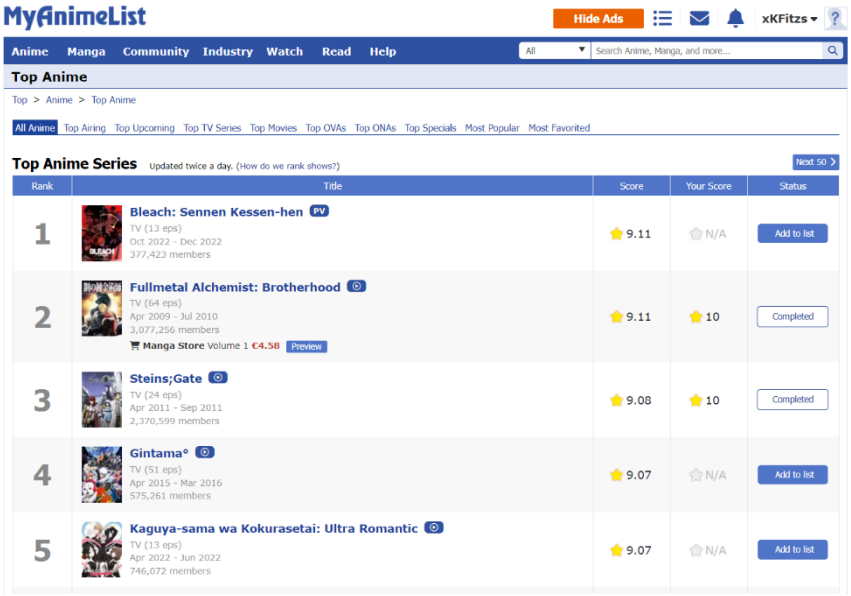


Figure 25: MyAnimeList list page.

As seen above in figure 25, content can be displayed in predefined or manually filtered lists. Each item can be rated by the user and/or added to predefined or user created lists, such as ‘favourites’, ‘plan to watch’ or ‘currently watching’.

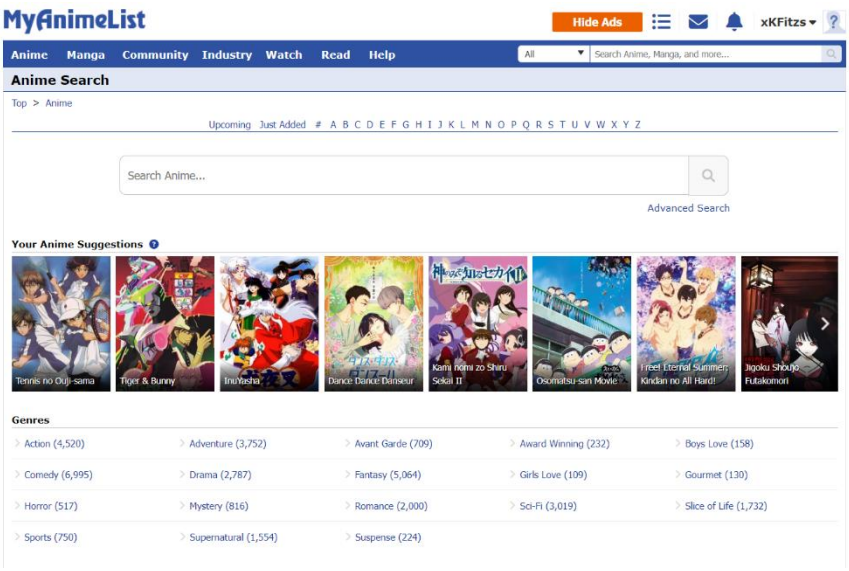


Figure 26: MyAnimeList search page.

As can be seen above, the search pages have search bars at the top and quick links to filtered search pages by the likes of genres, themes, rankings, year of release and more. When a search term is entered, a list page is returned.

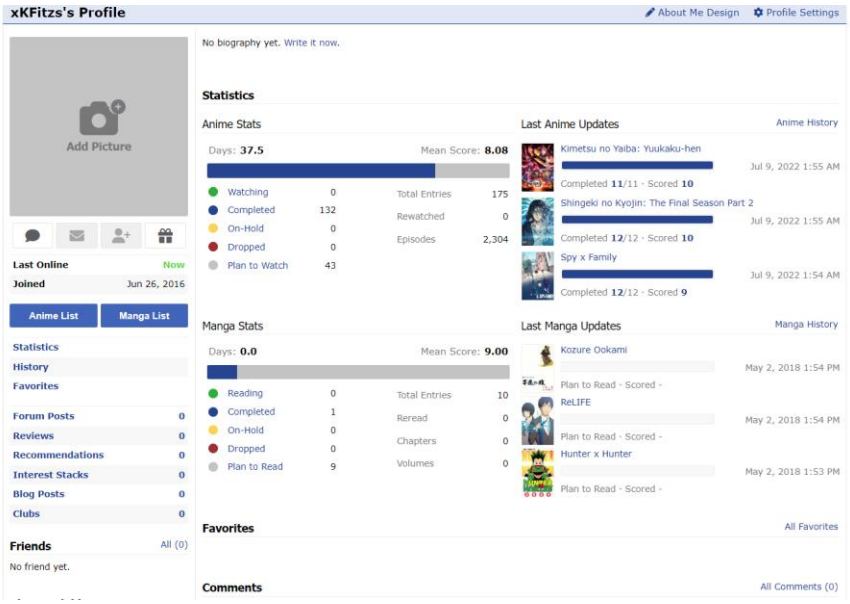


Figure 27: MyAnimeList user profile page.

The profile page shows account details and statistics, users can comment on each other's profile.

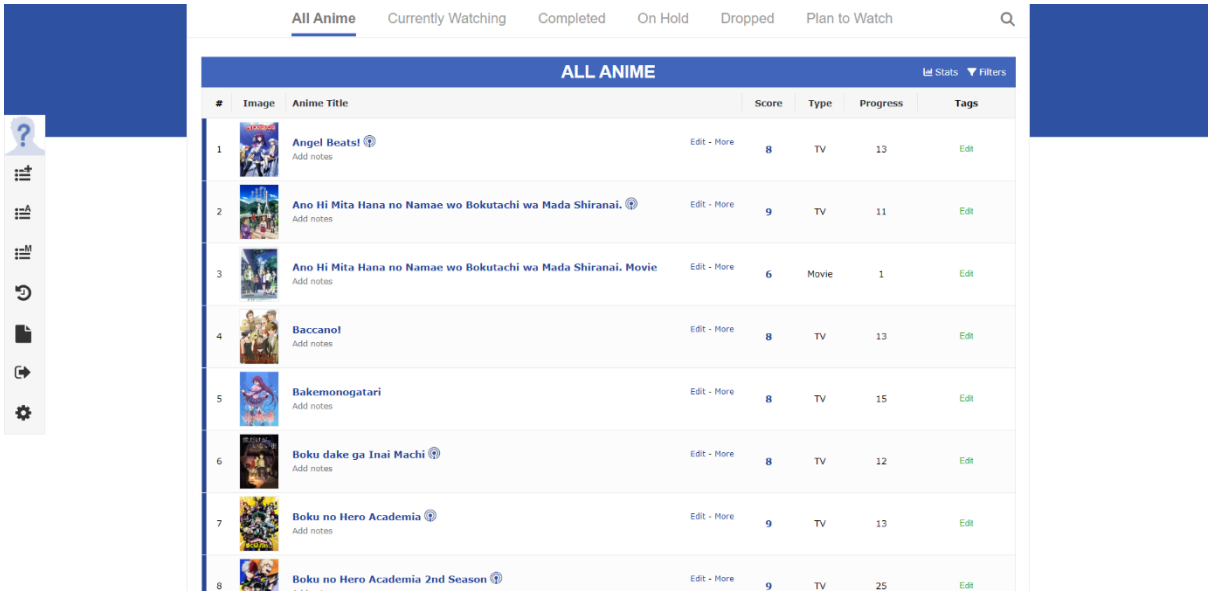


Figure 28: MyAnimeList user's list page.

Users can manage their lists, add notes and tags, movie items between their own lists and view other peoples lists.

2.2.1.3.1 Advantages

- A strong sense of community for users who want that, offering and promoting forums, clubs, blogs, user-to-user interaction and user created content.
- Users can create and manage lists to keep track of what they've seen or read.
- Allows users to use premade or write their own CSS to customise their lists.

2.2.1.3.2 Disadvantages

- Being run by volunteers, the moderation team can have limitations when working with such a big website, such as slow response times with user reports or user submissions
- No flexibility doesn't scale at all.
- Some pages (e.g., Homepage) can feel very cluttered and be jarring even.

2.2.2 Interviews

Interview questions were constructed with the goal of understanding how people interact with movies and recommendation systems, and what they prioritise when seeking a movie to watch. The interviews were conducted with three people over online voice chat and recorded for documentation.

2.2.2.1 Interview questions and answers

Question 1: What movie information might you ever like to see when looking for one to watch?

Answer 1: “Genres, Theme, Year movie takes place in, actors”

Answer 2: “Title, short description, IMDb score, Rotten Tomatoes score, actors, release date, genres, poster, reviews”

Answer 3: “Rating, reviews, genres, cast”

Question 2: Which of the following information can you see as being useful to you when looking for a movie to watch, and which would you consider essential?

Interviewees were given over 30 areas of information to consider, the information areas with the highest priority are:

- Global average rating (Essential to 2/3 interviewees)
- Synopsis
- Release date
- The movie’s all time ranked position based on quality (average rating) (Essential to 2/3 interviewees)
- How many users have rated it
- User reviews
- Cast
- Director (Essential to 2/3 interviewees)
- Link to external website (IMDB, RottenTomatoes)
- Subtitles

Question 3: In what ways would interacting with movies be helpful or interesting to you?

- Being able to give ratings (3/3 interviewees would find this helpful/interesting)
- Being able to hide/do not recommend individual movies (3/3 interviewees would find this helpful/interesting)
- Being able to hide/do not recommend entire genres (3/3 interviewees would find this helpful/interesting)
- Being able to add to premade or custom lists (e.g., ‘My Favourites’, ‘Want To Watch’ or ‘To Watch With Partner’) (3/3 interviewees would find this helpful/interesting)

Question 4: When searching or browsing movies, which of the following filters might you find useful?

Interviewees were given over 20 options to give an opinion on, the filter options most highly prioritised are:

- A minimum rating
- By specific director
- With specific cast member
- Results per page
- Excluding x genres
- Only including x genres

Question 5: What ways would you like to be able to sort lists of movies (all optionally ascending or descending)?

- By prediction score (3/3)
- By global average user ratings (3/3)
- By quantity of user ratings (2/3)
- Alphanumerically (2/3)
- By release date (2/3)
- Awards (1/3)

Question 6: Would you like to be able to interact with other users in any ways?

Only one of the three interviewees would like to be able to interact with other users, they would like to be able to interact with other users in the following ways:

- View other users' profiles
- View other users' ratings
- View other users' lists
- View other users' recommendations
- Add other users as friends
- Follow user to see when they rate/review something new

At the end of the interview, interviewees were asked if there were any questions they were expecting to be asked, and if so, their answers to those questions. One of the interviewees had an expected question, which was subsequently asked to the other two interviewees.

Question 7: How would you like to be able to give ratings?

Interviewee 1: "Be able to give overall rating and rating to individual aspects e.g. (5 stars to acting, 4 starts to music, 5 stars to writing etc.)"

Interviewee 2: "Score out of 10 or star system"

Interviewee 3: "Out of 10"

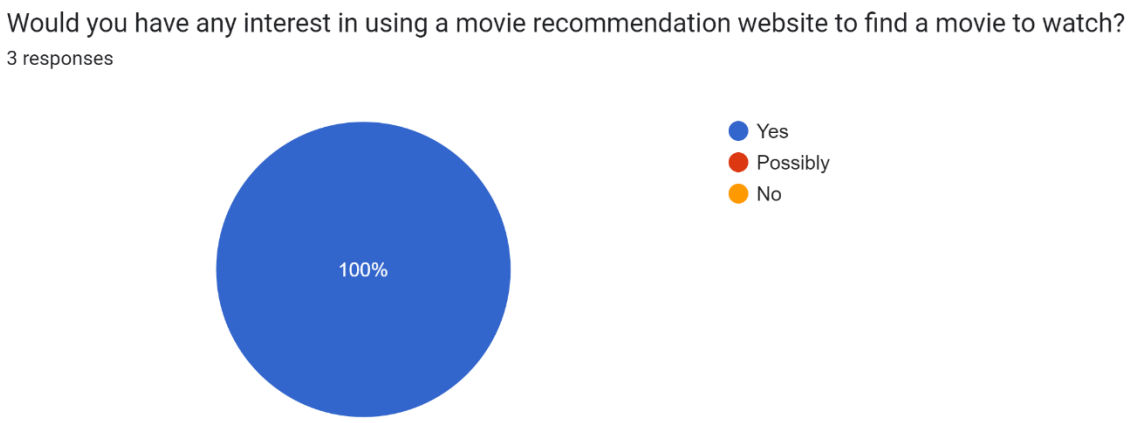
2.2.2.2 *Interview conclusion*

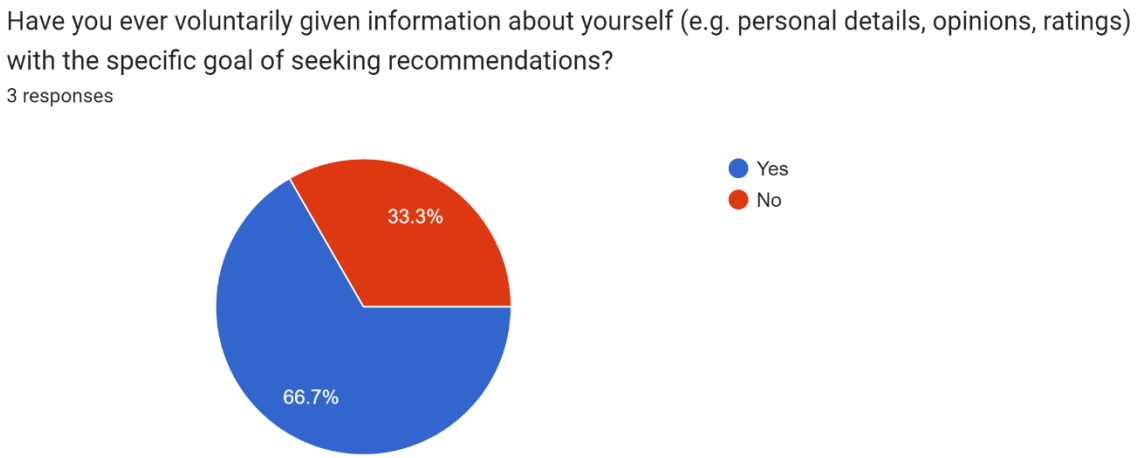
When searching for a movie to watch, some information is valuable to most people, some information is worthless to most people, but some information that is worthless to most people can be essential to some. Ratings and scores are consistently one of the most essential attributes in all areas of interacting with movies.

Two of the interviewees first language was not English, despite the small sample size, some patterns began to emerge from this difference. Those two interviewees placed a higher priority on areas to do with language and subtitles.

2.2.3 Survey

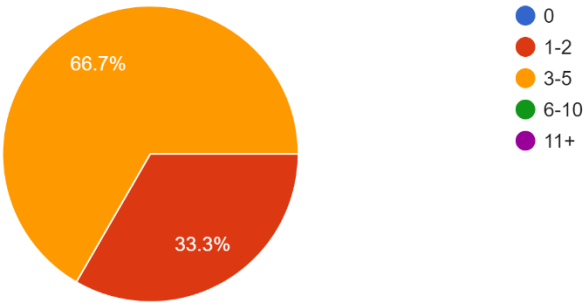
A survey was created with the goal of gauging people's interest in using a movie recommendation system.





On average, how many movies do you think you watch each month?

3 responses

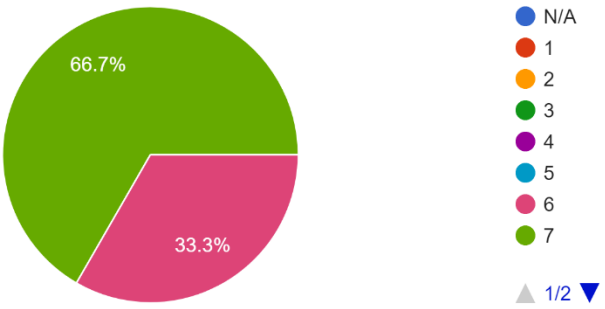


How important is the average user rating of a movie to you when looking for something to watch?

3 responses

Do you generally have a minimum average rating you would consider when looking for a movie to watch and if so, what is that minimum rating (out of 10)?

3 responses



2.2.3.1 Survey conclusion

It is difficult to draw meaningful conclusions from such a small sample size, but from this small sample size some things are indicated. People are interested in being recommended movies they would like and the average rating of a movie can be very important.

2.3 Requirements modelling

2.3.1 Functional requirements

Create a numbered list of what the application should be able to do. Start with the most important feature.

1. Allow users to sign up and create accounts
2. Make and present simple, generic recommendations based on initial account creation details
3. Show important movie information (e.g., Title, poster, average rating)
4. Allow users to submit a general rating for movies
5. Make personalised movie recommendations to users
6. Allow users to search and browse movies
7. Allow users to submit detailed ratings for movies (e.g., 5 stars to acting, 4 starts to music, 5 stars to writing)
8. Allow users to modify the method that their recommendations are determined
9. Allow users to view individual movies in more detail, showing more movie information (e.g., Cast & characters, synopsis, reviews)
10. Allow users to apply various filters to searches/recommendations (e.g., Rating range, genres)
11. Allow users to sort items by various attributes (e.g., Average rating, release date)
12. Allow users to hide individual movies
13. Allow users to add to premade lists such as 'Favourites' or 'Want to watch'
14. Allow users to track movies they've interacted with and how they interacted with them in their profile
15. Allow users to submit reviews of movies
16. Allow users to create and add to custom lists of movies (e.g., To watch with partner)
17. Allow users to switch between a dark and light mode
18. Allow users to hide groups of content (e.g., Movies by director, movies with actor)
19. Allow users to message each other
20. Allow users to add each other as friends
21. Allow users to view other users' profiles
22. Give users the option of having various parts or the entirety of their profile private and hidden
23. Allow users to follow other users to be notified when/if the followed user rates or reviews

2.3.2 Non-functional requirements

1. Recommendations update within a couple of seconds of adding a new rating
2. Have multiple recommendation models for the user to switch between
3. Movie information to be able to show:
    - Title
    - Poster
    - Age Certification
    - Runtime
    - Release Date
    - Global Average Rating
    - Quantity of Ratings
    - The systems prediction score for the user
    - Language
    - Synopsis
    - Plot Summary

- Trailer
- Subtitle options
- Global ranked position based on average rating
- Global ranked position based on quantity of ratings
- User Reviews
- Critic Reviews
- Cast
- Director
- Writer
- Producer
- Production Company
- Link to external websites (e.g., IMDb, RottenTomatoes)
- Related tags (e.g., for Shawshank Redemption 'serious' or 'prison' etc.)
- Genre specific ranked position based on average rating
- Genre specific ranked position based on quantity of ratings
- Screenshots
- Country of Origin
- Colour
- Aspect ratio
- Frame rate
- Official movie website
- If it's Dubbed

4. Filter methods:
    - Rating range
    - Quantity of ratings range
    - Prediction score range
    - Release date range
    - Age rating range
    - Language
    - By director
    - Containing actor/actress
    - By Writer
    - By Production Company
    - Results per page
    - Genres
    - Exclude genres
    - Subtitles
    - Dub

5. Sort methods
    - By prediction score
    - By global average rating
    - By quantity of ratings
    - Alphanumerically
    - By release date

2.3.3 Use case diagram

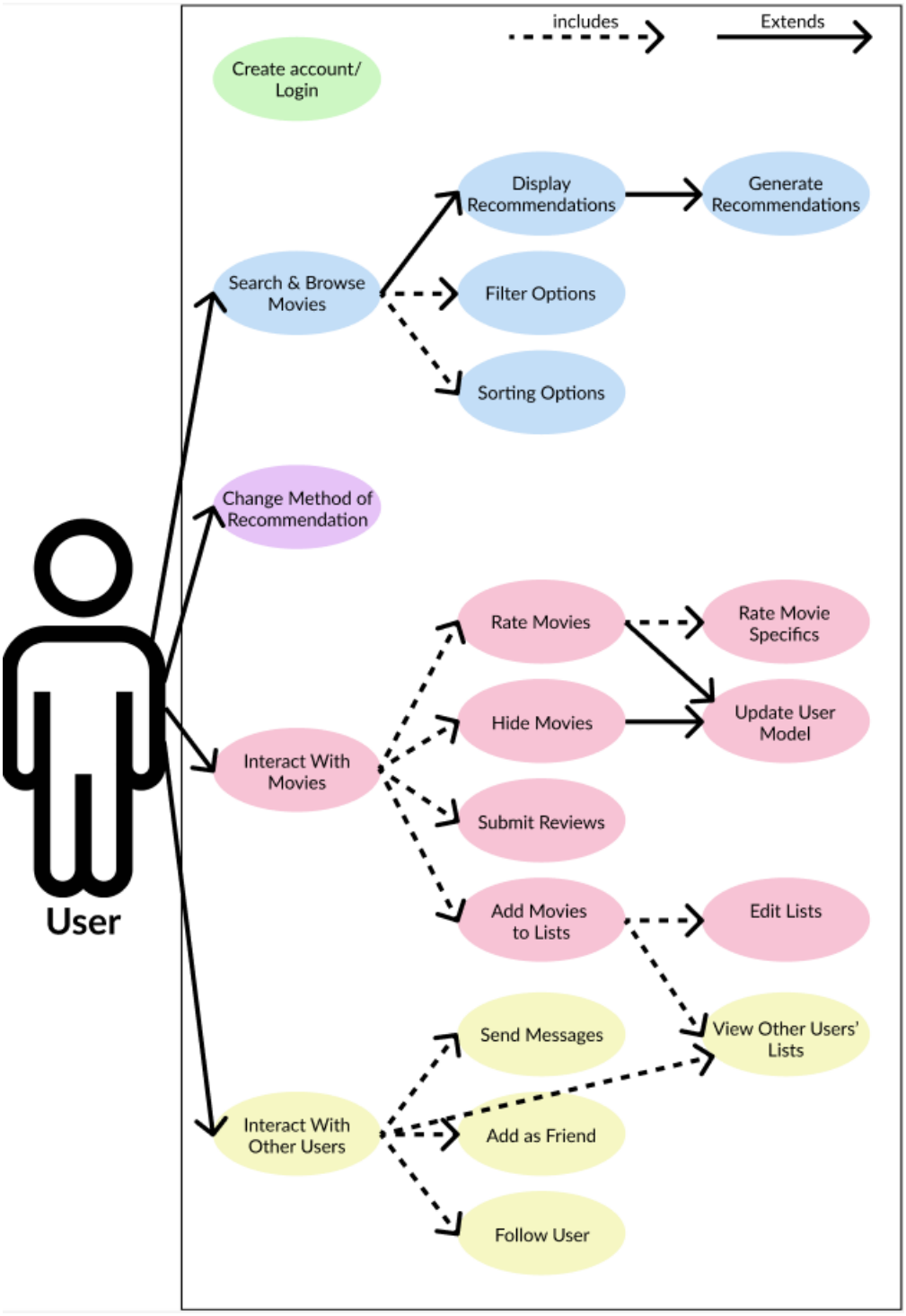


Figure 29: Use case diagram for movie recommendation application

2.4 Feasibility

This section describes which technologies are planned to be used in the development of the application. It then explains if there are any issues in terms of the technical feasibility of the project, for example, if there are two different types of software which may have compatibility issues.

2.4.1 MongoDB

MongoDB is a free to use, flexible and scalable document database that stores data in JSON-like documents with optional schemas. MongoDB uses automatic indexing and has ad-hoc query capabilities, providing powerful methods of analysing data.

2.4.2 Express

Express is a minimal and flexible open-source backend web application framework for Node.js. Express provides a robust set of features that can be used to create an API quickly and easily. Express is the most popular Node web application framework and the basis of many other popular frameworks.

2.4.3 React

React is a free and open-source frontend JavaScript library for building interactive user interfaces. As one of the most popular frontend libraries, there are a considerable amount of additional libraries made for React, giving developers many different options to work with. React is a component-based library the allows developers to build individual components and bring them together to create complex user interfaces.

2.4.4 Node

Node.js is an open-source, cross-platform JavaScript runtime environment and is designed to build scalable network applications. By running single threaded, asynchronous, non-blocking programming, Node.js can handle server requests faster than its counterparts and allows it to be very memory efficient.

2.4.5 Visual Studio Code

Visual Studio Code (VS Code) is a streamlined code editor that offers support for debugging, syntax highlighting, task running, intelligent code completion, version control and more. With over 30,000 extensions available, developers have a wide range of customisation options to create the environments they need to work effectively and efficiently.

2.4.6 GitHub

GitHub is hosting service for code that specialises in version control and collaboration. Using GitHub allows multiple people to work on projects simultaneously from around the world.

2.4.7 Anaconda

Anaconda is an open-source platform that allows developers to write and execute Python code and is the most popular platform for creating scientific computing, data science and machine learning applications with Python. Anaconda offers extensive package options, with 250 packages automatically installed and over 7,500 more open-source packages available for installation. Anaconda Navigator, one of the preinstalled packages, is a desktop graphical user interface that allows developers to launch other applications such as Jupyter Notebook or Visual Studio Code.

2.4.8 Jupyter Notebook

Jupyter Notebook is a web-based interactive development environment for creating and sharing notebooks, code and data. It offers a simple, streamlined, document-centric experience.

2.4.9 Pandas

Pandas is a powerful, fast, flexible and easy to use open-source Python library for data analysis and manipulation.

2.4.10 NumPy

NumPy is an open-source Python library used for the creation of multi-dimensional arrays and matrices and offers an extensive collection of high-level mathematical functions that can be applied to those arrays.

2.4.11 Scikit-Learn

Scikit-Learn is a free to use, open-source machine learning library for Python build on NumPy, SciPy and matplotlib. It features a variety of classification, regression, clustering, dimensionality reduction, model selection and pre-processing algorithms.

2.5 Conclusion

In this chapter, similar applications were researched, interviews were conducted, requirements were defined and the technologies to create the application were decided upon.

A variety of similar applications were researched, each with their own different, but valuable qualities that could be implemented in the intended application. MovieLens showed how a simple recommendation system can be effective, IMDb proved how valuable extensive movie information can be to users and MyAnimeList highlighted how important a community focused system can be to people.

The interviews and surveys were conducted with less people than preferred, but some valuable information could be taken away from them regardless. They gave some insight into what features people might want to see in a movie recommendation application, the information that's important to them when deciding on a movie to watch and how they would like to manage information in such an application.

Lists of functional and non-functional requirements for the intended application were then more easily determined and sorted in order of priority based on the research conducted. With lists of requirements defined, the technologies necessary to make an application with those requirements were selected and explained.

3 Chapter Three: Design

3.1 Introduction

With the application requirements defined, this chapter will explore both the program design and user interface design elements of the application, the options available, the best practices and the chosen directions.

The technologies chosen for the development of this application will be defined and justified, with aspects of the most important technologies being farther expanded on and explained. With the technologies defined, the architecture of the application and its database system can then be developed.

The user interface for the application will describe how the application will be formatted and displayed, with the goal of being as user friendly an experience as possible. Typefaces and colour schemes suitable for the applications purposes and with good design elements are chosen with the help of online tools.

3.2 Program Design

The program design refers to the design required to make the task of programming and coding of the application more straightforward.

3.2.1 Technologies

The technologies being used to create this application are:

- MongoDB
- ExpressJS
- ReactJS
- NodeJS
- Flask
- Pandas
- NumPy
- Scikit-Learn
- Visual Studio Code
- Anaconda
- Jupyter Notebook
- Insomnia
- GitHub

MongoDB, ExpressJS, ReactJS and NodeJS form the MERN stack, a grouping of JavaScript technologies that excels at fast and efficient application development. Flask was used as the Python backend because of how easily it can connect with a React application. Pandas and NumPy libraries were used to manipulate data into acceptable formats for the AI model, which was made using the scikit-learn library. Visual Studio Code and Anaconda/Jupyter Notebook were used as code editors for JavaScript and Python respectively. Insomnia was used to test API requests, and finally GitHub was used as an online repository.

Possible alternatives to these technologies include but are not limited to:

- MEAN/MEVN stacks instead of the MERN stack, which replaces ReactJS with either AngularJS or VueJS in the stack. The MERN stack was chosen over these options because React offered more capability then Vue, but didn't have as steep a learning curve as Angular.
- Django over Flask. Django is a powerful full-stack Python framework, but this application does not need such an extensive framework, so the lightweight framework, Flask, was used instead.
- TensorFlow instead of Scikit-learn. Scikit-learn was chosen for its flexibility, adaptability and classification and clustering options.
- Any number of code editors over Visual Studio Code and Anaconda/Jupyter Notebook. There are a great many options when selecting code editors, these ones were chosen because they're at the top of their field and are familiar to the developer.
- Postman instead of Insomnia. Both have almost the exact same capabilities, any differences were negligible for the intended purposes, Insomnia was chosen because of familiarity.

3.2.2 Structure of React and Express

3.2.2.1 React

The creators of React don't have any specific recommendations on how developers should structure their projects, but there are a few common approaches that are popular and should be considered. There are however two popular methods among developers of structuring projects, by file feature or by file type.

When structuring by file type, files should be grouped inside folders that have been designated for specific features in the project, though what constitutes a 'feature' is subjective and inconsistent across the development community.

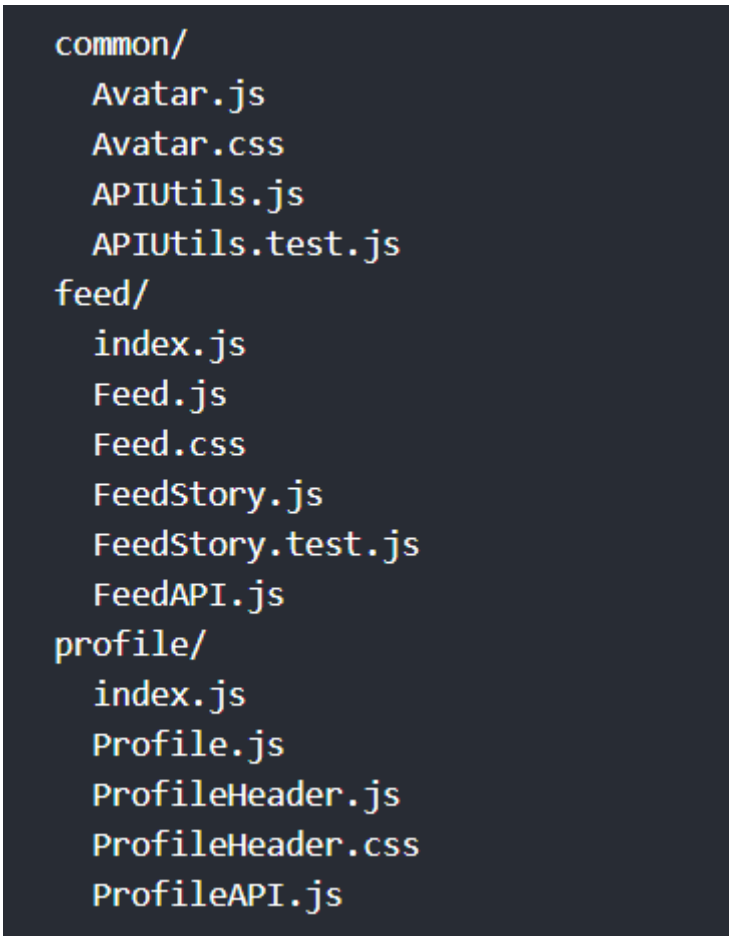


Figure 30: React structuring via grouping files by features. Image taken from ReactJS.org.

Another common method of structuring projects is to group files together based on file type (not file extension). For example, files concerning the API would be contained in an API folder, a middleware folder would contain all of the middleware files and all the components would be in a component folder. Some developers might use subfolders to subdivide the files even farther, especially in larger projects with many files.

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

Figure 31: React structuring via grouping files by file type.Image taken from ReactJS.org.

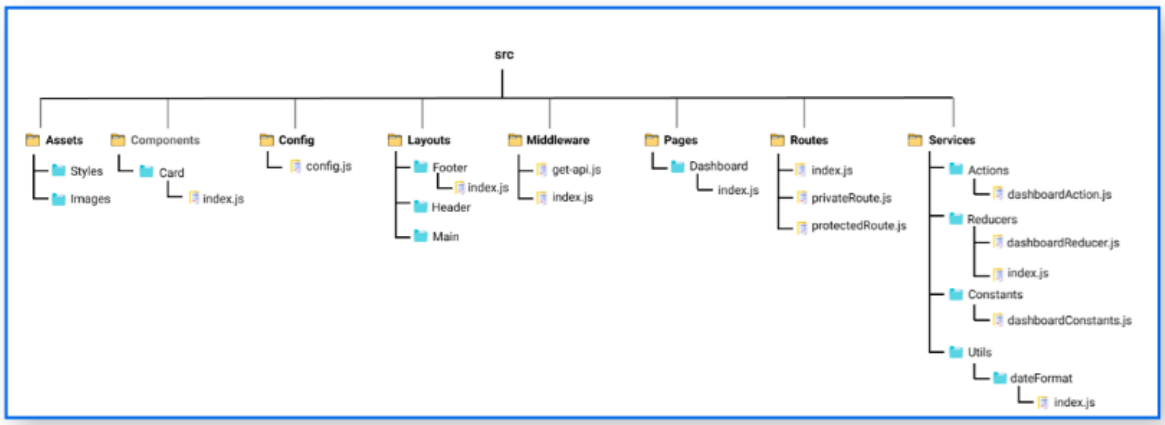


Figure 32: React structuring vie grouping files by file type #2. Image taken from XenonStack.com.

3.2.2.2 ExpressJS

The ExpressJS folder structure should be first and foremost predictable. Projects should be formatted in ways so that it's easy for the developer and outside parties to locate code in the source files. Folder and files names should be as clear, relevant and accurate as possible.

Depending on the size of the application being developed, the ExpressJS folder structure can range from quite simple to rather complex. A small-scale application with not very many files doesn't need

an in-depth folder structure, for example, a looser structure with a few general folders with 3 or so files each is often better and easier to work with than 9 very specific folders each containing a single file.

In ExpressJS, the structure of files and folders can impact the applications functionality, the order of middleware and routes matters and can break an application if structured incorrectly and clashes between different middlewares or routes occur. Very important application-wide middleware should come first, then all the routes and route middleware and lastly the error handlers.

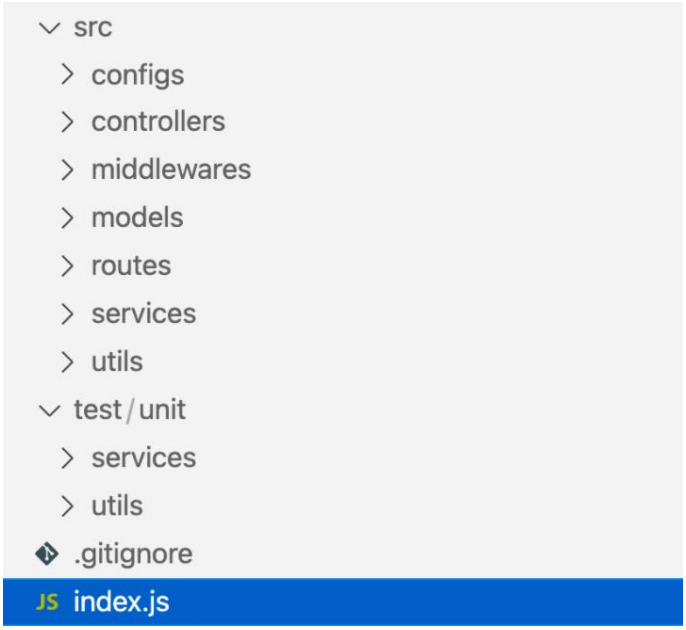


Figure 33: ExpressJS predictable folder structure.

3.2.3 Design patterns

As a UI library, React doesn't have a definitive design pattern that it's based on, but there have been numerous design patterns created and popularised among the community, these design patterns can vary from patterns that effect a single aspect of the application to patterns that effect the entire application.

3.2.3.1 Flux

One of the popular application-wide design patterns for React is Flux, which manages how data goes through a React application. Flux follows the concept of Unidirectional Data Flow, which can be very useful for applications that use dynamic data. Flux methodology has three primary roles for dealing with data: 'dispatcher', 'stores' and 'views'(React components).

The dispatcher acts as a central hub, a registry of callbacks into the stores and it manages the data flow of a Flux application. The dispatcher waits to be called by an action, and when called sends the action to all stores.

Similar to a traditional MVC model, the stores contain the applications state and logic. Stores update themselves when an action is dispatched and produce the change event to notify the controller view.

Views, also known as controller-views, is situated at the end of the chain to contain the logic to generated actions and take in new data from the stores. It listens for change events and re-renders the application after receiving data from the stores.

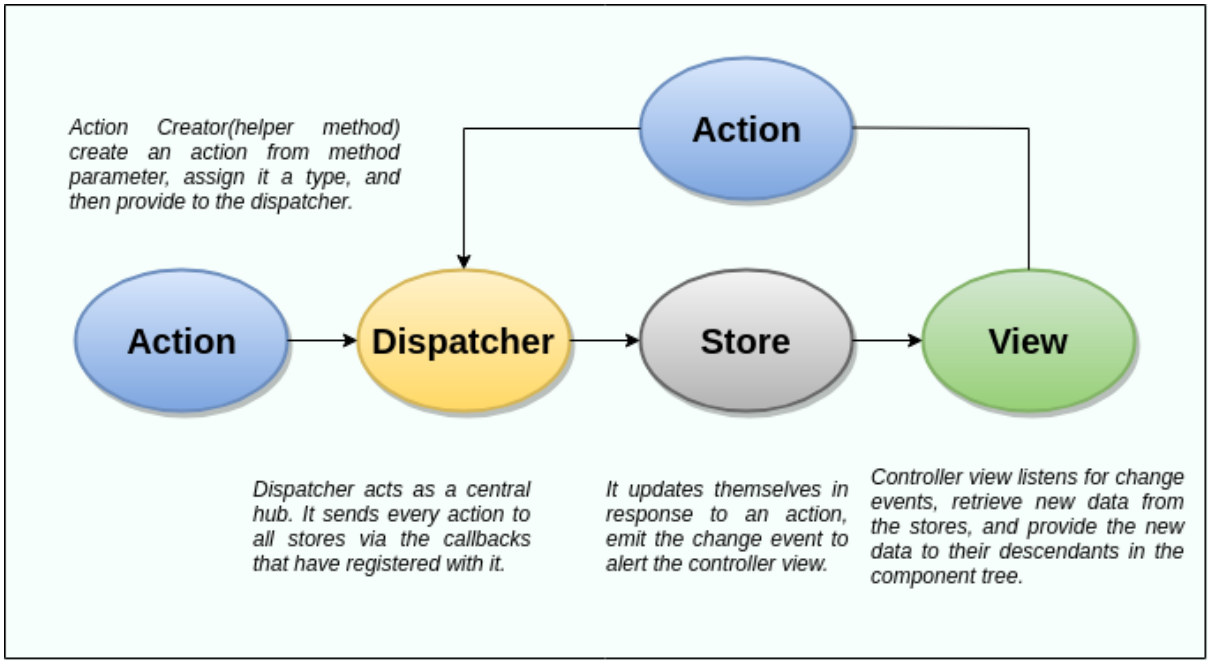


Figure 34: Flux data flow chart. Taken from Javatpoint.com.

3.2.3.2 Higher order component pattern (HOC)

A HOC pattern is an advanced React pattern used to reuse component logic throughout the application, React documentation defines HOC as "A function that takes in a component and returns a new component". HOC functions take components as arguments and return another component based on the taken component after adding data or changing functionality. Reusing components like this means that similar components don't have to be coded manually, saving time and space.

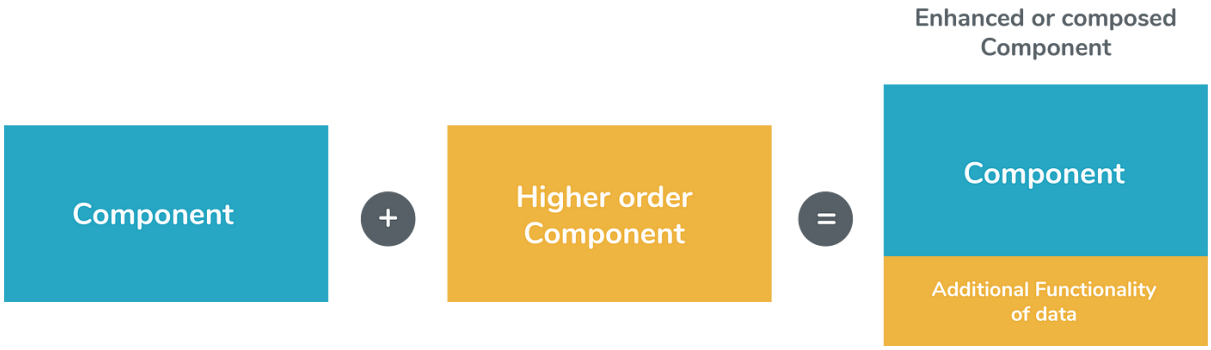


Figure 35: How HOC design pattern works. Component + HOC function = altered component.

3.2.4 Application architecture

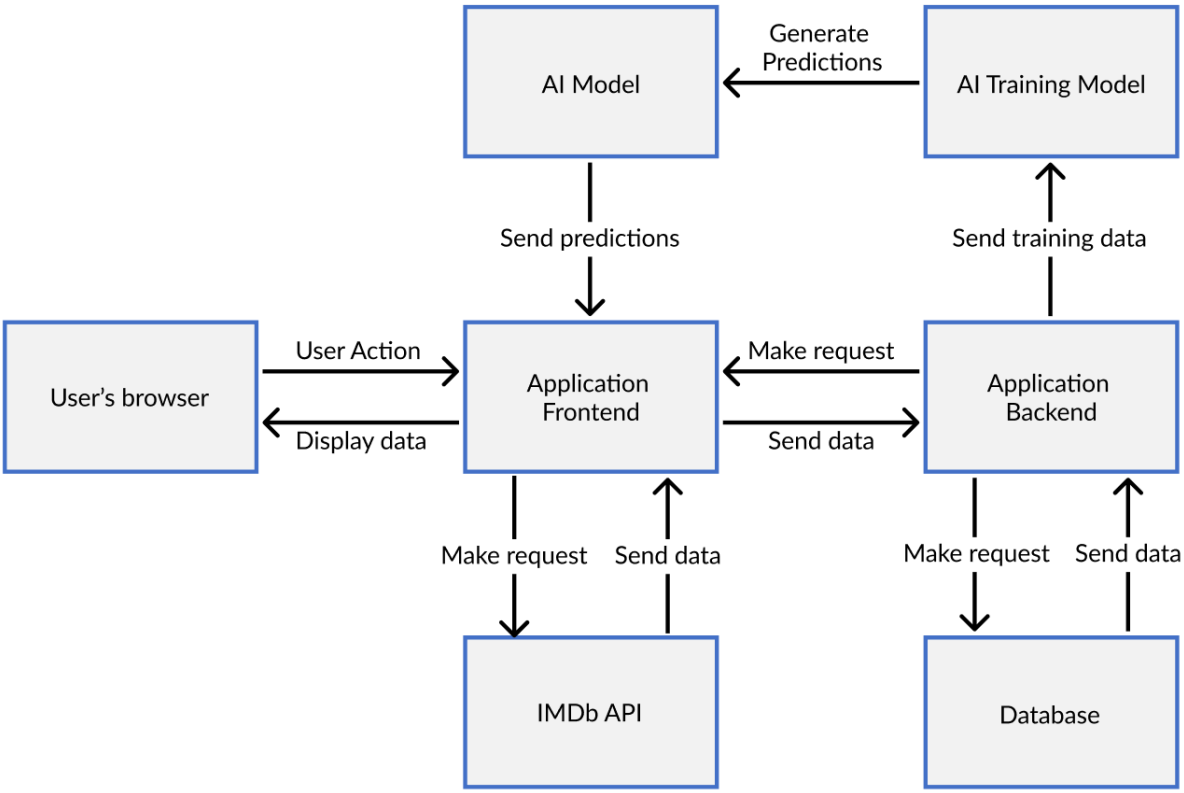


Figure 36: Planned application architecture for a movie recommendation application.

3.2.5 Database design

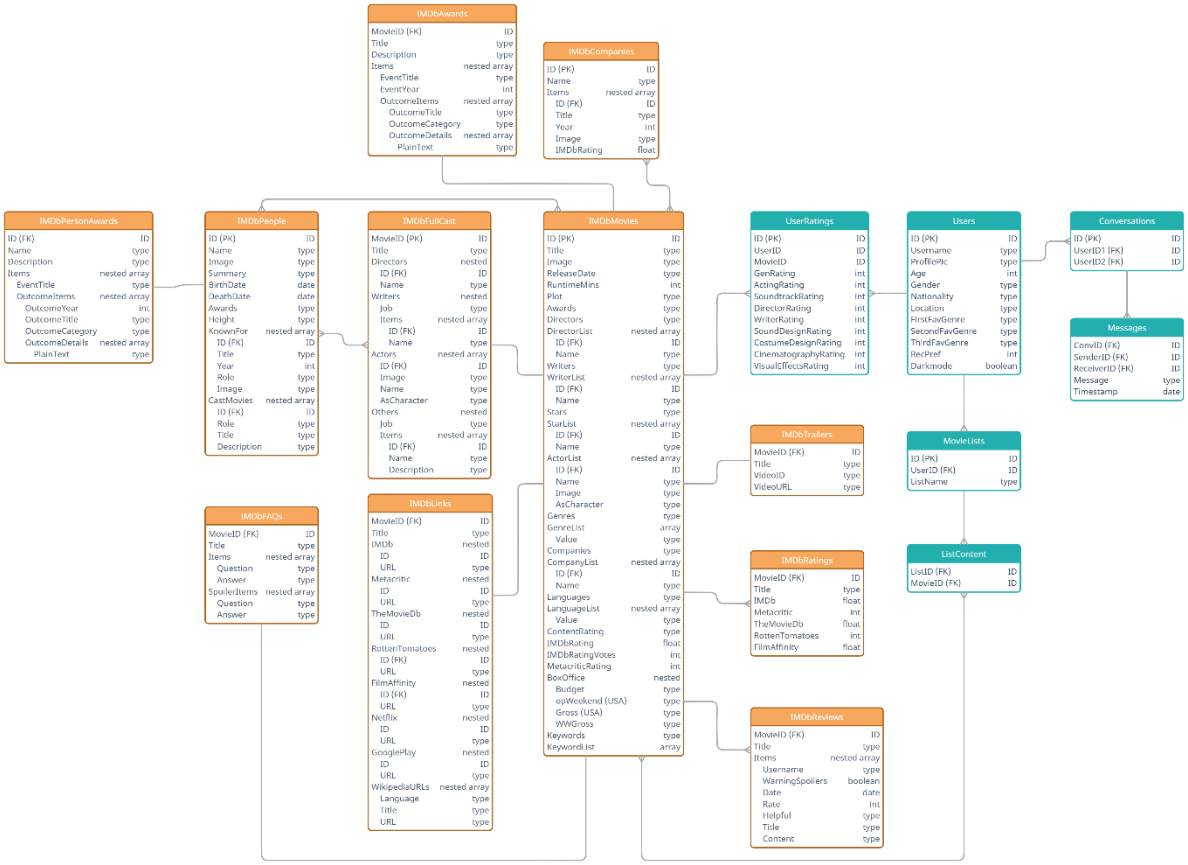


Figure 37: Database design, created using Creately.

The database for the application will be hosted on MongoDB to make use of their embedded data models, which allows for the embedding of additional related data in the same database record. In the diagram shown in figure 37 above, the blue boxes signify the Express/MongoDB backend, while the orange boxes signify the IMDb database accessed via IMDb's API.

3.3 User interface design

3.3.1 Hi-Fi prototype

A Hi-Fi prototype for the applications primary page was created using the free online design tool, Figma. The design was based on the needs of the application and the research carried out on similar applications.

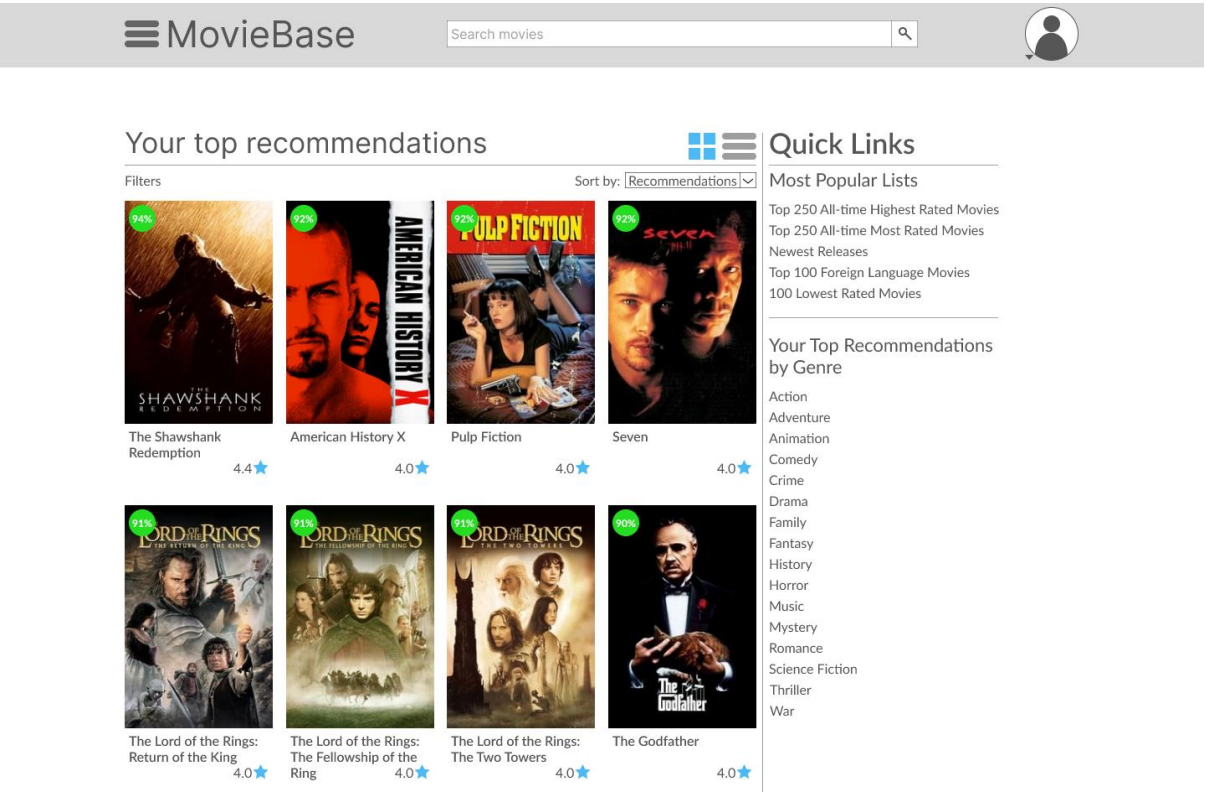


Figure 38: Hi-Fi prototype homepage.

As can be seen above in figure 38, the homepage shows the logged in user a selection of recommended movies in a card format, which can be changed to a list format, recommendations can be filtered and sorted in a variety of ways and pre filtered lists can be selected from the side bar. Figure 39 below shows how the homepage looks when changed to list view.

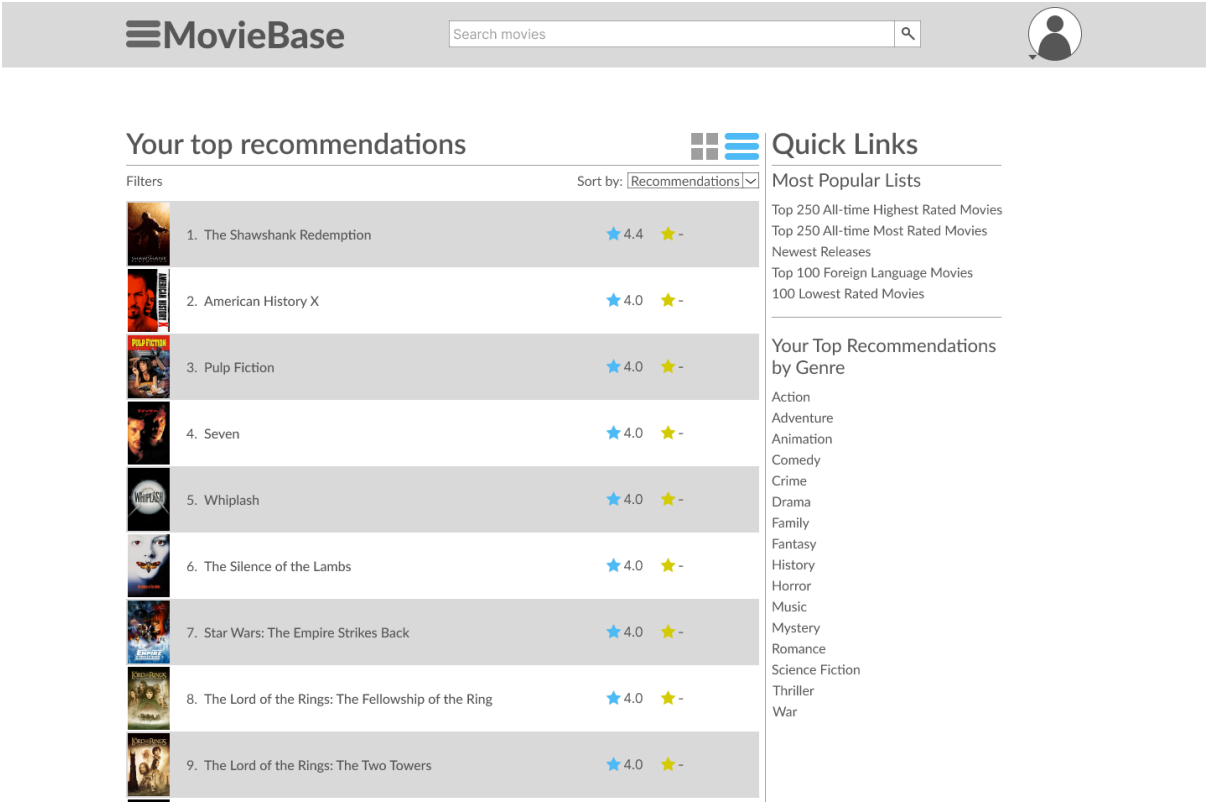


Figure 39: Recommendations in list view.

Figure 40: Extra movie information is shown on card hover.

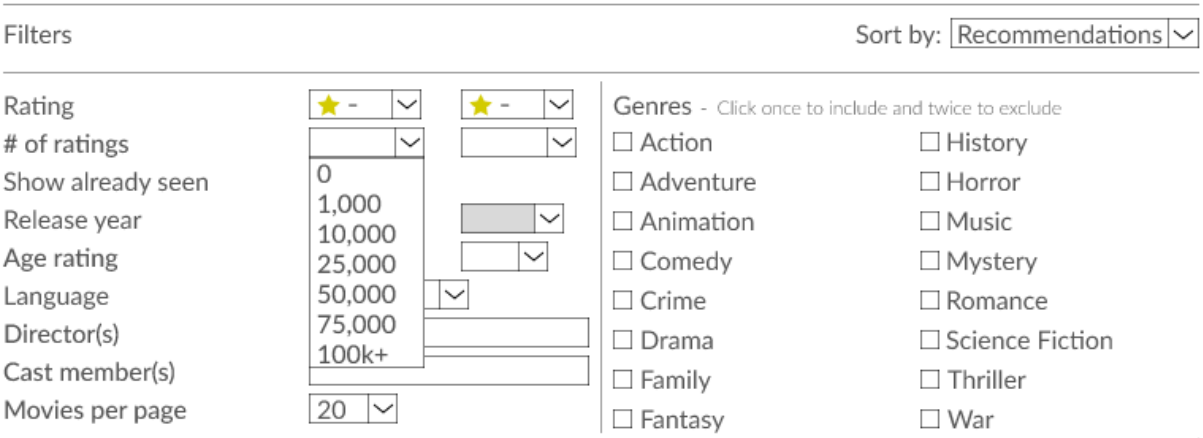


Figure 41: Filter menu.

As can be seen above in figure 41, a filter menu drops down when clicked, offering a variety of filtering options. Movies can be filtered by information such as a minimum, maximum or range of rating scores, rating counts and/or release dates, they can be filtered by an entered director or cast member, or even by the selecting genre(s) recommendations must (or must not) be a part of.

3.3.2 User flow diagram

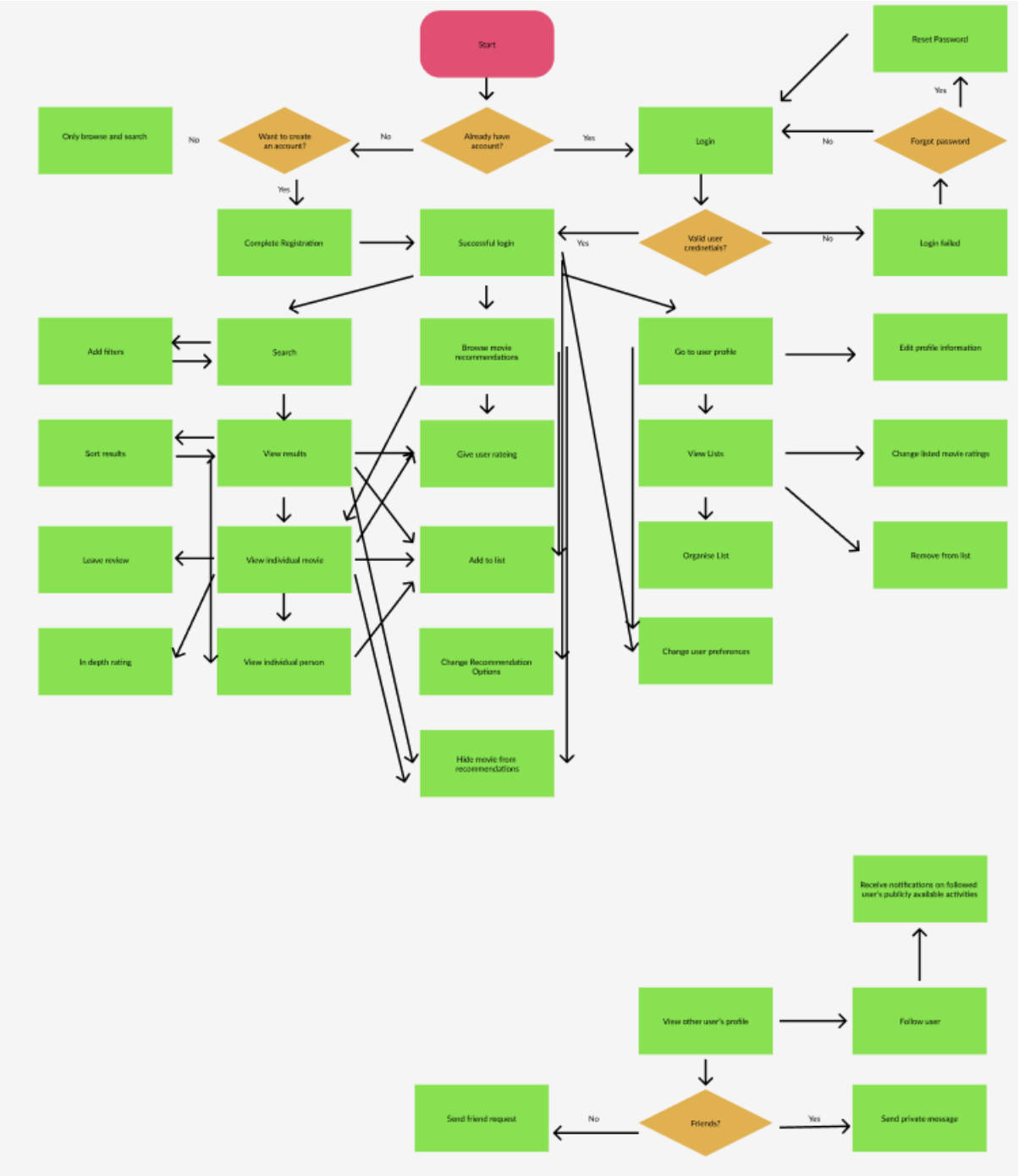


Figure 42: User flow diagram for movie recommendation application.

The user flow diagram shown above in figure 42 showcases the tasks a user can do in the application, and the processes required to do them.

3.3.3 Style guide

3.3.3.1 Typography

Priorities when selecting a font for a web application include style, legibility, load speed and format options. The font must be Sans Serif, it must be easy to read while looking nice, load on the page as quickly as possible and have a wide variety of scaling and weight capabilities. Two fonts that meet all of these requirements are Lato and Roboto. Different fonts could be used in different parts of the application, such as for heading text compared to body text, but for a consistent design, Roboto was chosen to be the font for this application.

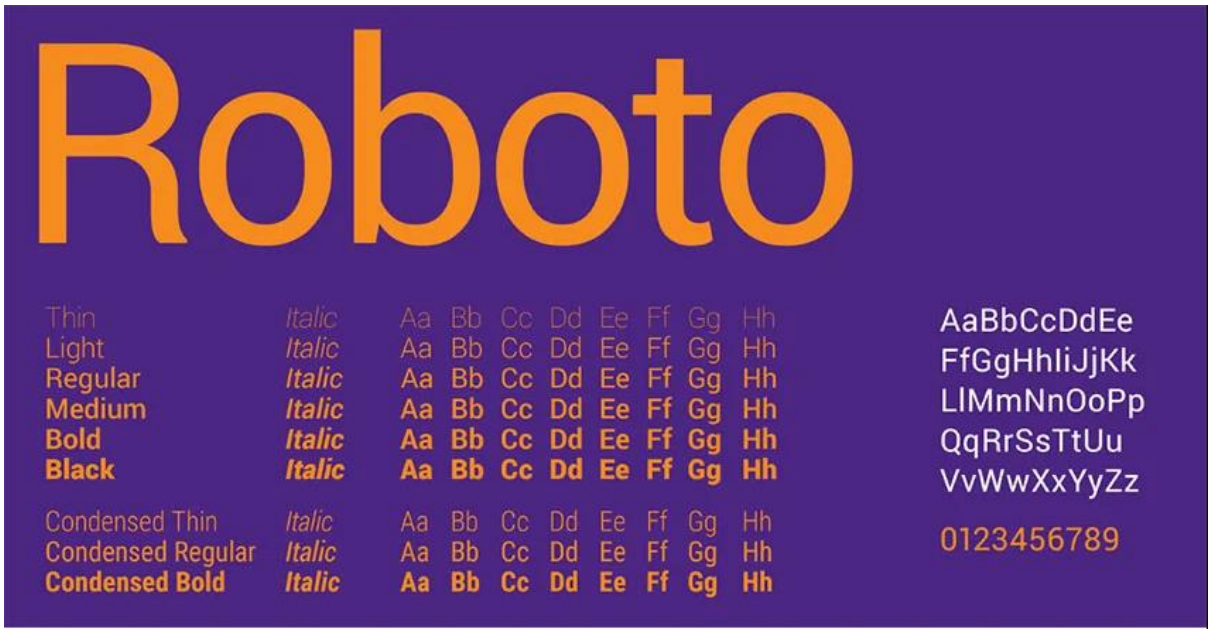


Figure 43: Image showcasing Roboto font from DafontFile.com.

3.3.3.2 Colour scheme

A colour scheme with an eye-catching colour and other complimentary colours was chosen. From these colours, dark and light versions of the application can be developed.

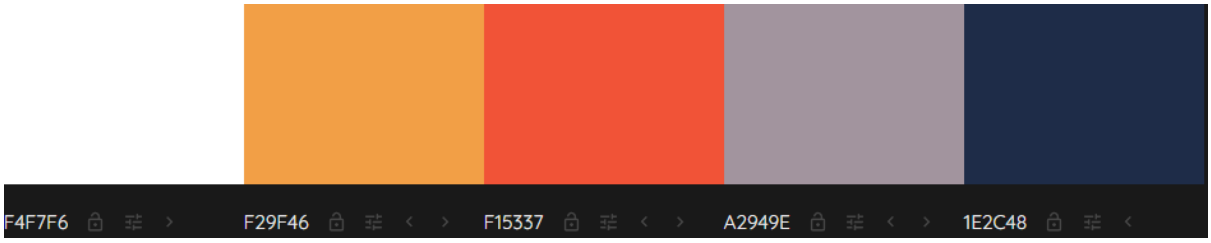


Figure 44: Colour scheme generated using Colormind.io.

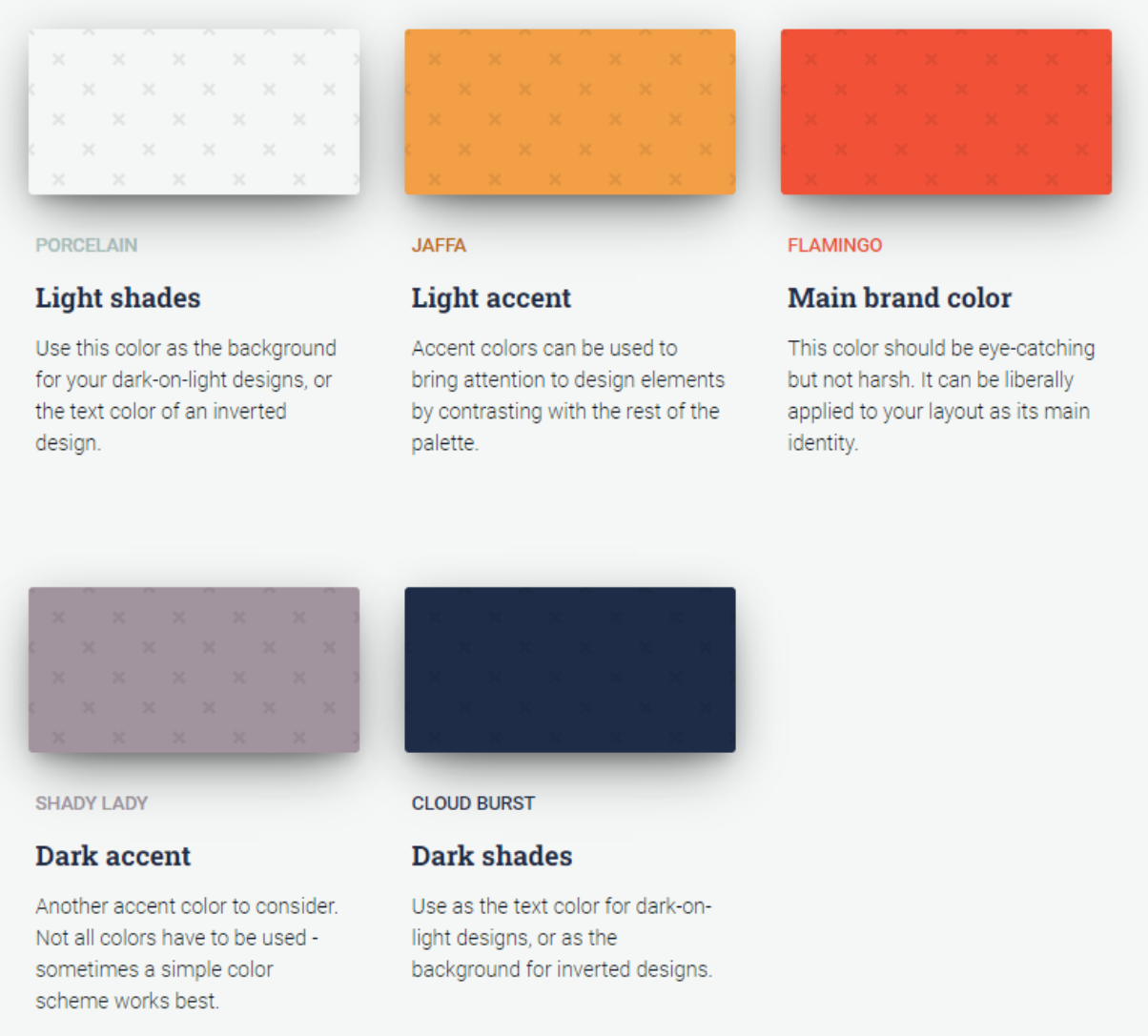


Figure 45: Explanation of each colour's purpose, from Colormind.io.

3.4 Conclusion

When determining the technologies that would be used to make this application, some areas had many great, viable options that are difficult to choose between, such as Angular/Vue or React, or Scikit-learn or TensorFlow, while some other areas were decided for the project by the project, for example, the rest of the MERN stack, MongoDB, Express and Node or the python libraries Pandas and NumPy had their places in the development of the application cemented.

Folder structures for React and Express were defined with a heavy priority being placed on predictability to help in creating an efficient and easily memorable development environment. Reacts flexibility offered many good potential design patterns to follow, while Express' design pattern was simpler and more straight forward.

The application architecture shows the structure of the entire application, and how each separate part interacts with each other, the database design combined the Express backend/MongoDB database with IMDb's API, creating a link between the two systems by storing movie's IMDb IDs in the MongoDB movie collection and the user flow diagram shows the different ways a user can use the application.

4 Chapter Four: Implementation

4.1 Introduction

This application was created following a sprint development cycle, each sprint was two weeklong and had specific goals that were to be attained over the two period. Over the course of the development of this application, the following technologies were used:

- React

ReactJS is an open-source JavaScript framework library developed by Meta (formerly known as Facebook). It's used for frontend web development and is often used in the development of single-page applications.

- Express

ExpressJS is an open-source, backend web application framework based in JavaScript for Node.js, it excels at working with RESTful APIs.

- Flask

Flask is a web application framework written in Python, it is based on the Werkzeg WSGI toolkit and the Jinja2 template engine.

- MongoDB

MongoDB is a free to use, scalable and flexible document database. MongoDB stores data in JSON-like documents and allows object mapping between database collections.

- Anaconda/Jupyter Notebook

Anaconda is a free, open-source platform for Python development that allows users to launch other applications, manage libraries, packages and environments, and develop new Python applications.

- Figma & Creately

Figma and Creately are two online design tools that can be used individually or collaboratively to design things such as relationship or database diagrams, infographics, or website's user interface/experience prototypes.

- Axios

Axios is an isomorphic, promise-based HTTP client based in JavaScript for web browsers and node.js. Axios handles requests being sent from an application's frontend to a server.

- Material UI

Material UI is an open-source component library for React that implements Google's Material Design. Material UI offers a comprehensive collection of prebuilt components that are easy to integrate into many different kinds of projects.

- Pandas

Pandas is a Python library use for data analysis and manipulation, primarily the structuring and manipulation of tables

- NumPy

NumPy is a Python library used for the creation and management of multi-dimensional arrays and matrices and offers a wide range of complex mathematical functions.

- Scikit-learn

Scikit-learn is a machine learning library for Python that offers a variety of classification, regression and clustering machine learning algorithms such as support-vector machines (SVM), k-nearest neighbours (KNN), random forests, gradient boosting and more.

The application for this project is called MovieBase, it was developed using the technologies listed above. MovieBase is a web application where users can go to receive personalised, AI generated movie recommendations that have been determined by a model based on the KNN recommendation algorithm. As users rate more movies, their ratings are added to the dataset that is passed through the AI model, improving the accuracy of their recommendations.

4.2 Sprint Methodology

A sprint system was followed over the course of this project and consisted of eight official sprints in total, six before the Easter break and two after, with the easter break itself serving as an unofficial additional sprint. Each sprint had their own goals and covered a two-week period. The first three sprints focused mostly on research, preparation and other report aspects of the project, the fourth and fifth sprints focused on developing and testing prototype applications, the sixth, easter break, and seventh sprints mostly focused on the development of the application, leaving the eighth sprint to finish anything that needed to be finished in the report and bug fix.

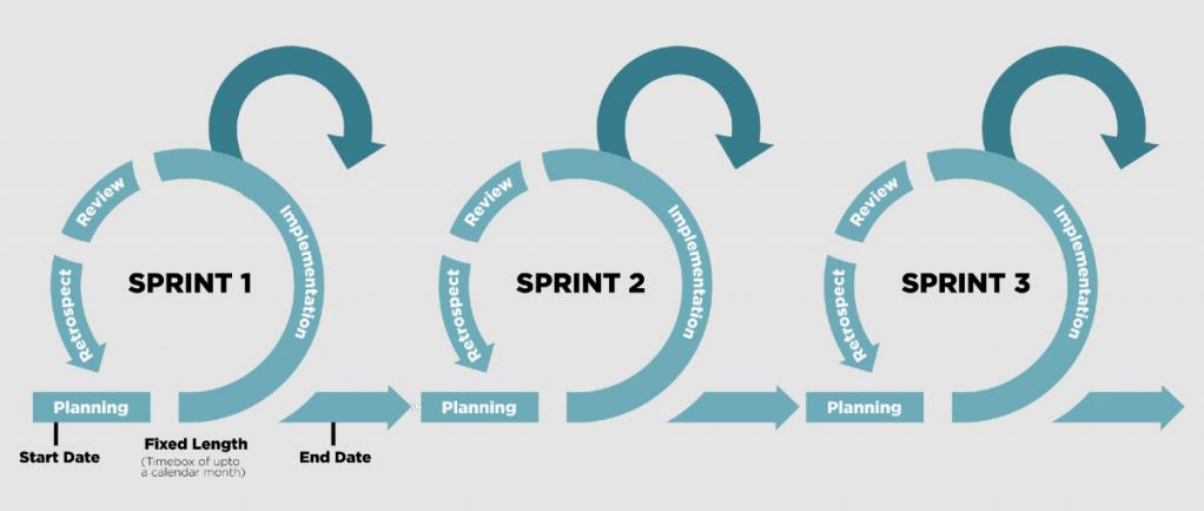


Figure 46: Diagram showing sprint cycles. Image from Niftypm.com.

4.3 Development Environment

- Visual Studio Code

Visual Studio Code was the code editor used for the entire development of the React based frontend and the Express based backend parts of the application, and also for some of the Flask based backend part of the application.

- Anaconda/Jupyter Notebook

Through Anaconda, Jupyter Notebook was used for the manipulation and structuring of different datasets as well as the development of the AI models and their interaction with the properly structured datasets to return accurate predictions.

- Git BASH

Git BASH was used to run Git and console commands from the command line to run/manage the various parts of the application and to interact with GitHub.

- GitHub

GitHub was used to host various versions of the application throughout the development cycle and to upload/download the application from and to different computers so it could be worked on with multiple machines simultaneously.

- Insomnia

Insomnia was used during the development of the application to test API requests and to more easily and clearly see exactly how data was being received.

- Microsoft Word

Microsoft Word was used to keep a log of coding activity throughout the development cycle of the application.

- MongoDB/Compass

MongoDB was used to store the data with object-based relationship capabilities, while MongoDB Compass was used to easily upload premade sets of data that had been manually structured as needed in Jupyter Notebook.

- Google Chrome/Mozilla Firefox

Chrome and Firefox were used to test how the application works in a browser, and to troubleshoot by using their in-built consoles.

4.4 Sprint 1

4.4.1 Goals

- Research the project area and start research section of the thesis.
- Define a backlog of features for the intended project.
- Create a paper prototype of the intended project.

4.4.2 Item 1: Start research document

The research for this project started with a search for peer-reviewed books, papers and articles relevant to the project area. After much consideration, three books were chosen to be farther researched and used as primary sources of information for the development of this project. These books are:

- 'Practical Recommender Systems' by Kim Falk.
- 'Recommender Systems Handbook' by Francesco Ricci et al.
- 'Recommender Systems: Algorithms and Applications' by P. Pavan Kumar et al.

4.4.3 Item 2: Backlog of features

To determine a backlog of features for the intended application, similar applications were researched, and various aspects of their features and functionality were considered for the intended application. The similar applications researched were:

- IMDB.com
- MovieLens.org
- MyAnimeList.net
- Netflix.com
- Primevideo.com

4.4.4 Item 3: Paper prototype

The aforementioned researched websites were also used as inspiration in the creation of several paper prototypes, for both desktop views and mobile views.

4.5 Sprint 2

4.5.1 Goals

- Finish the first version of the research section of the thesis.
- Start the requirements section of the thesis.
- Finish defining intended features.
- Create a Hi-Fi prototype.

4.5.2 Item 1: Finish research document

All four of the books chosen as primary sources of information were researched and their information referenced in the research chapter, as well as several papers and articles. The recommendation systems that YouTube uses for their videos and Amazon use for their marketplace were researched and referenced too.

4.5.3 Item 2: Start requirements document

Several of the similar applications previously researched were researched in more depth, cataloguing their different functionalities, weighing up their advantages and disadvantages and determining what would be good to and what would be possible to implement into my intended application.

Interviews and surveys were created to help determine the best direction for the intended application, how users would like to interact with such an application.

4.5.4 Item 3: Finish defining backlog of features

From the websites that were being researched for intended features, the determined strongest parts of each of them were

From IMDB, their extensive user base, catalogue of movies and movie information highlighted features such as search and filter as well as user-movie interaction functionalities that could be used in the intended application

From MovieLens, their focus on recommendations showcased features that would be useful to the intended application, such as handling new users who aren't in the system by asking for favourite genres on sign up to start the recommendation process, or hiding movies that have already been rated by the user.

From MyAnimeList, they showed how a community focused website can do things differently, focusing on user-user interaction and the managing of user created lists.

From Netflix and Primevideo, their organisation and methods of displaying movies with only the most important information in user friendly ways.

4.5.5 Item 4: Create Hi-Fi prototype

An interactive, Hi-Fi prototype of the homepage was created using Figma

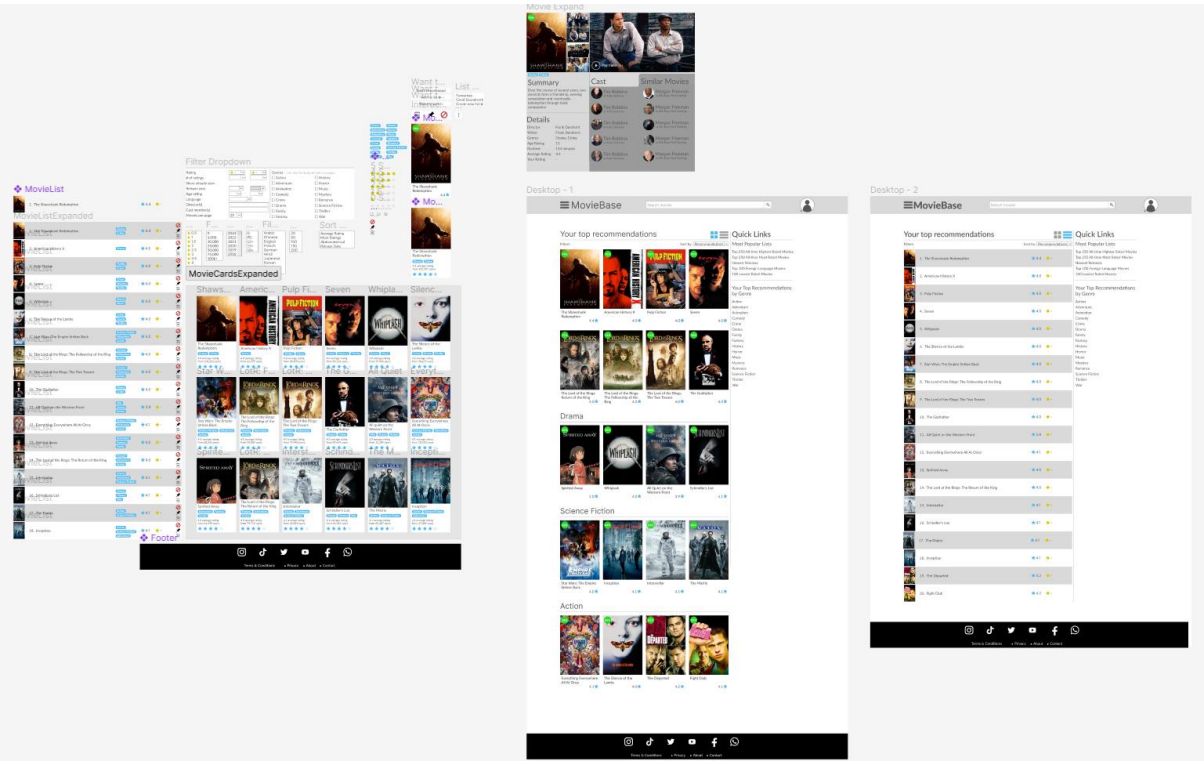


Figure 47: Hi-Fi protype made in Figma

4.6 Sprint 3

4.6.1 Goals

- Finish the first version of the requirements section of the thesis.
- Start the design section of the thesis.
- Research and test methods of python integration with React.

4.6.2 Item 1: Finish requirements document

The interviews were carried out with several people via online calls, their answers were manually recorded in a document and the audio of the calls was also recorded so that they could be revisited if necessary.

The survey was posted online and given to several people to complete in their own time if they were able.

Valuable information was gained from the interviews and survey that could be implemented in the intended application, such as additional filter and search options or what information is important to them when determining what movie they want to watch.

4.6.3 Item 2: Start design document

Design patterns for React were researched, suitable typography and colour schemes were chosen and application architecture, database design, use flow and use case diagrams were created.

4.6.4 Item 3: Python integration

Different potential options of integrating Python into a React application were researched and considered, such as Django, but Flask was determined to be most suited to the applications needs. An online guide was successfully followed to develop a test application in React and Flask using TensorFlow that determined whether a picture was of a cat or a dog.

```
app = Flask(__name__) # new

if __name__ == '__main__':
    app.run(debug=True, host="127.0.0.1", port=5000)
```

Figure 48: App.py: How the Flask app is set up to run on a local server (at port 5000).

```
@app.route('/upload', methods=['POST'])
def upload():
```

Figure 49: App.py: How routing works with Flask/Python. Post requests sent to 'localhost:5000/upload' will run the function 'upload()'.

```
model = load_model("keras_Model.h5", compile=False)
```

Figure 50: App.py: The AI model is loaded in as 'model'.

```
# Predicts the model
prediction = model.predict(data)
index = np.argmax(prediction)
class_name = class_names[index]
confidence_score = prediction[0][index]

# Print prediction and confidence score
print("Class:", class_name[2:], end="")
print("Confidence Score:", confidence_score)

return {"message": class_name + "1"}
```

Figure 51: App.py: The data is passed through the model, printed in file and the result returned to the frontend.

```
const onSubmit = async e => {
  e.preventDefault();
  const formData = new FormData();
  formData.append('file', file);
  console.log(file);

  fetch('http://localhost:5000/upload', {
    method: 'POST',
    body: formData
  })
    .then(res => res.json())
    .then(data => {
      console.log(data);
      setPrediction(data);
    });
};
```

Figure 52: App.js: The request to the Flask backend from the React frontend which also saves the returned data with 'setPrediction'.

```
{prediction && <h1>{prediction.message}</h1>}
```

Figure 53: App.js: Displaying the prediction in the client.

4.7 Sprint 4

4.7.1 Goals

- Finish the first version of the design section of the thesis.
- Make a small-scale application prototype in Flask/React.

4.7.2 Item 1: Finish design document

Roboto was selected as the typeface, and a colour scheme was generated using Colourmind.io, an online, deep learning colour scheme generator.

4.7.3 Item 2: Application prototype – Making the AI model

After successfully building a Flask application by following a tutorial, a small prototype of the intended application was developed following a similar path as the tutorial. Before starting on the application, datasets of movies and user ratings needed to be acquired, and a model needed to be made, trained and exported. Small datasets were obtained from GroupLens' MovieLens system.

```
In [2]: movies_df = pd.read_csv('movies.csv')
        ratings_df = pd.read_csv('ratings.csv')
```

Figure 54: Jupyter Notebook: Reading in the MovieLens datasets.

```
In [9]: combined_df = pd.merge(movies_df, ratings_df)
```

```
In [12]: combined_df
```

Out[12]:

	movieId	title	userId	rating
0	1	Toy Story (1995)	1	4.0
1	1	Toy Story (1995)	5	4.0
2	1	Toy Story (1995)	7	4.5
3	1	Toy Story (1995)	15	2.5
4	1	Toy Story (1995)	17	4.5
...	...	...	...	...
100831	193581	Black Butler: Book of the Atlantic (2017)	184	4.0
100832	193583	No Game No Life: Zero (2017)	184	3.5
100833	193585	Flint (2017)	184	3.5
100834	193587	Bungo Stray Dogs: Dead Apple (2018)	184	3.5
100835	193609	Andrew Dice Clay: Dice Rules (1991)	331	4.0

100836 rows × 4 columns

Figure 55: Jupyter Notebook: Merging the datasets and result.

```
In [16]: movieUser_df = refined_dataset.pivot(
             index='userId',
              columns='title',
               ## Replacing all movies users haven't rated with a rating of 0
               values='rating').fillna(0)
```

```
In [17]: movieUser_df
```

Out[17]:

title	'71 (2014)	'Hellboy': The Seeds of Creation (2004)	'Round Midnight (1986)	'Salem's Lot (2004)	'Til There Was You (1997)	'Tis the Season for Love (2015)	'burbs, The (1989)	'night Mother (1986)	(500) Days of Summer (2009)
userId									
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
606	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
607	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
608	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
609	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
610	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.5

Figure 56: Jupyter Notebook: Creating a pivoted table with both movies and users on their own axis, and ratings as the datapoints.

```
In [18]: movieUser_scipy_df = csr_matrix(movieUser_df.values)
```

```
In [19]: knn_model = NearestNeighbors(metric='cosine', algorithm='brute')
         knn_model.fit(movieUser_scipy_df)
```

```
Out[19]: NearestNeighbors(algorithm='brute', metric='cosine')
```

Figure 57: Jupyter Notebook: Turning the pivoted table into a SciPy sparse matrix, setting up the KNN recommendation method and then training the model on the scipy sparse matrix.

```
In [19]: from joblib import Parallel, delayed
         import joblib

In [20]: joblib.dump(knn_model, 'firsttest.h5')
Out[20]: ['firsttest.h5']
```

Figure 58: Jupyter Notebook: Using Joblib to export the trained AI model.

4.7.4 Item 3: Application prototype – The Flask backend

After the AI model was trained and exported in Jupyter Notebook, it could be imported to the Flask backend, along with some pre-formatted datasets, to be used in the application prototype.

```
# Load model
knn_model = joblib.load('firsttest.h5')

# Load csv(s)
movieUser_df = pd.read_csv('movieUser.csv')
refined_dataset = pd.read_csv('refined.csv')
```

Figure 59: App.py: Importing the trained AI model and loading already manipulated datasets.

```
target_user = int(request.form["target_user"])
no_of_highest = int(request.form["no_of_highest"])
no_of_similar_users = int(request.form["no_of_similar_users"])
no_of_movies = int(request.form["no_of_movies"])
```

Figure 60: App.py: Receiving form data from the frontend. Had to be encased in int() or else it treated all received data as string, even if it was sent from frontend as an int.

```
def similar_users(user, n = 5):
    knn_input = np.asarray([movieUser_df.values[user-1]])
    distances, indices = knn_model.kneighbors(knn_input, n_neighbors=n+1)

    print("Top ", n ," users who are most similar to the user #",user, " are:", sep="")
    print("_____")

    for i in range(1,len(distances[0])):
        print(i,". User #", indices[0][i]+1, " separated by distance of ",distances[0][i],sep="")
    return indices.flatten()[1:] + 1, distances.flatten()[1:]
```

Figure 61: App.py: The function to determine the most similar users based on the given user's movie ratings by comparing the given users ratings with all other user's ratings.

```
# Calling the similar_user function, providing the target user to check and how many similar users to find (5)
# Passing similar users to similar_user_list and similar user's distances from the target user to ditance_list
similar_user_list, distance_list = similar_users(target_user, no_of_similar_users)

# Adding weights to similar user's ratings depending on their distance from the target user
weighted_list = distance_list/np.sum(distance_list)

# Storing all of the ratings submitted by the users determined as most similar
similar_user_ratings = movieUser_df.values[similar_user_list]

# Adding a column vector, increasing dimensions of weighted_list by adding axis of movies from movies_list
weighted_list = weighted_list[:,np.newaxis] + np.zeros(len(movie_list))

# Adding the weights to user ratings and creating a new list of mean, weighted ratings
ratings_matrix = weighted_list*similar_user_ratings
mean_ratings_list = ratings_matrix.sum(axis =0)
```

Figure 62: App.py: Manipulating the data so it's formatted properly to be able to generate recommendations based on the determined similar users.

```
def recommend_movies(n):
    n = min(len(mean_ratings_list),n)
    pprint(list(movie_list[np.argsort(mean_ratings_list)[::-1][:n]]))
```

Figure 63: App.py: The function that determines movie recommendations based on similar users.

```
recommend_movies2(no_of_movies)
print_output = buffer.getvalue()

return {"message": print_output}
```

Figure 64: App.py: Calling the recommend_movies function with the amount of movies to be recommended, and then returning that data to the frontend.

4.7.5 Item 4: Application Prototype – React frontend

The React frontend just needed to consist of a form, a fetch request, and the ability to receive and display data.

```
const onSubmit = async e => {
  e.preventDefault();

  const formData = new FormData();
  formData.append("target_user", document.getElementById("target_user").value)
  formData.append("no_of_highest", document.getElementById("no_of_highest").value)
  formData.append("no_of_similar_users", document.getElementById("no_of_similar_users").value)
  formData.append("no_of_movies", document.getElementById("no_of_movies").value)

  fetch('http://localhost:5000/upload', {
    method: 'POST',
    body: formData
  })
    .then(res => res.json())
    .then(data => {
      console.log(data);
      setPrediction(data);
    });
};
```

Figure 65: App.js: Fetch request to the Flask backend

```
<form onSubmit={onSubmit}>
  <div className='custom-file'>
    <label for="target_user">target_user</label>
    <input type="number" id="target_user" name="target_user" />
    <br />
    <br />
    <label for="no_of_highest">no_of_highest_rated_movies_by_target_user</label>
    <input type="number" id="no_of_highest" name="no_of_highest" />
    <br />
    <br />
    <label for="no_of_similar_users">no_of_similar_users</label>
    <input type="number" id="no_of_similar_users" name="no_of_similar_users" />
    <br />
    <br />
    <label for="no_of_movies">no_of_movies_to_recommend</label>
    <input type="number" id="no_of_movies" name="no_of_movies" />
  </div>
  <input
    type='submit'
    value='Submit'
    className='btn btn-primary btm-block mt-4'
  />
</form>
{prediction && <h1>{prediction.message}</h1>}
```

Figure 66: App.js: Form that's submitted and displaying the returned results

4.8 Sprint 5

4.8.1 Goals

- Expand Application Prototype.

4.8.2 Item 1: Expanding application prototype – Flask backend

After successfully implementing a custom movie recommendation model into the Flask tutorial, the application could be farther expanded upon to receive more recommendation data and allow the data to be formattable on reception.

```
# Creating empty arrays that will hold the data
simUsers = []
userDistances = []
highestMovies = []
recommendedMovies = []
```

Figure 67: App.py: Declaring empty arrays to hold the data that will be generated.

```
# Find most similar users to target user
def similar_users(user, n = 5):
    # Convert values to numpy array and pass through model and output values for user and distance
    knn_input = np.asarray([movieUser_df.values[user-1]])
    distances, indices = knn_model.kneighbors(knn_input, n_neighbors=n+1)

    for i in range(1,len(distances[0])):
        # Passing the data to empty arrays
        simUsers.append(indices[0][i]+1)
        userDistances.append(distances[0][i])
    return indices.flatten()[1:] + 1, distances.flatten()[1:]
```

Figure 68: App.py: Storing the similar users and their distances from the target user.

```
# Function that outputs the top n movies based on ratings of similar users from the mean rating list
def recommend_movies(n):
    n = min(len(mean_ratings_list),n)
    recommendedMovies.append(list(movie_list[np.argsort(mean_ratings_list)[::-1][:n]]))
    return recommendedMovies
```

Figure 69: App.py: Storing the recommendations in the empty array as separate objects now so that they can be individually interacted with when passed to React.

```
# Store arrays in a dictionary to transform into JSON
dict = {'Users_Top_Movies': highestMovies, 'Similar_Users': simUsers, 'Sim_User_distances': userDistances, 'Recommendatons': recommendedMovies}
```

Figure 70: App.py: Storing all the arrays in a dictionary in JSON format, to be sent to the React frontend.

4.8.3 Item 2: Expanding application prototype – React frontend

With the data now being sent from the Flask backend formatted as JSON, the data can be received in React and used much more easily and in many more ways.

```
const formatTop = () => {
    let top = [];
    for (let i = 0; i <= document.getElementById("no_of_highest").value; i++) {
        top.push(<p>{prediction.data.Users_Top_Movies[0][i]}</p>);
    }
    return top;
};
```

Figure 71: App.js: An example of how the data is being used in React, each item is being pushed to an empty array in paragraph tags.

```
const formatRec = () => {
    let rec = [];
    // let indicator = 1;
    // let shitest = [];
    let no_movies = document.getElementById("no_of_movies").value;
    // let imdbHolder = "1";

    for (let i = 1; i <= no_movies; i++) {
        fetch(`https://imdb-api.com/en/API/SearchMovie/k_zvyi75fh/${prediction.data.Recommendatons[0][i]}`, {
            method: 'GET',
        })
        .then(res => res.json())
        .then(data => {
            global.imdbHolder = data.results[0].id;
        });

        rec.push(<p>{prediction.data.Recommendatons[0][i]} - <a href={'https://www.imdb.com/title/' + global.imdbHolder}>IMDb link</a></p>);
    }

    return rec;
};
```

Figure 72: App.js: Taking the returned movie recommendation titles, and using the IMDb API to search the title and save the id of the first returned result.

Users Top Movies

Pinocchio (1940)

Schindler's List (1993)

Once Were Warriors (1994)

Pulp Fiction (1994)

Snow White and the Seven Dwarfs (1937)

Users Most Similar to Target User

User #8, seperated by a distance of 0.5397268757781548

User #58, seperated by a distance of 0.5787738640350151

User #38, seperated by a distance of 0.6167428250351248

User #46, seperated by a distance of 0.6191847021390342

User #43, seperated by a distance of 0.623090924867261

Recommendations Based on Similar Users

Star Wars: Episode IV - A New Hope (1977) - IMDb link

Star Wars: Episode VI - Return of the Jedi (1983) - IMDb link

E.T. the Extra-Terrestrial (1982) - IMDb link

Silence of the Lambs, The (1991) - IMDb link

Psycho (1960) - IMDb link

Figure 73: Output of recommendation from Flask backend displayed in React.

4.9 Sprint 6

4.9.1 Goals

- Start developing final application.

4.9.2 Item 1: New datasets

After developing a relatively simple prototype, it became clear that for the final application, it was going to be necessary to use datasets that have IMDb's unique ID (tconst) for each item. This is so the datasets the AI model is working with can be reliably connected with the data in the MongoDB collections and from IMDb API requests via this unique ID. Datasets for all of the titles in IMDb's database and all titles average ratings are offered officially by IMDb, and contain over 9.5 million unique titles, but a dataset for individual user ratings is still needed. There are a few options available, individuals who have scraped the data from IMDb's website, turned it into a dataset and made it available for free, the user ratings dataset that was chosen is created by Vahid Baghi and contained almost 4.7 million ratings from 1.5 million unique users on 350,000 different movies.

With all necessary datasets on hand, they can be manipulated and formatted to fit with the AI model.

```
In [2]: ## Reading the CSVs
movie_titles_df = pd.read_csv('Title_basics.tsv', dtype={"isAdult": "string"}, sep='\t')
movie_ratings_df = pd.read_csv('Title_ratings.tsv', sep='\t')
user_ratings_df = pd.DataFrame(np.load('User_ratings.npy'))

## convert loaded np array to pd dataframe
user_ratings_df[['userID', 'titleID', 'rating', 'date']] = user_ratings_df[0].str.split(',', expand=True)
user_ratings_df = user_ratings_df.drop(0, axis=1)
```

Figure 74: Jupyter Notebook: Reading the datasets as CSVs.

When reading the datasets as in figure 74, the 'Title_basics' dataset was having trouble dealing with the boolean of 'isAdult', as it was a column that would be dropped anyway, it was set to read it as a string. The 'user_ratings' dataset was a NumPy array file, and had to be converted to a Pandas dataframe. Loading all 3 files took a few minutes.

```
In [3]: movie_titles_df
```

Out[3]:

	tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres
0	tt0000001	short	Carmencita	Carmencita	0	1894	\N	1	Documentary,Short
1	tt0000002	short	Le clown et ses chiens	Le clown et ses chiens	0	1892	\N	5	Animation,Short
2	tt0000003	short	Pauvre Pierrot	Pauvre Pierrot	0	1892	\N	4	Animation,Comedy,Romance
3	tt0000004	short	Un bon bock	Un bon bock	0	1892	\N	12	Animation,Short
4	tt0000005	short	Blacksmith Scene	Blacksmith Scene	0	1893	\N	1	Comedy,Short
...	...	...	...	...	...	...	...	...	...
9689096	tt9916848	tvEpisode	Episode #3.17	Episode #3.17	0	2010	\N	\N	Action,Drama,Family
9689097	tt9916850	tvEpisode	Episode #3.19	Episode #3.19	0	2010	\N	\N	Action,Drama,Family
9689098	tt9916852	tvEpisode	Episode #3.20	Episode #3.20	0	2010	\N	\N	Action,Drama,Family
9689099	tt9916856	short	The Wind	The Wind	0	2015	\N	27	Short
9689100	tt9916880	tvEpisode	Horrid Henry Knows It All	Horrid Henry Knows It All	0	2014	\N	10	Adventure,Animation,Comedy

9689101 rows × 9 columns

Figure 75: Jupyter Notebook: The unchanged movie titles dataset.

```
In [4]: movie_titles_df = movie_titles_df.drop(movie_titles_df[movie_titles_df.titleType.isin(
            ["tvEpisode", "videoGame", "video", "tvSpecial", "tvShort", "tv", "tvSeries", "tvMiniSeries", "tvPilot", "short"]
        )].index)
```

Figure 76: Jupyter Notebook: Dropping rows based if they matched these column values.

Most of the over 9.5 million rows in the dataset weren't actually movies and could be dropped, this brought the row count down to 780,000 titles. The only columns needed for now were 'tconst' and 'primaryTitle', the rest could be dropped.

```
In [8]: user_ratings_df
```

Out[8]:

	userID	titleID	rating	date
0	ur4592644	tt0120884	10	16 January 2005
1	ur3174947	tt0118688	3	16 January 2005
2	ur3780035	tt0387887	8	16 January 2005
3	ur4592628	tt0346491	1	16 January 2005
4	ur3174947	tt0094721	8	16 January 2005
...	...	...	...	...
4669815	ur0581842	tt0107977	6	16 January 2005
4669816	ur3174947	tt0103776	8	16 January 2005
4669817	ur4592639	tt0107423	9	16 January 2005
4669818	ur4581944	tt0102614	8	16 January 2005
4669819	ur1162550	tt0325596	7	16 January 2005

4669820 rows × 4 columns

Figure 77: Jupyter Notebook: The unchanged user ratings dataset. 'Date' was the only unnecessary column, and could be dropped, while 'titleID' was renamed to 'tconst'.

```
In [13]: combined_df = pd.merge(movie_titles_df, user_ratings_df)

In [14]: combined_df
```

Out[14]:

	titleID	primaryTitle	userID	rating
0	tt0000147	The Corbett-Fitzsimmons Fight	ur2483625	8
1	tt0000147	The Corbett-Fitzsimmons Fight	ur1234929	8
2	tt0000574	The Story of the Kelly Gang	ur1609079	10
3	tt0000574	The Story of the Kelly Gang	ur13283282	10
4	tt0000574	The Story of the Kelly Gang	ur10334028	9
...	...	...	...	...
3559107	tt9916270	Il talento del calabrone	ur0430892	5
3559108	tt9916270	Il talento del calabrone	ur126474842	2
3559109	tt9916270	Il talento del calabrone	ur24536688	6
3559110	tt9916362	Coven	ur66663169	10
3559111	tt9916428	The Secret of China	ur22484170	8

3559112 rows × 4 columns

Figure 78: Jupyter Notebook: Merging the movie title dataset with the user rating dataset. The combined dataset contained over 3.5 million rows, containing ratings on 150,000 unique titles from 1.2 million unique users.

```
In [21]: movieUser_df = refined_dataset.pivot_table(
             index='userID',
              columns='primaryTitle',
               ## Replacing all movies users haven't rated with a rating of 0
               values='rating').fillna(0)

C:\Users\Fitzi\anaconda3\lib\site-packages\pandas\core\reshape\reshape.py:130: RuntimeWarning: overflow encountered in long_sca
lars
  num_cells = num_rows * num_columns
```

Figure 79: Jupyter Notebook: Trying to make a pivot table from the merged dataset, but running into an error.

With the dataframes now merged, the combined dataframe could then be turned into a pivot table. However, as shown in figure 79, the size of the dataframe was too large, after a few minutes it would return with an error saying “IndexError: index 875914235 is out of bounds for axis 0 with size 875909652”and the process aborted, this is a Pandas bug that has been known about for over 5 years and occurs when handling very large datasets like this, there is no solution.

With such huge datasets, many of the processes involved in structuring and manipulating the data had large memory demands and were taking multiple minutes each to complete, if they completed at all. Some processes were too demanding and required more than the available 16GB of memory that was on hand, which would cause the process to crash. In order to deal with this issue, the datasets would have to be reduced in size.

```
In [8]: ## Marging movie titles with movie ratings
        movie_rating_merge_df = pd.merge(only_movie_titles_df, movie_ratings_df, on="tconst")
```

Figure 80: Jupyter Notebook: Merging the two dataframes on ‘tconst’.

First all of the rows that weren't movies were removed again from the movie titles dataset, but most of the columns were kept for now, as they would be needed later when uploading the final dataset to a MongoDB collection for later use. The movie titles dataset was then merged with the movie ratings dataset as shown in figure 80, which contained each movie's average rating and quantity of user ratings.

```
In [10]: ## Converting list to pd.Series before stack() breaks the pd.Series into a multi-index Series
         genres = movie_rating_merge_df.genres.apply(pd.Series).stack()

         ## Creating a dummy multi-index dataframe where each genre is a column
         genres = pd.get_dummies(genres)

         ## Collapse the dataframe
         genres = genres.groupby(level=0).sum()
```

```
In [11]: ## merging the dummy dataframe with the movie_rating_merge_df dataframe
         movie_rating_merge_df = pd.concat([movie_rating_merge_df, genres], axis=1)
```

Figure 81: Jupyter Notebook: The unique genres were taken and turned into columns with each datapoint becoming a Boolean.

```
In [12]: ## Creating new dataframes with minimum rating count breakpoints to be exported as seperate CSV files
         title_merge_ratings_10 = movie_rating_merge_df[movie_rating_merge_df['numVotes'] > 10]
         title_merge_ratings_100 = movie_rating_merge_df[movie_rating_merge_df['numVotes'] > 100]
         title_merge_ratings_1000 = movie_rating_merge_df[movie_rating_merge_df['numVotes'] > 1000]
         title_merge_ratings_10000 = movie_rating_merge_df[movie_rating_merge_df['numVotes'] > 10000]
         title_merge_ratings_100000 = movie_rating_merge_df[movie_rating_merge_df['numVotes'] > 100000]
         title_merge_ratings_500000 = movie_rating_merge_df[movie_rating_merge_df['numVotes'] > 500000]
```

Figure 82: Jupyter Notebook: Creating multiple dataframes of various sizes that can be exported as CSVs and be storable in the MongoDB collection, but will need to be farther manipulated to be usable by the AI model.

```
In [13]: title_merge_ratings_500000
```

Out[13]:

	tconst	titleType	primaryTitle	startYear	runtimeMinutes	genres	averageRating	numVotes	Action	Adult	...	News	Reality-TV	Romance	Sci-Fi	Sh
12252	tt0034583	movie	Casablanca	1942	102	[Drama, Romance, War]	8.5	578015	0	0	...	0	0	1	0	
22712	tt0050083	movie	12 Angry Men	1957	96	[Crime, Drama]	9.0	801467	0	0	...	0	0	0	0	
25784	tt0054215	movie	Psycho	1960	109	[Horror, Mystery, Thriller]	8.5	679732	0	0	...	0	0	0	0	
30145	tt0060196	movie	The Good, the Bad and the Ugly	1966	161	[Adventure, Western]	8.8	769762	0	0	...	0	0	0	0	
31798	tt0062622	movie	2001: A Space Odyssey	1968	149	[Adventure, Sci-Fi]	8.3	677625	0	0	...	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
306015	tt6966692	movie	Green Book	2018	130	[Biography, Comedy, Drama]	8.2	506565	0	0	...	0	0	0	0	
307352	tt7131622	movie	Once Upon a Time in Hollywood	2019	161	[Comedy, Drama]	7.6	761699	0	0	...	0	0	0	0	
308733	tt7286456	movie	Joker	2019	122	[Crime, Drama, Thriller]	8.4	1321768	0	0	...	0	0	0	0	
318986	tt8579674	movie	1917	2019	119	[Action, Drama, War]	8.2	607576	1	0	...	0	0	0	0	
321465	tt8946378	movie	Knives Out	2019	130	[Comedy, Crime, Drama]	7.9	713136	0	0	...	0	0	0	0	

293 rows × 36 columns

Figure 83: How the filtered dataframes are structured.

The example in figure 83 is the result of filtering out titles that had less than 500,000 total user ratings, bringing the row count down from 780,000 to less than 300. For comparison, the dataframe that filtered out movies with less than 10 total user ratings had 290,000 rows, less than 1,000 had 42,000 rows and less than 100,000 had 2,250 rows.

```
In [10]: user_ratings_df_trimmed = user_ratings_df.groupby('userID').filter(lambda x : len(x)>14)
```

Figure 84: Jupyter Notebook: Filtering the user ratings dataset.

The user ratings dataset in figure 84 was filtered to remove users that had less than 15 total ratings. This lowered the row count from almost 4.7 million down to just over 2.1 million. This may not seem like a very big drop, but the unique user count dropped from 1.5 million to just 29,500. Not only does this dramatically reduce the size of a future necessary pivot table, but it improves AI accuracy by only considering users that have given a minimum number of ratings when determining similar users and subsequently movie recommendations.

```
In [22]: movieUser_df = refined_dataset.pivot_table(
             index='userID',
              columns='titleID',
               ## Replacing all movies users haven't rated with a rating of 0
               values='rating').fillna(0)

In [23]: movieUser_df

Out[23]:
```

titleID	tt0034583	tt0050083	tt0054215	tt0060196	tt0062622	tt0066921	tt0068646	tt0071562	tt0071853	tt0073195	...	tt5463162	tt6320628	tt6644200	tt6
userID															
ur0000011	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur0000039	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur0000066	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur0000157	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur0000685	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
ur99955002	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur99964320	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur99965244	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur99966337	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
ur99999734	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	

21177 rows × 287 columns

Figure 85: Jupyter Notebook: After merging the newly filtered datasets, pivoting the table now works and results in a table with 287 columns (unique movies) and 21,000 rows (unique users).

```
In [20]: dict1 = {'Users_Top_Movies': highestMovies, 'Similar_Users': simUsers, 'Sim_User_distances': userDistances, 'Recommendatons': re

In [21]: dict1

Out[21]: {'Users_Top_Movies': [['tt0121766',
                               'tt0093058',
                               'tt0121765',
                               'tt3659388',
                               'tt0120915']],
          'Similar_Users': [20392, 17018, 612, 10902, 6681],
          'Sim_User_distances': [0.2340139075168851,
                                 0.24029849559731942,
                                 0.2868343211902684,
                                 0.3214285714285714,
                                 0.3423596141160732],
          'Recommendatons': [['tt0066921',
                              'tt0133093',
                              'tt0264464',
                              'tt0167261',
                              'tt0167260']]}
```

Figure 86: Jupyter Notebook: With the data now structured properly, it can be used by the AI model and generate recommendations in the form of the movies IMDb ID.

4.9.3 Item 2: Setting up Express backend registration and login functionality

An Express application was started, and registration and login functionality were implemented.

```
let validateEmail = function (email) {
    let re = /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/;
    return re.test(email)
};
```

Figure 87: User_schema.js: a function to validate entered email addresses.

```
const userSchema = new Schema({
    name: {
        type: String,
        trim: true,
        unique: true,
        //lowercase: true,
        required: [true, 'Name is required.']
    },
    email: {
        type: String,
        trim: true,
        unique: true,
        lowercase: true,
        required: [true, 'Email is required.'],
        validate: [validateEmail, 'Please enter a valid email address.']
    },
    password: {
        type: String,
        trim: true,
        required: [true, 'Password is required.']
    },
    pfp: {
        type: String
    },
}, {
    timestamps: true
});
```

Figure 88: User_schema.js: A standard user schema was created, requiring a name, email and password to be stored in a MongoDB collection, with the password calling the validateEmail function to verify whether the entered email is formatted correctly or not.

```
userSchema.methods.comparePassword = function (password) {
    console.log("password - " + password)
    console.log("this.password - " + this.password)
    return bcrypt.compareSync(password, this.password, function (result)
        return result
    });
};
```

Figure 89: User_schema.js: Bcrypt was installed to handle hashing passwords for security. This function compares the saved hashed password with the just entered password to verify a log in.

```
const register = (req, res) => {
    let newUser = new User(req.body)

    newUser.password = bcrypt.hashSync(req.body.password, 10)

    newUser.save((err, user) => {
        if (err) {
            return res.status(400).send({
                message: err
            })
        } else {
            user.password = undefined;
            return res.json(user)
        }
    })
```

Figure 90: User_controller.js: Standard function for post request in user_controller.js with password hash functionality from bcrypt.

```
const login = (req, res) => {
    User.findOne({
        email: req.body.email,
    })
    .then((user) => {
        if (!user || !user.comparePassword(req.body.password)) {
            return res.status(401).json({
                messsage: 'Authentication failed - Invalid name/email or password.'
            });
        }
        // create token
        res.json({
            user,
            userID: user.id,
            token: jwt.sign({
                    email: user.email,
                    name: user.name,
                    _id: user._id
                },
                'Y4Project'
            )
        })
    })
    .catch(err => {
        throw err
    });
}
```

Figure 91: User_controller.js: Standard function to log in. On user log in, finds email matching the one entered and then compares the saved password with the entered one. If everything is a match, the user logs in and a unique token is generated for tracking and identification purposes.

```
require('dotenv').config();
require('./utils/db.js')();

const { register, login} = require('./controllers/user_controller')

app.post('/register', register)
app.post('/login', login)
```

Figure 92: Server.js: The database URL and port number were saved in a .env file and imported to server.js. The register and login functions are imported from user_controller.js and connected to relevant routes.


```
{
  "userID": "641c87d06a97e629837fc079",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RAdGVzdC5jb20
iLCJuYW1lIjoiTWFyY3VzIiwiX2lkIjoiNjQxYzg3ZDA2YTk3ZTYyOTgzN2ZjMDc5Iiwi
aWF0IjoxNjgyMTc1NTA2fQ.2MQGJYfwIDqjxSI9ZV6aWVY5-ChARskz0C-2M7SFJZM"
}
```


Figure 93: Insomnia: A successful user login return.

4.9.4 Item 3: React Registration and Login Functionality

With registration and login functionality set up in the Express backend, a new React app could be initiated and login/registration functionality could be added to the frontend.

```
import axios from 'axios';

export default axios.create({
    baseURL: 'http://localhost:4000'
});
```

Figure 94: index.js: Axios was installed to handle routing and requests to the server.

```
const [form, setForm] = useState({
    email: "",
    password: ""
});
```

Figure 95: LoginForm.js: Setting up a useState with an array of the required login keys with empty values in JSON format.

```
const handleForm = (e) => {
    let name = e.target.name;
    let value = e.target.value;

    setForm(prevState => ({
        ...prevState,
        [name]: value
    }));
};
```

Figure 96: LoginForm.js: A function to set the states keys to the values entered into the form as they're entered.

```
const submitForm = () => {
    console.log("Email: ", form.email);
    console.log("Password: ", form.password);

    axios.post('/login', {
            email: form.email,
            password: form.password
        })
        .then((response) => {
            const users = response.data.user._id
            setErrorMessage("");
            props.onAuthenticated(true, response.data.token);
            localStorage.setItem('userID', users);
            const userID = localStorage.getItem('userID');
            console.log('userID: ', userID)
            console.log("Token: ", response.data.token);

        })
        .catch((err) => {
            console.error(err);
            console.log(err.response.data);
            setErrorMessage(err.response.data.message);
        });
};
```

Figure 97: LoginForm.js: The submitForm function is called when the form is submitted and sends the login data to the backend server.

```
<>
    Email: <input type="text" name="email" value={form.email} onChange={handleForm} />
    <br />
    Password: <input type="password" name="password" value={form.password} onChange={handleForm} />
    <button onClick={submitForm}>Submit</button>
    <p style={styles}>{errorMessage}</p>
    <button onClick={regForm}>Register</button>
</>
```

Figure 98: LoginForm.js: The login form, on any change handleForm is called, and on submission submitForm is called. A register button was added to redirect to the registration form for new users.

The registration form is set up in the exact same way as the login form, except it expects a name as well as the email and password.

4.9.5 Item 4: React navigation menu

To build a navigation menu, navigation components were imported from react-router-dom and design components were imported from Material UI.

```
const [anchorEl, setAnchorEl] = useState(null);
const open = Boolean(anchorEl);

const handleClick = (event) => {
  setAnchorEl(event.currentTarget);
};

const handleClose = () => {
  setAnchorEl(null);
};
```

Figure 99: Navbar.js: Anchor points for nested menu options were set up, but didn't work as intended. All submenus in the navbar would link to the same location.

```
//Dynamically set anchors for multiple buttons/drop downs
const [anchorState, setAnchorState] = useState({
    btn1: null,
    btn2: null,
});

const handleClick = (e) => {
    setAnchorState({ [e.target.name]: e.currentTarget });
};

const handleClose = (e) => {
    setAnchorState({ [e.target.name]: null });
};
```

Figure 100: Navbar.js: Turning the useState into an array with a key for each button/drop down, which solved the problem.

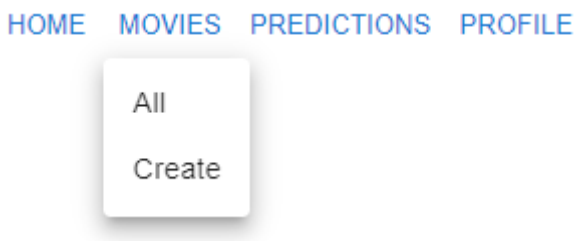


Figure 101: Navbar.js: The created navigation menu

4.9.6 Item 5: React Cards

The movies are to be displayed in a card format and only needs to show the movie's poster, title and rating. Design components from Material UI were used to make cards for the movies.

```
import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Typography from '@mui/material/Typography';
import { Button, CardActionArea, CardActions } from '@mui/material';
```

Figure 102: MovieCard.js: The necessary card components imported from Material UI.

```
<Card id="movieCard" sx={{ minWidth: 220, maxWidth: 220}}>
    <CardActionArea href={`/movies/${props.movie._id}`}>
        <CardMedia
            component="img"
            image={`http://img.omdbapi.com/?i=${props.movie.tconst}&apikey=e729aa7f`}
            width="220"
            height="326"
        />
        <CardContent>
            <Typography id="cardTitle" gutterBottom variant="h5" component="div">
                {title}
            </Typography>
            <Typography id="cardExtra" variant="body2" color="text.secondary">
                <a href={`https://www.imdb.com/title/${props.movie.tconst}`}>{mID}</a>
            </Typography>
        </CardContent>
    </CardActionArea>

    <CardActions>
        {starRating()}
        <AddToListBtn id={props.movie._id} resource='movies' callback={props.callback} />
    </CardActions>
</Card>
```

Figure 103: MovieCard.js: The Card. The movies _id (MongoDB ID), tconst (IMDb ID) and title are taken in to be used in the card. The card uses the passed tconst value to get the movies poster from a third-party API, OMDb API, and also to link to the movies IMDb page for now. The _id is used to create an action area around the card that links to a page of the individual movie that was in the card.

4.10 Sprint 6.5 (Easter Break)

4.10.1 Goals

- Continue development on and expand final application.

4.10.2 Item 1: Expanding Express application

Before adding any more requests in the Express backend, the movie data needed to be added to the MongoDB collection, MongoDB Compass was used to do this by uploading one of the previously exported csv files of filtered movies. The smallest dataset was used, with just under 300 movies being added to the collection. A movie schema was created following the structure of the uploaded dataset. Once that was done, requests could be added to interact with the collection.

```
const getAllMovies = (req, res) => {
    Movie.find()
        .then((data) => {
            if (data) {
                res.status(200).json(data)
            } else {
                res.status(404).json("No movies found.")
            }
        })
        .catch((err) => {
            console.error(err)
            res.status(500).json("None found.")
        });
}

const getSingleMovie = (req, res) => {
    Movie.findById(req.params.id)
        .then((data) => {
            if (data) {
                res.status(200).json(data)
            } else {
                res.status(404).json(`Movie with id ${req.params.id} not found.`)
            }
        });
}
```

Figure 104: Movie_controller.js: Get all movies and get specific movie requests.

Identical get requests were added to the user controller too.

```
const loginRequired = (req, res, next) => {
    if(req.user) {
        next()
    }
    else {
        return res.status(401).json({
            message: 'Unauthorised user.'
        })
    };
}
```

Figure 105: User_controller.js: A function that requires log in when called was added. Just add the function to any routes that a user should be logged in to see to protect the routes from guests.

```
const editUser = (req, res) => {
    let userData = req.body

    User.findByIdAndUpdate(req.params.id, userData, {
        new: true
    })
        .then((data) => {
            if(data){
                res.status(201).json(data)
            }
        })
        .catch((err) => {
            if(err.name === "ValidationError"){
                res.status(422).json(err)
            }
            else {
                console.error(err)
                res.status(500).json(err)
            }

        });
}

const deleteUser = (req, res) => {

    User.findByIdAndRemove(req.params.id)
        .then((data) => {
            if (data) {
                res.status(200).json(`User deleted`)
            } else {
                res.status(404).json(`User of id ${req.params.id} not found.`)
            }
        });
}
```

Figure 106: User_controller.js: Added standard edit and delete functionality to the users collection.

```
const listSchema = new Schema({
    userID: {
        type: Schema.Types.ObjectId,
        ref: "User",
        trim: true,
        required: [true, 'User ID is required.']
    },
    listName: {
        type: String,
        trim: true,
        required: [true, 'List type is required.']
    }
}, {
    timestamps: true
});
```

Figure 107: List_schema.js: A schema was created for a user created lists collection, sotring only the userID as an ObjectID for relation between collections and the list name. A list controller was also created with standard get and post requests.

```
const listContentSchema = new Schema({
    listID: {
        type: Schema.Types.ObjectId,
        ref: "Lists",
        trim: true,
        required: [true, 'List ID is required.']
    },
    movieID: {
        type: Schema.Types.ObjectId,
        ref: "Movies",
        trim: true,
        required: [true, 'Movie ID is required.']
    }
}, {
    timestamps: true
});
```

Figure 108: ListContent_schema.js: A schema was created for list content; it only takes a listID and a movieID as ObjectIDs. Any item from any list is stored here.

```
const getAllListContent = (req, res) => {
    ListContent.find()
    .populate('movieID')
    .populate('listID')
```

Figure 109: listContent_controller.js: Standard get and add requests were added, but .populate was added to the get requests to populate the ObjectIDs with their objects information.

```
{
  "_id": "6425b76634650488dc5ee60c",
  "listID": "6425b055820c1b34772e90b6",
  "movieID": "64247be1edc29d0dee0ae91c",
  "createdAt": "2023-03-30T16:23:02.828Z",
  "updatedAt": "2023-03-30T16:23:02.828Z",
  "__v": 0
}
```


Figure 110: Insomnia: Return of get request for list content without .populate.


```
{
  "_id": "6425b76634650488dc5ee60c",
  "listID": {
    "_id": "6425b055820c1b34772e90b6",
    "userID": "641c87d06a97e629837fc079",
    "listName": "Favourites",
    "createdAt": "2023-03-30T15:52:53.977Z",
    "updatedAt": "2023-03-30T15:52:53.977Z",
    "__v": 0
  },
  "movieID": {
    "_id": "64247be1edc29d0dee0ae91c",
    "tconst": "tt0034583",
    "titleType": "movie",
    "primaryTitle": "Casablanca",
    "startYear": 1942,
    "runtimeMinutes": 102,
    "genres": "['Drama', 'Romance', 'War']",
    "averageRating": 8.5,
    "numVotes": 578015,
```


Figure 111: Insomnia: Return of get request for list content with .populate.

```
const userRatingSchema = new Schema({
    userID: {
        type: Schema.Types.ObjectId,
        ref: "User",
        trim: true,
        required: [true, 'User ID is required.']
    },
    movieID: {
        type: Schema.Types.ObjectId,
        ref: "Movies",
        trim: true,
        required: [true, 'Movie ID is required.']
    },
    rating: {
        type: Number,
        trim: true,
        required: [true, 'A rating is required.']
    }
}, {
    timestamps: true
});
```

Figure 112: UserRating_schema.js: A schema was created for user ratings, taking in just a userID, a movieID and the assigned rating. A userRating controller was created with standard get, post and put requests.

4.10.3 Item 2: Mapping the movies to cards in React

With the movies added to the MongoDB collection and the Express backend functionality set up, the movies can now be requested and received in React. The first thing to do with that capability is to map the movies to the card previously set up.

```
const [ movies, setMovies ] = useState(null);

useEffect(() => {
    axios.get('/movies')
        .then((response) => {
            console.log(response.data);
            setMovies(response.data);
        })
        .catch((err) => {
            console.error(err);
        });
}, []);
```

Figure 113: Index.js: Axios request to get all movies and store in a useState.

```
const moviesList = movies.map((movie) => {
    return  <Grid xs={6} md={4}>
                <MovieCard
                    key={movie._id}
                    movie={movie}
                    authenticated={props.authenticated}
                />
            </Grid>;
});
```

Figure 114: Index.js: Mapping the data received from the axios requests to cards.

4.10.4 Item 3: Adding movies to lists

In order for users to be able to add movies to their lists from the frontend, the system needs to be able to retrieve a user's list(s) without knowing any specific list IDs.

```
const getUserLists = (req, res) => {
    List.find({userID: req.params.userID})
        .populate('userID')
        .then((data) => {
            if (data) {
                res.status(200).json(data)
            } else {
                res.status(404).json(`No lists by ${req.params.userID} found.`)
            }
        });
}
```

Figure 115: List_controller.js: A get requests to the lists collection that finds all entries that match o user's ID.

```
const thisUserID = localStorage.getItem('userID');

useEffect(() => {
    axios.get(`/lists/user/${thisUserID}`)
        .then((response) => {
            // console.log("first call response = '
            setListID(response.data[0]._id);
        })
        .catch((err) => {
            console.error(err);
        });
}, [thisUserID]);
```

Figure 116: AddToListBtn.js: The user's ID, which is stored in local storage on log in, is taken from local storage and used in the get request that gets lists matching a user ID.

```
const onAdd = () => {
    axios.post('/ble', {
        listID: listID,
        movieID: props.id
    })
    .then((response) => {
        console.log(response.data);
    })
    .catch((err) => {
        console.error(err);
        console.log(err.response.data);
    });
};
```

Figure 117: AddToListBtn.js: When a user submits a movie to a list, onAdd is called and sends a post request to add the movie to the MongoDB collection.

4.10.5 Item 4: Rating in React

When a user rates a movie in this application, the rating must be sent and stored in the MongoDB collection and at the same time, sent to the Flask backend to be added to the dataset that the AI model is working with.

A star rating component from Material UI was used, movies can be rated out of 5 stars in increments of 0.5.

```
useEffect(() => {
    axios.get(`/userRatings/${thisUserID}/${props.movie._id}`)
        .then((response) => {
            setValue(response.data[0].rating);
            setID(response.data[0]._id)
            setCurrentRating(1)
        })
        .catch((err) => {
            console.error(err);
            console.log(err.response.data.message);
        });
}, [thisUserID, props.movie._id]);
```

Figure 118: MovieCard.js: A get request is sent to see if the current user has already rated a movie. If the request successfully returns with a rating, the movies stars will be displayed as blue instead of gold, so users can easily differentiate between movies they have and haven't already rated.

```
useEffect(() => {
    if(value && !currentRating){
        onRate();
        onSubmit();
    }
    else if (value && currentRating) {
        onEdit();
    }
}, [value, currentRating]);
```

Figure 119: MovieCard.js: An if else statement that's supposed to manage whether a rating is being added or updated. If there is no current rating, onRate and onSubmit are ran to add the new rating to both the MongoDB collection and Flask backend, if there is currently a user rating, onEdit should run and update the rating that was already there. It doesn't do that though, if a user has already rated a movie and tried to edit their rating, the old row is updated, but new rows with the new rating are also added, creating duplicate entries.

```
const onRate = () => {
    console.log("onRate value = " + value)

    axios.post('/userRatings', {
        userID: thisUserID,
        movieID: props.movie._id,
        rating: value
    })
```

Figure 120: MovieCard.js: When the onRate function is called, the rating for the specified movie is sent to the Express server and on to the MongoDB collection.

```
const onSubmit = () => {
    console.log("onSubmit value = " + value)

    const formData = new FormData();
    formData.append("tconst", props.movie.tconst)
    formData.append("this_user_id", thisUserID)
    formData.append("primaryTitle", props.movie.primaryTitle)
    formData.append("rating", value * 2)

    fetch('http://localhost:5000/rate', {
        method: 'POST',
        body: formData
    });
```

Figure 121: MovieCard.js: When the onSubmit function is called, the rating for the specified movie is sent to the Flask backend.

```python
@app.route('/rate', methods=['POST'])
def rate():

    refined_dataset = pd.read_csv('IMDbRefined2.csv')

    refined_dataset = refined_dataset.drop('Unnamed: 0', axis=1)

    print("request.form[tconst] = ", request.form["tconst"])
    print("request.form[this_user_id] = ", request.form["this_user_id"])
    print("request.form[primaryTitle] = ", request.form["primaryTitle"])
    print("int(request.form[rating]) = ", int(request.form["rating"]))

    newTitleID = request.form["tconst"]
    newUserID = request.form["this_user_id"]
    newPrimaryTitle = request.form["primaryTitle"]
    newRating = int(request.form["rating"])
    row = [newTitleID, newUserID, newPrimaryTitle, newRating]

    refined_dataset.loc[len(refined_dataset)] = row

    refined_dataset.to_csv('IMDbRefined2.csv')
```

Figure 122: App.py: The rating route.

To update the dataset with new user ratings, a new route had to be made in the Flask backend, as shown in figure 122. This new route reads in the dataset, receives form data, creates a new row from that form data, adds the row to the dataset before finally exporting the updated dataset to save over the old dataset, which can immediately be used to generate updated recommendations.

4.10.6 Item 5: Individual movie Page

Individual movie pages need to have all the information a user could want from a movie, as the dataset being used for the AI model and MongoDB collection have already been set up to use or store the IMDb ID, the quickest and easiest way to get extensive information on a specific movie is to make a request to the IMDb API with the specific movies IMDb ID. The only problem with this is that IMDb only allows 100 free requests per day, and as API requests are sent twice in the React development environment, this limits the application to only be able to request information on 50 movies per day.

```
useEffect(() => {
    axios.get(`/movies/${id}`, {
        headers: {
            "Authorization": `Bearer ${token}`
        }
    })
    .then((response) => {
        console.log(response.data);
        setMovie(response.data);
    })
    .catch((err) => {
        console.error(err);
        console.log(err.response.data.message);
    });
}, [token, id]);

useEffect(() => {
    if (movie) {
        fetch(`https://imdb-api.com/en/API/Title/k_zvyi75fh/${movie.tconst}`, {
            method: 'GET',
        })
        .then(res => res.json())
        .then(data => {
            console.log("17" + data);
            setIMDb(data);
        });
    }
}, [movie]);
```

Figure 123: Show.js: A get request is made for the specific movie, then when the request returns and fills the movie state with data, a request to the IMDb API is sent with that movies IMDb ID (tconst) and receives extensive information on the specified movie.

```
<h1>IMDb ID</h1>
{ IMDb?.id }
<h1>Title</h1>
{ IMDb?.title }
<h1>Poster</h1>
<img src={ IMDb?.image } alt="poster" width="343" height="508"></img>
<h1>Release Date</h1>
{ IMDb?.releaseDate }
<h1>Runtime</h1>
{ IMDb?.runtimeStr }
<h1>Plot Summary</h1>
{ IMDb?.plot }
<h1>Awards</h1>
{ IMDb?.awards }
<h1>Director(s)</h1>
{ IMDb?.directors }
<h1>Writer(s)</h1>
{ IMDb?.writers }
<h1>Main Cast</h1>
{ IMDb?.stars }
<h1>Production Companies</h1>
{ IMDb?.companies }
<h1>Language(s)</h1>
{ IMDb?.languages }
<h1>Age Rating</h1>
{ IMDb?.contentRating }
<h1>IMDb Rating</h1>
{ IMDb?.imDbRating }
<h1>IMDb Number of Ratings</h1>
{ IMDb?.imDbRatingVotes }
<h1>Box Office Gross</h1>
{ IMDb?.boxOffice?.cumulativeWorldwideGross }
<h1>Similar Movies</h1>
{ similarsList }
```

Figure 124: Show.js: The data retrieved from IMDb's API can now be used and displayed.

4.11 Sprint 7

4.11.1 Goals

- Finish developing all functionality of the final application
- Start the implementation section of the thesis
- Start the testing section of the thesis

4.11.2 Item 1: React search bar

Users need to be able to search for specific movies in the system so that they can rate movies they've already seen, or just see more information on a specific movie.

```
function SearchBar({ onSearch }) {
    const onSub = (e) => {
        onSearch(e.target.value.toLowerCase());
    };

    return (
        <form>
            <div>
                <input
                    onChange={onSub}
                    className="search-input"
                    placeholder="Search"
                    name="search"
                />
            </div>
        </form>
    );
}
```

Figure 125: Index.js: The search bar, onSub function is called on any input change and runs the onSearch event with the currently entered search term.

```
const [ filter, setFilter ] = React.useState("");
```

Figure 126: Index.js: A useState is setup to store the entered search term.

```
<SearchBar onSearch={(searchTerm) => setFilter(searchTerm)}/>
```

Figure 127: Index.js: The search bar is called and passes entered search terms to the search bar function and the filter useState.

```
const filteredData = React.useMemo(() => {
    if (filter === "") return movies;
    return movies.filter((item) =>
        item.primaryTitle.toLowerCase().includes(filter)
    );
}, []);

const moviesList = filteredData.map((movie) => {
    return  <Grid xs={6} md={4}>
                <MovieCard
                    key={movie._id}
                    movie={movie}
                    authenticated={props.authenticated}
                />
            </Grid>;
});
```

Figure 128: Index.js: A function is set up to pass an array of movies through, returning the array unchanged if the filter useState is empty, but if the filter is not empty it returns the array after filtering out movies who's title's don't match the filter term. UseMemo is used to store a memoised value to optimise performance when dealing with large arrays of movies.

This set up resulted in a broken search bar. If a user typed in the search bar, it would take the first letter they typed and start filtering results, as intended, but it would not save the character typed or even display it in the search bar, if a second character is entered, instead of being added on to the current search term, it overrides it. It also took focus away from the search bar. For example, if a user wanted to search for 'Casablanca' and they start by entering 'C' in the search bar, it will start filtering the movies and only movies with 'C' in their title will be displayed, to continue the search, the user must re-click on the search bar to enter 'a', which should now filter the movies by 'Ca', but the 'C' was overridden and it's actually just filtering by 'a'.

```
function SearchBar({ onSearch }) {
        let testing = "";

        const onSub = (e) => {
            onSearch(e.target.value.toLowerCase());
            testing = testing + e.target.value;
            console.log("testing = " + testing)
        };

        return (
            <form>
                <div>
                    <input
                        onChange={onSub}
                        className="search-input"
                        placeholder="Search"
                        name="search"
                        value={testing}
```

Figure 129: Index.js: An attempted solution for the search bar.

One of the attempted solutions, depicted in figure 129 above, had a variable set up in the search bar function to store the entered text as a string, then concatenate newly entered text with the already saved string and set the value of the input to the variable. This did not work at all, not only did the string not concatenate and still just overrode, but it also still didn't even display any entered terms in the search bar.

```
const onSub = (e) => {
    onSearch(e.target.value.toLowerCase());
    setTyped(e.target.value);
};

return (
    <form>
        <div>
            <input
                onChange={onSub}
                className="search-input"
                placeholder="Search"
                name="search"
                value={typed}
```

Figure 130: Index.js: Partial solution for the search bar problems.

The solution to two of the three problems turned out to be rather simple. As shown in figure 130 above, another useState was set up just to handle storing and displaying the entered search term, and setting the value of the input as the useState. This solved the problem of the search term not displaying in the search box, as well as the problem of each character entered overriding the one before it.

```
function SearchBar({ onSearch }) {
    const inputReference = useRef(null);

    const onSub = (e) => {
        onSearch(e.target.value.toLowerCase());
        setTyped(e.target.value);
    };

    useEffect(() => {
        inputReference.current.focus();
    }, [typed])

    return (
        <form>
            <div>
                <input
                    onChange={onSub}
                    ref={inputReference}
                    className="search-input"
                    placeholder="Search"
                    name="search"
                    value={typed}
                />
```

Figure 131: Index.js: Fallback solution to search bar losing focus on page render.

In order to resolve the issue of the client losing focus on the search bar on every character entered, causing users to have to click in the search bar after every character added to continue adding characters, a fallback solution was necessary. As can be seen in above in figure 131, it's been set so that on every render of the page, the focus is forced on to the search bar. So as a character is entered and the page re-rendered, the focus is immediately forced back on to the search bar where they can continue typing their search term without ever noticing.

4.11.3 Item 2: Filters

Options for filtering results based on user preferences.

```
const [ ratingFilter, setRatingFilter ] = useState(null);
const ratings = [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]

const [ genreFilter, setGenreFilter ] = useState(null);
const genres = ["Action", "Adult", "Adventure", "Animation", "Biography", "Come
```

Figure 132: Index.js: Arrays holding filter options and useStates to store selected options are set up.

```
const ratingFilterChange = (e) => {
    setRatingFilter(e.target.value);
    console.log("rating = " + ratingFilter)
};

const genreFilterChange = (e) => {
    setGenreFilter(e.target.value);
    console.log("genre = " + genreFilter)
};
```

Figure 133: Index.js: Functions are set up to wait for filters to be selected and then sets the states to their selected values.

```
const ratingFiltered = React.useMemo(() => {
    if (ratingFilter === null) return filteredData;
    return filteredData.filter((item) =>
        item.averageRating >= ratingFilter*2
    );
}, [filteredData]);

const genreFiltered = React.useMemo(() => {
    if (genreFilter === null) return ratingFiltered;
    console.log("GenreTest = " + ratingFiltered[0].genreFilter)
    console.log("GenreFilter = " + ['genreFilter'])
    return ratingFiltered.filter((item) =>
        item[genreFilter] === 1
    );
}, [ratingFiltered]);
```

Figure 134: Index.js: Rating and genre filter functions.

The functions shown above in figure 134 are set up similarly to the search bar filtering function, creating a chain of filter functions that takes the array of movies, passing it through one filter, either filtering or not filtering the array depending on whether a filter has been selected, before sending the result to the next filter to repeat the process.

4.11.4 Item 3: Pagination

Some kind of pagination functionality needs to be added so that the page containing all the movies can load faster and be easier to use. A paginator component was imported from Material UI to do this.

```
const [ perPage, setPerPage ] = useState(6);
const perPageOptions = [6,9,12,18,24,48]
```

Figure 135: Index.js: An array for the option of selecting how many items are shown per page is created, as well as a useState to hold the currently selected per page number, set to 6 by default.

```
function paginator(items, current_page, per_page_items) {
    let page = current_page || 1,
    per_page = per_page_items,
    offset = (page - 1) * per_page,
    paginatedItems = items.slice(offset).slice(0, per_page_items),
    total_pages = Math.ceil(items.length / per_page);
    console.log("item length: " + items.lgenth);

    return {
        page: page,
        per_page: per_page,
        pre_page: page - 1 ? page - 1 : null,
        next_page: total_pages > page ? page + 1 : null,
        total: items.length,
        total_pages: total_pages,
        data: paginatedItems
    };
}
```

Figure 136: Index.js: The paginator function takes in three values, the data, the currently selected page and how much of the data to show per page.

```
const count = Math.ceil(genreFiltered.length / perPage);
const [page, setPage] = React.useState(1);
const handleChange = (event, value) => {
    setPage(paginator(genreFiltered, value, perPage).page);
};
```

Figure 137: Index.js: The maximum page number is independently stored for future use, a new useState is declared to hold the current page number and an event listener added to update the useState with the currently selected page.

```
const moviesList = paginator(genreFiltered, page, perPage).data.map((movie) => {
    return  <Grid xs={6} md={4}>
                <MovieCard
                    key={movie._id}
                    movie={movie}
                    authenticated={props.authenticated}
                />
            </Grid>;
});
```

Figure 138: Index.js: Mapping the movies to cards.

After all the movies have been passed through all the filters, it's finally passed through the paginator before being mapped to the cards, as shown in figure 138 above. 'genreFiltered' contains the movie data after going through the genre filter, 'page' contains the currently selected page number and 'perPage' contains the currently selected number of items to see per page. For example, if

'genreFiltered' contains 200 movies, 'page' is set to 7 and 'perPage' is set to 12, only movies #73-84 will be mapped and displayed.

```
const reset = () => {
    setRatingFilter(null)
    setGenreFilter(null)
    setPerPage(6)
};
```

Figure 139: Index.js: An option to reset all filters to their default values was also added.

4.11.5 Item 4: Recommendations page

With all the pieces in place, the home page needs to be able to show a user their movie recommendations as soon as they enter the website.

First, functionality was added in the Express backend to allow getting a movie with a specific tconst (IMDb ID), then in React, a standard post request is sent to the Flask backend, sending the target user to generate recommendations for, the amount of similar users to consider and the number of recommended movies to return with.

```
useEffect(() => {
    let rec = [];
    if (prediction) {
        for (let i = 0; i < prediction.data.Recommendations[0].length; i++) {
            axios.get(`/movies/rec/${prediction.data.Recommendations[0][i]}`, {
                headers: {
                    "Authorization": `Bearer ${token}`
                }
            })
            .then((response) => {
                console.log(response.data);
                rec.push(response.data)
            })
            .then(() => {
                setPrediction2(rec)
                console.log("pred2 = " + rec)
            })
            .catch((err) => {
                console.error(err);
                console.log(err.response.data.message);
            });
        }
```

Figure 140: Home.js: Once the Flask backend has returned with the recommendations, get requests to the Express backend are looped through each of the recommendations and pushed to an empty array, then a useState is set to that array, as the array gets updated with a new item on each loop, so does the useState.

```
const moviesList = prediction2?.map((Recommendation) => {
    console.log("Checkpoint4 " + Recommendation[0].tconst);
    console.log(prediction2);
    return (
        <Grid xs={6} md={4}>
            <MovieCard
                key={Recommendation[0]._id}
                movie={Recommendation[0]}
                authenticated={props.authenticated}
            />
        </Grid>
```

Figure 141: Home.js: The movies returned from the Express backend are then mapped to cards.

This however does not fully work. On page load, 10% of the time it works as intended, all movies recommendations are displayed, but the other 90% of the time, all of the movie recommendations can be seen for less than a second, then all but one disappears, leaving only one recommendation. There are never any kind of errors or indicators of what's going wrong. All of the movies clearly are there, as they're always displayed for at least a few microseconds before disappearing. Many attempts to fix this were implemented, but with very limited success.

```
let moviesList = [];

if (prediction2?.length > no_of_movies) {
    moviesList = prediction2?.map((Recommendation) => {
        console.log("Checkpoint4 " + Recommendation[0].tconst);
        console.log(prediction2);
        return (
            <Grid xs={6} md={4}>
                <MovieCard
                    key={Recommendation[0]._id}
                    movie={Recommendation[0]}
                    authenticated={props.authenticated}
                />
            </Grid>
        );
    });
}
```

Figure 142: Home.js: Attempt to fix the movie cards displaying by adding a condition.

One attempted solution is shown above in figure 142, the movie mapping function in an if statement that required the calls to the Express backend to have finished making all the calls it's going to make before mapping the data. This unexpectedly made the problem worse, now no movies are ever displayed, even for a microsecond, but if the React application is saved, the page re-renders and shows all recommendations for less than a second before disappearing again.

```
useEffect(() => {
    if (prediction2?.length > no_of_movies) {
        setPrediction3(prediction2)
        console.log("prediction3 " + prediction3);
    }
}, [prediction2, no_of_movies, prediction3]);
```

Figure 143: Home.js: Another attempt to fix the movie cards display issue

Another attempted solution is shown in figure 143 above. An if statement again requires the calls to the Express backend to have finished, then it sets a new useState to the value of the Express filled useState. This new useState is then mapped to the cards. This partially fixed the problem, the page is empty on first load, but when it re-renders via the React application being saved, it displays all recommendations and they don't disappear. This problem has not been fully fixed as of sprint 7.

4.11.6 Item 5: List creation on registration

In order for new users to be able to instantly favourite movies, a list needed to be created and added to the user list collection whenever a new user registered, but it didn't work.

```
const register = (req, res) => {
    let newUser = new User(req.body)
    let testID = "";

    newUser.password = bcrypt.hashSync(req.body.password, 10)

    newUser.save((err, user) => {
        if (err) {
            return res.status(400).send({
                message: err
            })
        } else {
            user.password = undefined;
            testID = user._id;
            return res.json(user)
        }
    })

    let listData = {userID: testID, listName: "Favourites"}

    List.create(listData)
        .then((data) => {
            if (data) {
                res.status(201).json(data)
            }
        })
```

Figure 144: User_controller.js: Attempt to add to the list collection as a row is added to the user collection.

As attempted above in figure 144, an empty variable was declared to store newly created user's IDs, when a new user is created, the variable is set to their _id. Then a new variable is declared and given JSON formatted data to create a new list called 'Favourites' for every new user registering. The problem is that when a new user registered, it ran through these functions, starts adding the new user to the user collection, then starts trying to add a new list to the list collection, but it gives ObjectID errors, indicating that the new user has not been saved to the user collection before it tried to reference that user when creating a list for them.

4.11.7 Item 6: Similar movies

A second recommendation generation method to get movies similar to other movies to display a list of similar movies on each movies individual page.

```python
@app.route('/similar', methods=['POST'])
def similar():

    # Load CSV
    movie_titles_df = pd.read_csv('title_merge_ratings_500000.csv')

    # Drop unnecessary columns
    movie_titles_df = movie_titles_df.drop(['Unnamed: 0', 'titleType', 'startYear', 'run

    # Storing the variable taken from front end
    titleID = request.form["tconst"]

    # Using TfidfVectorizer to transform text to feature vectors
    tfidfvectorizer = TfidfVectorizer(analyzer='word', stop_words='english')

    tfidf_wm = tfidfvectorizer.fit_transform(movie_titles_df['genres'])

    # Using Cosine Similarity to calculate similarity between movies based on the genres
    cosine_sim = cosine_similarity(tfidf_wm, tfidf_wm)

    ## Building a 1-dimensional array from the movie titles
    titles = movie_titles_df['tconst']
    indices = pd.Series(movie_titles_df.index, index=movie_titles_df['tconst'])

    ## Function that retrieves the top recommendations based on the cosine similarity to
    def recommend_from(id):
        idx = indices[id]
        sim_scores = list(enumerate(cosine_sim[idx]))
        sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
        sim_scores = sim_scores[1:21]
        movie_indices = [i[0] for i in sim_scores]
        return titles.iloc[movie_indices]

    # Storing the generated recommendations
    recommendations = recommend_from(titleID).head(10)

    output = recommendations.to_json()

    return output
```

Figure 145: App.py: Get similar movies route.

As depicted in figure 145 above, a new route was added to the Flask backend. The same CSV the recommendations are normally taken from is read in and the unnecessary columns dropped. TfidfVectorizer is used to transform text to feature vectors and cosine similarity is used to calculate the similarity between movies based on the genres they have in common. The target movie is read in from the form data sent in the React post request, and a function returns movies similar to the target movie, which is then formatted as JSON and sent back to React.

```
useEffect(() => {
    // e.preventDefault();
    if (movie) {
        const formData = new FormData();
        formData.append("tconst", movie.tconst)

        fetch('http://localhost:5000/similar', {
            method: 'POST',
            body: formData
        })
        .then(res => res.json())
        .then(data => {
            console.log("checkpoint1 = " + data[Object.keys(data)[0]]);
            console.log(data);
            setRecs(data);
        });
    }
}, [movie]);
```

Figure 146: Show.js: Once the movie data is filled, a request to the Flask backend can be made to look for similar movies.

```
useEffect(() => {
    let recHolder = [];
    if (recs) {
        for (let i = 0; i < 5; i++) {
            console.log("recs checkpoint #" + i + " = " + recs[Object.keys(recs)[i]])
            axios.get(`/movies/rec/${recs[Object.keys(recs)[i]]}`, {
                headers: {
                    "Authorization": `Bearer ${token}`
                }
            })
            .then((response) => {
                console.log(response.data);
                recHolder.push(response.data)
            })
            .then(() => {
                setSimilars(recHolder)
                console.log("Checkpoint99 = " + similars)
            })
```

Figure 147: Show.js: After similar movies have been retrieved from the Flask backend, requests can be made to the Express backend for the similar movie's data to be mapped to cards.

```
const similarsList = similars.map((movie) => {
    return  <Grid xs={6} md={4}>
                <MovieCard
                    key={movie._id}
                    movie={movie}
                    authenticated={props.authenticated}
                />
            </Grid>;
```

Figure 148: Show.js: Mapping the similar movies to cards.

There is a problem with this though, it is only displaying one similar movie when it should be displaying 5. Console logs show that all 5 movies are being stored, but none of the attempted format changes fix this.

4.11.8 Item 7: Start implementation document

Started off by listing and describing the technologies used in the development of the application, as well as the development environment before describing the sprint methodology used over the course of the entire project. The goals of each sprint were defined and progress towards each goal logged, using a development journal kept and added to throughout the development process as reference.

4.11.9 Item 8: Start testing document

The testing chapter of the thesis contained two primary sections, one for functional testing and one for user testing. The functional testing section was tackled first, starting by listing the specifications of the system that the tests would be run on, before describing the tests that would be carried out and why they were to be carried out.

Multiple tests were carried out to determine system capabilities of working with various dataset sizes, prioritising speed and efficiency, and one test was carried out on the applications CRUD functionality.

4.12 Sprint 8

4.12.1 Goals

- Finish the implementation chapter of the thesis.
- Start and finish the project management chapter of the thesis.
- Finish any unfinished areas in the entire thesis.

4.12.2 Item 1: Finish implementation document

The implementation section for the last sprint, sprint 7, as well as this sprint were documented.

4.12.3 Item 2: Start and finish project management document

Each of the projects phases were briefly covered in the project management chapter of the thesis, going in to how each phase was approached and handled, before describing the sprint methodology used throughout the project and a few management tools used.

4.12.4 Item 3: Finish thesis

Mostly formatting updates, grammatical corrections. Added numbering to the multileveled headings, reformatted some image captions from implementation chapter as paragraphs, changed headings #3+ to just have capitalised first words rather than all words in the sentence, made headings #1 and #2 bold, made text size of comments in testing chapter smaller, made sure terminology was consistent (Like all usages of backend being ‘backend’ rather than some being ‘backend’, added page breaks before each heading #2 and proof read the thesis for any spelling errors.

4.13 Conclusion

In this chapter, the technologies used to develop the application were discussed, the application was developed primarily using the MERN stack for the web side and some Python libraries for the AI side. The development environment mostly involved the use of the code editors Visual Studio Code and Jupyter Notebook, and other important tools such as Git, GitHub, MongoDB and Insomnia.

A sprint methodology was utilised throughout this project and consisted of a total of 8 sprints, 6 before Easter and 2 after. The first 4 sprints were mostly focused on the report aspect of the project, while the last 4 sprints (and the unofficial Easter break sprint) were focused more on the development/coding aspects of the application.

5 Chapter Five: Testing

5.1 Introduction

This chapter will cover the testing that has been undertaken for this application, it involves performing tests on various aspects of the application to determine the most efficient way to handle data and verify that functionality is working as it should. All functional testing was carried out on one PC for consistency and will have its specifications described. Functional tests will be carried out on the filtering of some datasets, on the speed the AI model works with different dataset sizes and on CRUD functionality.

5.2 Functional Testing

All functional testing was carried out on a machine with the following specifications:

Processor: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz

Installed RAM: 16.0 GB

Operating System: Windows 10 19045.2846

System Type: 64-bit operating system, x64-based processor

The AI model(s) needed a refined dataset combined from individual movie titles and user ratings datasets to generate recommendations. The chosen movie titles dataset was 810 MB and contained 780,000 rows of movie title data, while the chosen user ratings dataset was 785 MB and contained 4,670,000 rows. These two datasets would need to be merged and transformed to a pivot table, which can then be given to the AI model, this avoids unnecessary time wasted manipulating data in the application.

Merging the two datasets without filtering them results in a dataset with 3,550,000 rows, attempting to pivot this table caused a Pandas overflow error, which is a known bug when working with large datasets. To deal with this, one or both of the original datasets must be reduced in size.

5.2.1 Movie title dataset size

To reduce the size of the movie title dataset, it was merged with a movie ratings dataset and then filtered by how many user ratings each movie had received. The merged dataset contained 327,000 rows of unique movies.

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Filtering out movies with less than 10 user ratings	Dataset with 327,000 rows	Dataset with 150,000 rows	Dataset with 290,000 rows	It was surprising how few rows were dropped
2	Filtering out movies with less than 100 user ratings	Dataset with 327,000 rows	Dataset with 200,000 rows	Dataset with 132,000 rows	Then it was surprising how many rows were dropped
3	Filtering out movies with less than 1,000 user ratings	Dataset with 327,000 rows	Dataset with 50,000 rows	Dataset with 42,000 rows	The pattern became somewhat predictable
4	Filtering out movies with less than 10,000 user ratings	Dataset with 327,000 rows	Dataset with 18,000 rows	Dataset with 10,500 rows	
5	Filtering out movies with less than 100,000 user ratings	Dataset with 327,000 rows	Dataset with 3,000 rows	Dataset with 2,250 rows	

6	Filtering out movies with less than 500,000 user ratings	Dataset with 327,000 rows	Dataset with 500 rows	Dataset with 293 rows	A smaller dataset than this should not be needed for any reason

5.2.2 User ratings dataset size

The user ratings dataset contained 4,670,000 rows from 1,500,000 unique users. The row count is more important to the eventual refined dataset size, while the unique user count is more important for the eventual pivot table and AI model. To reduce these numbers, the users with less than x amount of ratings are filtered out of the user ratings dataset.

Test No	**Description of test case**	**Input**	**Expected Output**	**Actual Output**	**Comment**
1	Filtering out users who have rated less than 5 movies	Dataset with 4,670,000 rows and 1,500,000 unique users	Dataset with 3,800,000 rows and 1,000,000 unique users	Dataset with 2,820,000 rows and 120,000 unique users	The amount of unique users dropped from just filtering out users with less than 5 ratings was drastically more than expected. Considering the row count didn't drop nearly as drastically, there must have been a lot of users with just a single rating.
2	Filtering out users who have rated less than 10 movies	Dataset with 4,670,000 rows and 1,500,000 unique users	Dataset with 2,000,000 rows and 80,000 unique users	Dataset with 2,370,000 rows and 49,500 unique users	With every additional user being dropped having at least 5 ratings/rows, the difference in dropped rows and dropped unique users is more consistent.
3	Filtering out users who have rated less than 15 movies	Dataset with 4,670,000 rows and 1,500,000 unique users	Dataset with 2,000,000 rows and 20,000 unique users	Dataset with 2,140,000 rows and 29,500 unique users	The remaining users are clearly having a lot of ratings each, as the filter gets narrower, the unique user count is almost halving while the row count loses about 10%.

4	Filtering out users who have rated less than 25 movies	Dataset with 4,670,000 rows and 1,500,000 unique users	Dataset with 1,800,000 rows and 10,000 unique users	Dataset with 1,880,000 rows and 15,600 unique users	
5	Filtering out users who have rated less than 50 movies	Dataset with 4,670,000 rows and 1,500,000 unique users	Dataset with 1,300,000 rows and 8,000 unique users	Dataset with 1,590,000 rows and 6,900 unique users	
6	Filtering out users who have rated less than 100 movies	Dataset with 4,670,000 rows and 1,500,000 unique users	Dataset with 1,200,000 rows and 2,500 unique users	Dataset with 1,335,000 rows and 3,200 unique users	A dataset smaller than this should not be needed for any reason. From the original dataset, 66% of the rows have been dropped, while 99.8% of the unique users have been dropped.

5.2.3 Merged dataset file size

All different combinations of filtered datasets were merged and exported to CSVs. This is the dataset that will be read and used by the Flask backend of the application, the smaller the file size the faster the dataset can be loaded and manipulated. The unique movie and user counts are an important aspect for the AI to generate accurate recommendations, but the entire process must be completed as fast as possible within the application for users to receive real time recommendations.

Test No	**Dataset of movies that have been rated more than _ times.**	**Dataset of users that have rated more than _ movies**	**Merged dataset unique movies**	**Merged dataset unique users**	**Merged dataset file size**	**Read merged dataset time**	**Total app run time**	**Comment**
1	10	5	130,000	115,600	107 MB	1.63s	N/A: Overflow crash on pivot table	
2	10	10	123,800	48,700	90 MB	1.47s	N/A: Overflow crash	

3	10	15	120,100	29,200	82 MB	1.16	N/A: Overflow crash	
4	10	25	115,400	15,500	72 MB	1.06	490.58s	Pivoting the table was using over 13 GB of memory
5	10	50	109,600	6,800	60 MB	0.89s	61.54s	A surprisingly dramatic reduction in time compared to previous test, but still far too long.
6	10	100	103,500	3,200	50 MB	0.74s	26.21s	
7	100	5	97,900	115,500	104 MB	1.55s	N/A: Overflow crash	The unique user count slightly drops compared to the previous test with the largest user ratings dataset because some of the unique users hadn't rated any of the movies in the smaller movie dataset.
8	100	10	94,600	48,700	88 MB	1.26s	N/A: Overflow crash	As datasets with users with more ratings are used, the number of users who haven't rated any movies in the smaller movie dataset drops closer to what it was with the larger movie dataset.
9	100	15	92,500	29,200	80 MB	1.15s	N/A: Overflow crash	
10	100	25	89,800	15,500	70 MB	0.96s	258.56s	
11	100	50	86,300	6,800	59 MB	0.8s	48.53s	

12	100	100	82,600	3,200	49 MB	0.67s	19.76s	
13	1,000	5	37,800	114,500	92 MB	1.31s	N/A: Overflow crash	
14	1,000	10	37,300	48,400	77 MB	1.06s	396.25s	First merged dataset with users with 10 (or 15) or more ratings that doesn't cause overflow crash
15	1,000	15	37,000	29,100	70 MB	0.96s	132.41s	
16	1,000	25	36,700	15,400	61 MB	0.82s	42.48s	
17	1,000	50	36,200	6,800	51 MB	0.69s	19.31s	Still roughly the same amount of unique users as with the movie dataset with the widest filter.
18	1,000	100	35,600	3,200	42 MB	0.58s	9.09s	As the datasets of more popular movies are being used, the more heavily filtered user datasets are effecting the unique movie count less and less.
19	10,000	5	9,875	109,600	70 MB	1.01s	137.04s	First merged dataset with users with 5 or more ratings that doesn't cause overflow crash
20	10,000	10	9,870	47,000	58 MB	0.8s	33.23s	
21	10,000	15	9,864	28,400	52 MB	0.7s	18.66s	
22	10,000	25	9,856	15,200	44 MB	0.62s	14.37s	
23	10,000	50	9,840	6,800	37 MB	0.52s	7.15s	
24	10,000	100	9,810	3,100	30 MB	0.39s	2.9s	
25	100,000	5	2,154	95,800	38 MB	0.55s	14.1s	
26	100,000	10	2,154	43,700	31 MB	0.46s	9.68s	
27	100,000	15	2,154	27,000	27 MB	0.4s	5.12s	
28	100,000	25	2,154	14,700	23 MB	0.35s	3.04s	

29	100,000	50	2,154	6,600	19 MB	0.28s	1.5s	
30	100,000	100	2,153	3,100	15 MB	0.23s	1.05s	At this stage the unique movie count is hardly effected, if at all, by merging with user ratings datasets.
31	500,000	5	287	59,700	11 MB	0.22s	1.39s	
32	500,000	10	287	31,200	9 MB	0.18s	0.9s	
33	500,000	15	287	21,200	8 MB	0.15s	0.79s	
34	500,000	25	287	12,400	6 MB	0.13s	0.59s	
35	500,000	50	287	5,900	5 MB	0.11s	0.5s	
36	500,000	100	287	2,900	4 MB	0.09s	0.4s	Only about 300 unique users are lost when merged with the smallest movie dataset compared to the largest.

5.2.4 CRUD

Create, read, update and delete requests need to be sent to the Express backend, the Flask backend, the IMDb API and the OMDb API

Test No	**Description of test case**	**Expected Output**	**Expected output achieved?**	**Comment**
1	Get all movies from Express.	An array of all movies	✓	
2	Get specific movie from Express.	A specific movie.	✓	
3	Get specific movie by IMDb ID from Express.	One movie specified by IMDb ID, ‘tconst’.	✓	
4	Post register data to Express.	Hash the password, send the data to the MongoDB collection and return with the successfully added row.	✓	
5	Post login data to Express.	Successful login, return with user ID and generated authentification token.	✓	

6	Get all users from Express.	An array of all users.	✓	
7	Get specific user from Express.	A specific user.	✓	
8	Edit specific user in Express.	Overwrite current entry in the collection.	✓	
9	Delete specific user in Express.	Delete entry in collection.	✓	
10	Get all user lists from Express.	An array of all user lists.	✓	
11	Get specific user list from Express.	A specific user's list.	✓	
12	Get all of a user's list by user ID from Express.	An array of all lists of a specific user.	✓	
13	Post new list data to Express.	List data added to MongoDB collection.	✓	
14	Get all list items from Express.	An array of all list items.	✓	
15	Get specific list item from Express.	A specific list item.	✓	
16	Get all list items in specific list by list ID in Express.	All items in a specific list.	✓	
17	Post new list item data to Express.	List item data added to MongoDB collection.	✓	
18	Get all user ratings from Express.	An array of all user ratings.	✓	
19	Get specific user rating from Express.	A specific user rating.	✓	
20	Get all ratings by a specific user in Express.	An array of all ratings by a specific user.	✓	
21	Get rating by specific user of specific movie by user ID and movie ID from Express.	A specific user rating.	✓	
22	Post new user rating data to Express.	Rating data added to MongoDB collection.	✓	
23	Edit specific rating in Express.	Overwrite current entry in the collection.	✓	
24	Post movie to Flask server for list of similar movies.	An array of similar movie's IMDb IDs.	✓	
25	Post rating data to Flask server so it can be added to the CSV file for the AI model.	Row successfully added to CSV.	✓	
26	Post user data to Flask server for a list of AI generated movie recommendations.	An array of 4 arrays; the target users highest rated movies, other users most similar to target user, those similar user's distances from target user and the IMDb IDs of generated recommendations.	✓	

27	Get request to IMDb API for specific movie information	Extensive information of a specific movie.	✓	100 free requests per day.
28	Get request to OMDb API for specific movie poster.	A specific movies poster.	✓	A subscription service, lowest tier used, received 150,000 requests per day.

5.2.5 Discussion of functional testing results

The purpose of filtering the movie title and user rating datasets was to create merged datasets of a variety of sizes in order to determine the largest dataset that could be passed through the AI model that would output movie recommendations quickly enough for a real time application. There are many important aspects to consider here, the speed and efficiency that the system runs at is essential, but row count as well as the number of unique users and movies in each dataset are hugely important for the model to be able to generate accurate recommendations, the bigger and more he better. So, it's essentially about finding the best balance between run speed and model accuracy.

In terms of speed, the AI model must be able to run through the data and output recommendations in absolutely no more than a couple of seconds, ideally in under a second. The biggest datasets that were tested couldn't even be passed through the model, the biggest one that could successfully be passed through the model had almost 1.8 billion data points and took 490 seconds to complete, which is far, far too long for the application's purposes. Of the 36 different datasets tested, the AI model could run through only 5 of them in less than 1 second, the largest of which still contained 8 million datapoints.

5.3 Conclusion

The primary limiting factor in most of these test scenarios is the capabilities of the PC running the tests. With, for all intents and purposes, standard PC specifications, the amount of data that tasks can be performed on is very limited in comparison to what a commercial applications capabilities would be. With the way this application has been set up, larger datasets can be seamlessly swapped in or out depending on the capabilities of the PC (or server) and still offer real time movie recommendations, though if a new dataset is being used, it must also be turned into a MongoDB collection.

If real time recommendations were not a priority, it would be possible to update a user's displayed recommendations at set times, once an hour or once a day even, allowing for datasets of more movies and more users to be used, as the time it takes for the processes to complete is less relevant. This would result in more accurate recommendations for users, but make the application less helpful for users looking for quick, fresh recommendations and new users for who the system would not have any information on to generate recommendations until after the next model update.

6 Chapter Six: Project Management

6.1 Introduction

This chapter describes how the project was managed and how well the group worked together as a team. It shows the phases of the project, going from the project idea through the requirements gathering, the specification for the project, the design, implementation and testing phases for the project. It also discusses Trello, GitHub and project member's journals as tools which assist in project management.

This chapter describes how the project was managed, it goes over the various phases of the project, going through the project proposal, the research, the requirements gathering, the design, the implementation and testing phases, describing the processes of each phase.

6.2 Project Phases

6.2.1 Proposal

Throughout the proposal phase of the project, various ideas were considered for an application. The only criteria for any ideas were that they had to be complex enough for the projects standards and that they had be of some interest to the developer. A movie recommendation system was one of the first ideas proposed, but was initially rejected for being too simple, but upon expansion of the idea it was accepted.

6.2.2 Research

Before starting the research phase of the project, a few primary sources of information were sought. Important details considered when seeking primary sources include how recently published they were, how many times they were cited, and the standing of authors in the field. Three published books were chosen as primary sources, these books were:

- ‘Practical Recommender Systems’ by Kim Falk.
- ‘Recommender Systems Handbook’ by Francesco Ricci et al.
- ‘Recommender Systems: Algorithms and Applications’ by P. Pavan Kumar et al.

A few secondary sources were used when more specific information was needed, such as when describing Amazon and YouTube’s recommendation systems.

6.2.3 Requirements

In the requirements phase of the project, various similar applications were researched and their qualities documented. There were many good options to choose from in this application area, eventually three were chosen, each were chosen because each of them prioritised different aspects of the target area. IMDb was chosen because of their extensive catalogue of movies, movie information and user ratings, MovieLens was chosen because of their prioritisation of personalised recommendations with many options and MyAnimeList was chosen for their community focused system.

Interviews and surveys were conducted to help in determining the direction of the application, but there were difficulties in finding willing participants, resulting in a fairly small sample size and limiting what could be discerned from them.

6.2.4 Design

The design phase of the project covered the research of the technologies being used and the structures and design patterns of the primary technologies, such as React and Express. Once the research was completed, the application architecture and database design could then be defined.

Also covered were user interface design properties, such as typography and colour scheme. There were many good typefaces to choose from, such as Lato or Inter, but Roboto was eventually decided

upon. The colour scheme was more difficult to decide, an online tool, Colormind, was used to help determine good potential colour schemes for the application, and one selected through this process.

During the creation of several diagrams using Creately, the account used to create the diagrams had it's access to Creately revoked for going over their free project limit (3 free projects). Not only did this revoke the privilege of creating new diagrams, but it also revoked the privilege of accessing the already created diagrams, they could not be seen, screenshotted, exported or accessed in any way. Deleting one of the three projects changed nothing. Creately was contacted to try and resolve this situation, but this was apparently their intended business practice, and nothing would happen so long as the account remained a free account. Figma was used for all diagram purposes after this.

6.2.5 Implementation

The implementation phase of the project was carried out over 8 sprints (and one unofficial sprint over the Easter break), each sprint covered a two-week period. Each sprint had predefined goals set up before the project started to act as a guide, but they were not so easy to keep up with, resulting in custom goals decided upon with the project supervisor before starting each sprint.

Writing about each sprint was significantly easier with the help of a development log that had been kept throughout the development process. This log mostly contained coding development, so it wasn't very helpful when discussing the first few sprints which mostly involved research and work on the report, but exceptionally helpful for later sprints that involved a lot of coding and app development.

6.2.6 Testing

The testing phase of the project covered functional testing of the application, which served to, among other things, determine the best dataset for the AI model to use, based on dataset size, how long it took the AI model to run through the dataset and the quality of output recommendations.

6.3 Sprint Methodology

Each two-week sprint had its own pre-defined goals from the beginning of the project, these pre-defined goals were loosely followed. Throughout the project, goals that were more contextually relevant to the current stage of the project were defined in meetings with the project supervisor and attempted in the proceeding sprint.

Some sprints progressed better than others, with goals being met or even exceeded, while other sprints were more challenging and all of their goals were not met. The first few sprints that entailed working on the application (Sprints 3-6.5) moved rather slowly and some goals were missed, which caused a ripple effect after missing the first sprints goals. The last couple of sprints (7 and 8) had loftier goals in order to make up for these setbacks, and these goals were met and exceeded.

6.4 Project Management Tools

6.4.1 GitHub

GitHub was used to back up the project in an online repository.

6.4.2 Journal

A personal log was kept during the development of the application, which kept track of what was being done. The log recorded what was being done, problems encountered with each thing and how solutions were attempted and implemented. The log also kept track of an ever-changing shortlist of things that had to be done.

7 Chapter Seven: Reflection and Conclusion

7.1 Reflection

7.1.1 Your views on the project

At the beginning, I found it quite difficult to settle on a project idea, then when I eventually had settled on an idea, I found it difficult to decide on a title for the thesis and a name for the application. I'm still not exactly happy with either thesis title or application name, but it is what it is. It was a good application area for me though, as I very much enjoy film and television and put significant value on ratings and recommendations. It was easier to work with something I had an interest in.

Overall, the project went fairly well, though there definitely were parts that went better than others, and some things that weren't done that I would have preferred if they were. Looking back on it, perhaps the initial proposal was a bit ambitious, for example, it would have been great if I could have implemented some kind of layered rating functionality for users, where they could rate aspects of movies such as rating a movie's music score or separately rating the same movie's acting quality, theoretically resulting in much more personalised recommendations. To do this though, I would have essentially had to develop my own AI system rather than using libraries, which while maybe possible, would have been a significant amount of additional work.

7.1.2 Completing a large software development project

Over the course of this project, I've learned a few things about how such a large-scale project can be tackled more easily. First and foremost is organisation, having clearly laid out plans with well-defined goals and tasks ripples throughout the entire project and makes almost everything easier. It's important to work as consistently as possible, to keep a good flow and the project moving forward, otherwise some things can build up and become much bigger problems than they ever should have been. If motivation becomes a problem, having the discipline to push through and do some work, even if it's just a little bit, is important in the rebuilding and maintaining of motivation and discipline, two of the most essential qualities when working on a big project.

7.1.3 Working with a supervisor

My supervisor for this project was John Montayne, we met about once a week to discuss how the last week had gone and what the next week would entail. I enjoyed working with John, I think we got on pretty well and I found him very easy to work with. John was always very direct when talking about the project, whether it was about how it was going well or about a problem, which is something I really appreciate. He made himself available for any problems I ran in to, offered a lot of great advice, helped me stay on track and give some motivation as it was needed, this project would not have gotten as far as it did without John's help.

7.1.4 Technical skills

Throughout the development of the application, some new technical skills that could be helpful in the future were learned, such as:

- How to integrate Python into a web application using frameworks like Flask.
- How to effectively use React hooks.
- How to create interactive, Hi-Fi prototypes.
- Various ways to manipulate and format data in Python.
- How to export and import AI models and updated datasets.
- How to convert data to CSV file format and create MongoDB collections from it.
- How to better make use of React components for cleaner code.

7.1.5 Further competencies and skills

Over the course of this project, many new competencies and skills were picked up and current competencies and skills were improved upon, such as:

- The important things to look for in research sources.
- Greater understanding of AI, machine learning and recommender systems.
- Greater understanding of recommendation methods, how they work and what they’re best suited for.
- Greater understanding of how some of the most successful commercial recommender systems work (YouTube, Amazon).
- More competence with working on large scale projects.

7.2 Conclusion

The project proposal was a movie recommendation system with ambitious additional functionality that aimed to give users the option to interact with a recommendation system in the ways that they wanted to, whether that was a simple recommendation system that took some user information and returned with recommendations, a more complex system that requested ratings for specific movie aspects before returning with recommendations, or even a community based system that focused on user to user relationships and more heavily based recommendations on them. By the end of the project, the final application could not meet all of these goals.

The application was developed primarily using the MERN (MongoDB, Express, React and Node) stack, with Flask used to handle any Python. Anaconda/Jupyter Notebook was used for any Python coding, libraries such as Pandas and NumPy were used to format and manipulate data for the AI model and database collections, and the scikit-learn library was used to make and train the AI model.

There's a lot of room that the project can still grow into. Additional recommendation options can be added, larger datasets can be used on more powerful machines for more accurate recommendations, many community features can be implemented such as user reviews, or direct messaging, and it could be expanded to recommend other mediums of entertainment, such as television, short films, or even the likes of books and video games.

8 References

Falk, Kim. (2019). Practical Recommender Systems. Manning Publications

Ricci, Francesco., Rokach, Lior., Shapira, Bracha. (2022). Recommender Systems Handbook (3rd edition). Springer.

Pavan Kumar, P., Vairachilai, S., Potluri, Sirisha., Nandan Mohanty, Sachi. (2021). Recommender Systems: Algorithms and Applications (1st edition). Taylor & Francis Group.

Rich, Elaine. Personal Blog, Earlier Work. The University of Texas at Austin.

Terry, Douglas., Oki, Brian M., Nichols, David., Goldberg, David. (1992). Using Collaborative Filtering to Weave and Information Tapestry (3rd edition).

Dong, Xhenhua., Wang, Xhe., Xu, Jun., Tang, Ruiming., Wen, Jirong. (2022). A Brief History of Recommender Systems. ByteDance, Tsinghua University.

Pu, Pearl., Chen, Li. (2010). A User-Centric Evaluation of Recommender Systems. CEUR-WS.org.

Knijnenburg, Bart P., Willemsen, Martijn C., Ganter, Zeno., Soncu, Hakan., Newell, Chris. (2012). Explaining the User Experience of Recommender Systems. Springer.

Hardesty, Larry. (2019). The History of Amazon's Recommendation Algorithm. Amazon.

Covington, Paul., Adams, Jay., Sargin, Emre. (2016) Deep Neural Networks for YouTube Recommendations. Google.

Lyons, Peter. (2016). Express Code Structure. GitHub.com.

Singh Gill, Navdeep. (2023). Understanding ReactJS Project Structure and Folder Setups. XenonStack.com.

React File Structure. ReactJS.org.

React Flux Concept. JavaTpoint.com.

Henderson, Luke. (2022). How to Manage a Sprint Cycle More Effectively. Niftypm.com.

MacKensie, Ian., Meyer, Chris., Noble, Steve. (2013) How retailers can keep up with consumers. McKinsey.com.