Seun Akanni

Registration number 100276393

2023

A Survey and Implementation of Collaborative Filtering Recommendation Systems

Supervised by Jason Lines

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

Recommendation systems are algorithms and models designed to suggest relevant items to users using information captured to characterise their preferences. This creation of a user profile can either be used independently to select suitable items or be used to find similar users to then branch into their own list of recommendations. This paper focuses on the personalised division of recommender systems which incorporates other users into the decision-making process of generating both a predicted rating for an item and a suggested list of items within the movie domain. To begin, we provide an overview of the fundamentals necessary to understand the field of recommender research before establishing a set of data we can use for our remaining experimentation. In addition to utilising the performance metrics required to evaluate the systems, we also discuss concepts overlapping both recommender research and machine learning before implementing and assessing the performance of our own algorithms and a learning-based technique derived from matrix factorisation titled singular value decomposition.

Acknowledgements

I would like to thank my supervisor Jason Lines who made researching and writing this report more manageable with his continuous guidance and advice. I also cannot neglect to express my appreciation for my family for their continued support throughout my university studies; without which I would not have been able to progress as far as I have.

Contents

1. Introduction

The quantity and diversity of information and products available online have exponentially increased as a result of the continuous growth of internet usage by both individuals and businesses. This overwhelming availability can lead to difficult circumstances when attempting to locate relevant items which is what recommender systems aim to alleviate. Recommendation systems are algorithms that suggest *items* to *users* that they may not have discovered otherwise; which can include videos, movies, or products to purchase. These algorithms incorporate machine learning solutions to offer suggestions based on a variety of factors including how a user responds to items deemed similar to that being recommended. The format of recommendations is most commonly regarded as *personalised*, which provides a set of recommendations tailored towards a particular user and can be divided into content-based filtering (CBF) and collaborative filtering (CF) methodologies. The alternatives are described as *non-personalised*, and recommend what is popular and relevant to all users regardless of their user profiles, for example, the most watched movie on a streaming site. According to Ricci et al. (2015), non-personalised systems are simpler to generate and have their use in certain scenarios but are not normally covered under recommendation system research. Ricci et al. (2015) goes on to describe recommender systems as 'an example of [the] large scale usage of machine learning and data mining algorithms in commercial practice' which has inspired research teams and commercial industries to further develop these algorithms. They can be found in a range of application domains[1] such as e-commerce, entertainment, and social media, however, before any system can be selected to use, the domain in which the system operates and overall goals must be thoroughly understood.

The main body of the report focuses on describing and experimenting with the fundamentals associated with recommender systems to provide insight into the ongoing research surrounding the research field. Our background section looks at existing literature and focuses on detailing the categories of recommender systems, an in-depth analysis of a model-based filtering technique known as matrix factorisation, and finally the metrics used to assess an algorithm's performance. Following this, the methodology of the report establishes and expands upon a basic memory-based CF algorithm before introducing the library used for our model-based research. Section 4 then uses our memory-based algorithm officially titled kNN to explore the dataset used to both

[1] Application domain refers to the area in which such systems are used e.g., e-commerce, e-business etc.

train and evaluate our chosen machine learning model.

2. Background

2.1. Case Study: LeShop

The value of recommendation systems can often be underestimated, but according to Amatriain and Basilico (2015), it was the significant business success of said systems that inspired the Netflix Prize challenge; a competition designed to find new innovative methods for enhancing Netflix's own recommender systems; further detailed in Section 2.4.2. In partnership with the e-commerce supermarket *LeShop*[2], Dias et al. (2008) assessed the value of recommender systems by implementing an in-store and checkout recommender on the site. Launched in October 2007, the in-store recommender displayed recommendations at the bottom of each category of the site for shoppers while the checkout recommender launched in May of the previous year provided recommendations during the checkout process. The outcome of their investigation found a positive impact on the business's revenue as a result of the recommendation systems. This included a rise in their direct extra revenue from items suggested by the system, as well as indirect revenue from repeated purchases and purchases influenced by other systems. According to Dias et al., LeShop was able to generate an additional total revenue amount of 0.48%; of which 0.19% was as a result of the indirect extra revenue and 0.29% being the total direct extra revenue generated. This notable increase is emphasised to have been a direct outcome of the recommender systems introducing customers to new categories of the website.

2.2. Machine Learning

Machine learning (ML) is a field within artificial intelligence (AI) that is used to simulate human learning by using algorithms (models) to make predictions on previously unseen data. Recommendation systems make use of said algorithms, but due to the massive quantity of options available, difficulties can arise in the selection of suitable algorithms for a given application domain. The categories of ML algorithms take many forms, however, there are conventionally three distinct classifications which they are

[2] *https://www.migros.ch/en*

grouped into based on their learning process; supervised, unsupervised, and reinforcement learning. Supervised learning learns a function to map an input to an output via the use of labelled data (Mahesh, 2020) which references data accompanied by a label (identifying information). The data is split into a training set which updates the internal parameters, therefore, teaching the algorithm a pattern before a prediction is made on the test set. The results of the prediction are then used to evaluate the algorithm's performance where adjustments can then be made to improve the model's predictive capabilities. As Mahesh (2020) explains, unsupervised learning differs from supervised learning in that there are no labels associated with the data; requiring the model to find meaning in data where there are 'no correct answers and there is no teacher.' The final classification, reinforcement learning, involves a program known as an agent that performs actions within an environment as a means of learning the optimal behaviour which will maximise its reward.

To evaluate the performance of an ML model using supervised learning, we compute the loss function for each prediction by comparing the model's output to the actual labels. The loss value for each prediction can then be summarised across all labels to produce a single loss value, indicating the algorithm's overall performance. Once the algorithm's performance has been determined, we aim to minimise the loss function by optimising the algorithm. This is achieved through a process known as *gradient descent*[3] which is what essentially fits (trains) an ML model by iteratively updating a set of internal model parameters. A developer has no control over these internal parameters but can adjust certain variables known as *hyperparameters* used to control the learning process and determine the overall complexity of the model (i.e., how a model is fitted to a dataset). Two of the more noteworthy hyperparameters we will focus on include *learning rate* and *regularisation rate* which will be covered later in the report.

2.3. Content-Based Filtering

The first division of recommendation systems, content-based techniques (Figure 1), are described as domain-dependant and most successful for recommending items such as web pages, publications, and news (Isinkaye et al., 2015). The nature of content-based filtering (CBF) involves learning a user profile based on features belonging to items a single user has rated. ML models are then used to analyse an item's features and

[3] *https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent*

compare them with those the user has previously interacted with, thus generating a set of recommendations. The type of user profile developed is based upon the 'learning method employed' and such models include vector-based representations (Burke, 2002) or probabilistic models such as Naive Bayes classifier, decision trees and neural networks (Isinkaye et al., 2015).

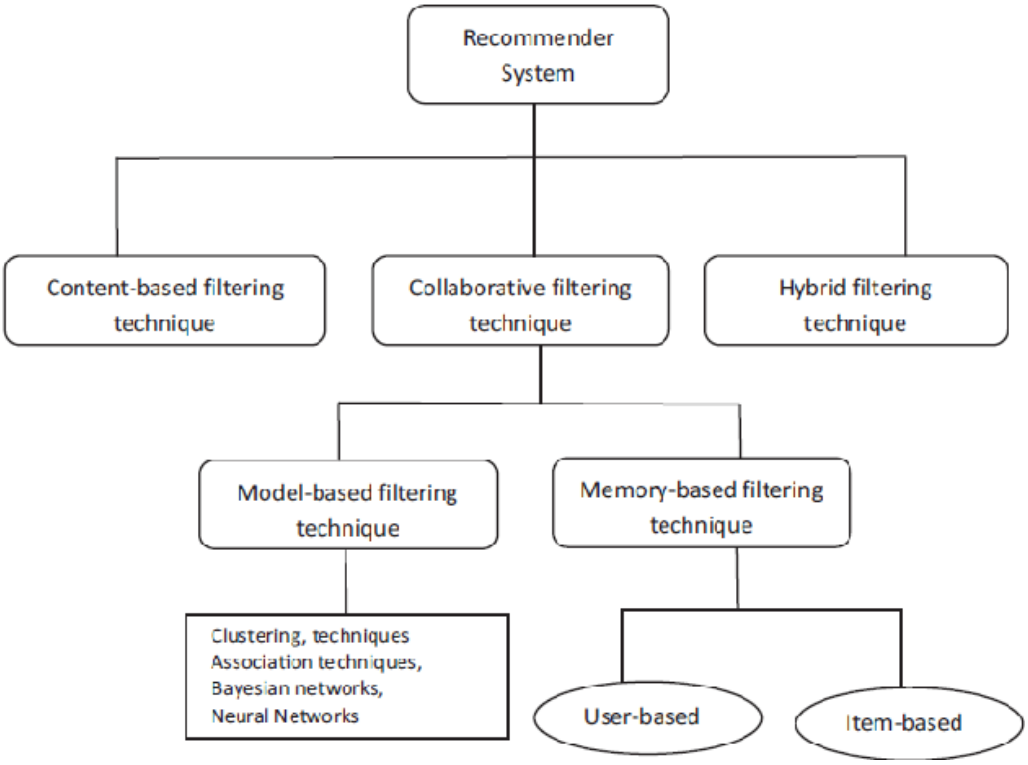


Figure 1: The three main classifications of recommendation systems (Zahrawi and Mohammad, 2021).

The filtering technique is dependent on items' metadata and user profiles which both must be highly descriptive to accommodate users' preferences. This has the added benefit that over time as users' likes and dislikes change, CBF methods can quickly accommodate these changes whilst items with no ratings are not excluded from being recommended (Isinkaye et al., 2015). Nevertheless, Ning et al. (2015) note that problems can arise with such a system that's effectiveness is dependent on the availability of descriptive data such as *limited content analysis* and *overspecialisation*. Limited content analysis arises from a lack of information regarding a user or item, preventing items from being effectively chosen. Overspecialisation leads to a lack of variation in

recommendations brought upon by items being "too" similar to those associated with a user.

2.4. Collaborative Filtering

Collaborative filtering (CF) is considered to be the more popular division of personalised systems with both Burke (2002) and Sarwar et al. (2001) stating that CF algorithms are the most recognised and widely implemented recommendation technique. As stated in the name, what differentiates CF from CBF is that the methodology involves considerations for other users, as opposed to only focusing on a single user profile. A typical CF scenario consists of a corpus of users $U = \{u_1, u_2, u_3, \ldots, u_m\}$ and items $I = \{i_1, i_2, i_3, \ldots, i_n\}$ in which users have expressed a rating or opinion on several items within the corpus. This can also be represented by a feedback (or ratings) matrix which is a table consisting of feature values obtained either explicitly as rating scores (e.g., likes and dislikes denoted by a value) or implicitly by analysing user engagement time.

One of the most notable advantages of collaborative recommendations is the ability for it to make *serendipitous* recommendations; an important factor for users to appreciate the functionality of recommendation systems (Ning et al., 2015). Serendipity is the ability of a system to recommend interesting items that a user would not have discovered otherwise and is attributed to the use of other users' preferences in CF. An example would be a movie being recommended that is different to an active user's taste or is not well known, for example, a movie with a director the user has not previously encountered. Where CF methods suffer is in relation to the density of the data which is often *sparse*, as users typically only provide feedback on a small proportion of items which may only cover 1% of products (Sarwar et al., 2000). Sparse data is then also a cause of another significant issue known as *limited coverage* which is associated with the similarity between users. A low percentage of user ratings reduces the overall effectiveness of the similarity measurements which can limit which items are to be recommended to the active user.

2.4.1. Memory-Based Filtering

Neighbourhood-based or memory-based systems use statistical methods to create a neighbourhood of users with 'similar rating patterns' (Ning et al., 2015) and once created, employ the use of various algorithms to generate predicted ratings or recommen-

dations for an active user[4]. The article by Sarwar et al. (2001) refers to memory-based algorithms as primarily *user-based* which is only concerned with forming a neighbourhood of similar users, however, memory-based algorithms can also be *item-based* which determine the similarity between items as opposed to just users based on how similar their given rating values are. While both user-based and item-based filtering makes use of a similarity measure, there can be differences in the performance of a recommender system depending on the methodology chosen (Ning et al., 2015).

2.4.2. Matrix Factorisation

Do et al. (2010) states that the main issue associated with memory-based filtering is the requirement of 'loading a large amount of in-line memory'. As the number of users and items in a system increases, the ratings matrix representing the corpus of users and items also increases which over time can degrade the system's performance. Model-based approaches aim to combat the issue by learning a predictive model (i.e., ML model/algorithm) that uses its parameters to predict new user ratings and such examples include 'Clustering models, Bayesian Networks, latent semantic technique and Dimensionality Reduction technique' (Zahrawi and Mohammad, 2021).

Matrix Factorisation (MF) is an example of model-based filtering classified as a dimensionality reduction technique often regarded as one of the most recognised model-based filtering methods in recommender system research and literature. It became popularised due to its effectiveness in the Netflix prize challenge, 2006; a competition designed to challenge 'the data mining, machine learning and computer science communities to develop systems that could beat the accuracy of Cinematch' (Bennett et al., 2007); the algorithm Netflix originally had in-place to recommend movies to subscribers. According to Bennett et al. (2007), the dataset provided to competitors was comprised of 100 million ratings on a scale of 1 to 5 stars which aimed to reflect the distribution of all ratings Netflix received during the period between 1998 and 2005. The grand prize would be awarded to the team whose system could beat the accuracy performance of Cinematch by an additional 10%.

Koren et al. (2009) notes the main challenge of MF models involves mapping users and items to a 'joint latent factor space of dimensionality f' depicted by Figure 2 which associates users and items as vector quantities $q_i \in \mathbb{R}^f$, $p_u \in \mathbb{R}^f$. The estimation of a

[4]The active user refers to the user within our corpus we are generating a set of recommendations for.

users u rating $\hat{r}_{ui}$ is then denoted by:

$$\hat{r}_{ui} = q_i^T p_u \tag{1}$$

which finds the dot product between the transpose item vector and the user vector. For identifying 'latent semantic factors', Koren et al. (2009) describes *singular value decomposition* as a useful technique (particularly in information retrieval) that requires 'factoring the user-item ratings matrix'. Confusingly, singular value decomposition (SVD) is a separate technique to the matrix factorisation algorithm used in the Netflix competition also called *SVD* which was named after the 'common matrix decomposition technique singular value decomposition' (Strömqvist, 2018). As mentioned in Section 2.4, the user-item ratings matrix often lacks sufficient ratings and so earlier SVD systems are stated to have used imputation[5] as a means of combating the problem of sparsity. This solution was not feasible according to Strömqvist (2018) who attributed this to the increase in information, computation expense, and an arise in bias generated from the added ratings. For MF solutions, this has evolved into fitting a model directly on the observed ratings while a regularisation term λ is used to simplify the ML model and avoid overfitting[6].

To 'learn the factor vectors (p_u and q_i)', Koren et al. (2009) states the system must minimise the 'regularised squared error':

$$\min_{q*,p*} \sum (r_{ui} - q_i^T p_u)^2 + \lambda(||q_i||^2 + ||p_u||^2) \tag{2}$$

through two common ML methods; *alternating least squares* (ALS) and *stochastic gradient descent*[7] which was popularised by Brandyn Webb (alias "Simon Funk") during the Netflix competition.

One of the main benefits attributed to MF within the CF domain is the concept of *Scalability* which refers to an increase in data as more users and items are added. The learning method employed by the MF models is 'generally more efficient than the batch method as it incorporates updates immediately instead of waiting to process the cumula-

[5] Substitution of values as a means of filling the ratings matrix.

[6] 'Creating a model that matches the training data so closely that the model fails to make correct predictions on new data' (Google Developers, nd).

[7] Follows the same principle as the gradient descent algorithm except it uses a single data point to update the parameters i.e., reduce the models loss computation.

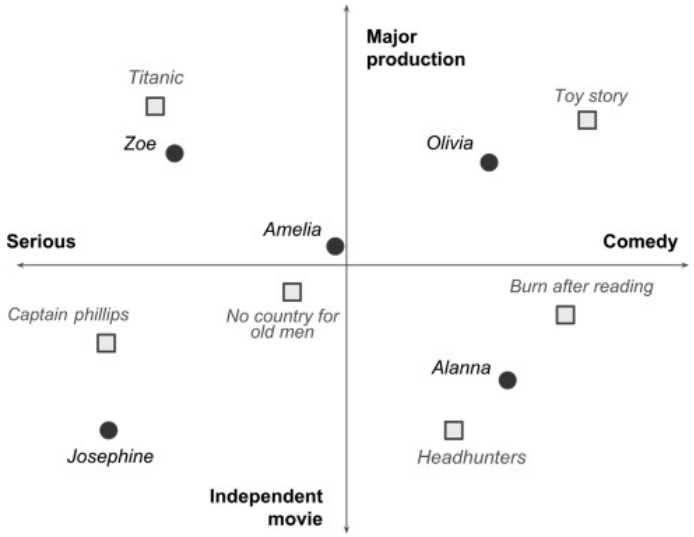


Figure 2: A two-dimensional latent factor space comprised of both user and item vector representations alongside factors that measure the seriousness or comedic value of a movie on the x-axis and the budget of the movie as pictured on the y-axis.

tive batch of data from all users at once' (Xin et al., 2015); thus creating an environment that offers a more memory-efficient system.

2.5. Hybrid Systems

Isinkaye et al. refers to both divisions of personalised recommendations as successful, but suffer from a variety of problems such as the cold-start problem; an issue faced by CF techniques where new items are unable to be recommended as they may have a lack of sufficient or positive ratings (Su and Khoshgoftaar, 2009). Hybrid systems are a combination of both CBF and CF and aim to overlap the advantages of both systems. As CF methods are susceptible to issues regarding new users/items, the addition of content-based systems eliminates the issue as it is dependent on the profile or description of new items (Ricci et al., 2015) as opposed to the feedback on items.

2.6. Assessing Performance

Section 2.4.1 notes that recommender systems aim to either generate predictions or recommend items that a user may find of interest. One may presume predictions and recommendations to be the same concept as both methods give an indication of the preferences of an active user, however, they each provide different outputs and are evaluated using separate techniques:

- *Predictions* are a numerical value that expresses the rating an active user may give an item, for example, a score S between 1 and 5. Given a collection of users U and items J, when these rating values are available, 'this task is most often defined as a regression or (multi-class) classification problem where the goal is to learn a function $f : U \times J \rightarrow S$ that predicts the rating $f(u,i)$ of a user u for a new item i' (Ning et al., 2015).

- *Recommendations* are a list of N items that the system believes the active user will find of interest and is often referred to as *top-N recommendation*; for instance, the set of movies Netflix suggests to a user.

Gunawardana and Shani (2015) uses the term *prediction accuracy* to classify the methods of assessing predictions or recommendations which is referred to as 'measuring the accuracy of ratings predictions, [and] measuring the accuracy of usage predictions' respectively. A third class of prediction accuracy is also mentioned and defined as 'measuring the accuracy of rankings of items' which focuses on the ordering of items according to the user's preferences. Accuracy is assumed to be indicative of a better-performing system where in the context of recommendations, 'more accurate predictions will be preferred by the user'. Starting with the first class of measuring the accuracy of ratings predictions, there are two popular measures used to compute the accuracy of predictions which Sarwar et al. (2001) refers to as statistical accuracy metrics; Root Mean Squared Error

$$RMSE = \sqrt{\frac{1}{N}\sum (actual - predicted)^2}, \tag{3}$$

and Mean Absolute Error

$$MAE = \frac{1}{N}\sum |actual - prediction|. \tag{4}$$

There are conflicting statements between the two articles regarding which of the two accuracy metrics is more popular, with Sarwar et al. (2001) claiming MAE to be more commonly used and 'easiest to interpret directly', however, the article by Gunawardana and Shani (2015) is part of a collection that was published more recently and has been extensively cited.

In regard to the second class of prediction accuracy, it is noted that for 'an offline evaluation of usage prediction', an active user has some of their selections hidden and the task of a recommender system is to produce a recommendation list composed of items that are not already associated with the active user (Sarwar et al., 2001). There are then four possible outcomes (Tables 1) which can be used to compute various quantities including precision, recall (or true positive rate) and false positive rate:

$$Precision = \frac{TP}{TP+FP} \tag{5}$$

$$Recall = \frac{TP}{TP+FN}. \tag{6}$$

Table 1: Classification of the possible result of a recommendation of an item to a user.

	Recommended	Not Recommended
Used	True-positive (tp)	False-negative (fn)
Not Used	False-positive (fp)	True-negative (tn)

For applications where the number of recommendations is of a fixed size, Gunawardana and Shani states the 'most useful measure of interest is precision at N (often written Precision@N)'. For other applications, it is 'preferable to evaluate algorithms over a range of recommendation list lengths', therefore the computation of curves comparing precision to recall known as precision-recall curves, and recall to false positive rate known as Receiver Operator Curves (ROC) curves are best suited. The use of each curve is said to be based on the goals and domain of the application, however, two measures are listed by the article for summarising each curve; F-measure 'the harmonic mean of the equally weighted precision and recall'

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{7}$$

and the Area Under the ROC Curve measurement. In addition to measuring the accuracy of a system, *Novelty*, *Diversity*, *Serendipity*, *Confidence*, *Coverage* and others (Kaminskas and Bridge, 2016) are all popular metrics used to evaluate the overall effectiveness of a recommendation engine given a particular domain.

3. Methodology

We aim to implement and progressively update a memory-based filtering algorithm based on our background research into the topic before moving to more complex machine-learning models within the implementation and evaluation portion of the report. Throughout our investigations, we used Jupyter Notebook alongside the standard `Pandas` and `NumPy` libraries to initialise and manipulate the relevant dataframes. Jupyter Notebook is a useful tool for translating our outputs into a report-ready format as well as visualising our statistical analyses using the `seaborn` library and `matplotlib` which the module is based on.

3.1. Memory-Based Implementation

Section 2.4 of our background research mentions that a list of user ratings can be represented by a ratings matrix. Following the implementation of a memory-based system discussed in the article by Herlocker et al. (1999), we started by creating our own 5×5 ratings matrix shown in Table 2 to represent a corpus of users who have explicitly given ratings for a particular set of items using a value between 1 and 5. As the ratings have been randomly generated, it is expected for there to be no intrinsic meaning in our experimental evaluations, as we are only concerned with testing and analysing the outputs of both user-based (user-user) and item-based (item-item) filtering.

Our investigation focuses on the more popular variant of memory-based filtering, user-based, which establishes a neighbourhood of like-minded users; however, we also implemented the item-based filtering method which computes the similarity between items as opposed to users. The ratings matrix depicted in Table 2 has been intentionally initialised with three hidden ratings, as we expect users with missing ratings on a particular item not to be incorporated into the decision-making process of computing a movie's predicted rating. This is a drawback of the collaborative filtering methodology where even though two users can be similar in all other aspects, we are only able to

Table 2: Ratings matrix with an index representing individual users.

userId	movie1	movie2	movie3	movie4	movie5
0	NaN	4	1	1.0	4
1	NaN	4	2	4.0	1
2	1.0	2	1	4.0	2
3	1.0	3	2	3.0	1
4	3.0	2	3	NaN	4

$user_1$ and $movie_1$ have been highlighted as the active user and item we are generating a predicted rating for.

incorporate those who have given their opinion on the item we are recommending. We have chosen to represent our items within the *Movie* domain on the basis that Portugal et al. (2018) found it to be 'the one mostly used with 31 occurrences among the 121 studies' that the article researched, which is accredited to the ease of data availability.

Our first experiment takes $user_1$ as our active user where we are attempting to predict the rating the active user would give to $movie_1$. Our user-user filtering algorithm titled `user_based_prediction` (see code listing 2) begins by excluding users from our neighbourhood that have not provided an explicit rating for $movie_1$; in this case $user_0$. Once removed, we move onto computing the similarity $sim(u,v)$ between the users remaining using the Pearson correlation coefficient (Herlocker et al., 1999):

$$sim(u,v) = Pearson(u,v) = \frac{\sum (r_{ui} - \bar{r}_u) \cdot (r_{vi} - \bar{r_v})}{\sqrt{\sum (r_{ui} - \bar{r}_u)^2} \cdot \sqrt{\sum (r_{vi} - \bar{r}_v)^2}}. \tag{8}$$

The Pearson measure is popular in literature and ranges between -1 and 1, with 1 representing directly proportional/identical users. An active user's u nearest neighbours $\{v_i \dots v_n\}$ is calculated using the ratings r each user has given the item i we are generating a prediction for; where $\bar{r}_u$ and $\bar{r}_v$ are the mean ratings of the active user and nearest neighbours respectively. This enables us to produce a similarity matrix (Tables 3 and 4) depicting all the possible correlations between user/item pairs.

The similarity matrix is then used to obtain the indices of the k-nearest neighbours of our active user (i.e., most similar users) where the average of their ratings (see Equation 9) on $movie_1$ are taken as the predicted rating for $user_1$. For our user-user based function, when k = 2, the nearest neighbours are $user_3$ and $user_2$ who produce a predicted

Table 3: User-based Pearson correlation

userId userId	1	2	3	4
1	1.000000	0.573964	0.986440	-0.981981
2	0.573964	1.000000	0.612372	0.000000
3	0.986440	0.612372	1.000000	-0.852803
4	-0.981981	0.000000	-0.852803	1.000000

Table 4: Item-based Pearson Correlation

	movie1	movie2	movie3	movie4	movie5
movie1	1.000000	-0.500000	0.866025	NaN	0.944911
movie2	-0.500000	1.000000	0.000000	0.0	-0.738549
movie3	0.866025	0.000000	1.000000	-0.5	0.577350
movie4	NaN	0.000000	-0.500000	1.0	0.500000
movie5	0.944911	-0.738549	0.577350	0.5	1.000000

rating $\hat{r}_{ui} = 1$. The item-item CF function computes a predicted rating $\hat{r}_{ui} = 1.5$ using a neighbourhood composed of $movie_5$ and $movie_3$ when k = 2.

$$\hat{r}_{ui} = \frac{1}{k} \sum \bar{r}_{vi} \tag{9}$$

3.1.1. Normalisation

We can expand upon our previous user-based implementation by adding the concept of *normalisation* to our function which considers that each user will have their own rating scale i.e., while some users are more inclined to give higher rating scores, others may be more reluctant. Two proposed normalisation schemes described by Ning et al. (2015) can be incorporated into our function which returns a final predicted output $\hat{r}_{ui}$ that converts individual ratings to a more universal scale. The first normalisation scheme *mean-centring* subtracts a user's average ratings from each rating where w_{uv} is

the weight or similarity $sim(u,v)$ between our active user and a normal user:

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum w_{uv}(r_{vi} - \bar{r}_v)}{\sum |w_{uv}|}. \tag{10}$$

The second scheme *Z-score*:

$$\hat{r}_{ui} = \bar{r}_u + \sigma_u \frac{\sum w_{uv}(r_{vi} - \bar{r}_v)/\sigma_v}{\sum |w_{uv}|} \tag{11}$$

considers the spread in individual rating scales by dividing a user's mean-centred rating by the standard deviation σ of their ratings. Each scheme has its benefits and drawbacks depending on the application and the variance of scores, for example, Z-score normalisation is prone to be more sensitive in comparison to mean-centring and may compute results outside of the rating scale. Mean-centring has the benefit of easily distinguishing whether a given rating is positive or negative based on the sign of the normalised rating ((Ning et al., 2015)).

3.1.2. Measuring Prediction Accuracy

By incorporating normalisation into our final prediction, we aim to investigate the effects it has on our output by computing a loss value, enabling us to compare both normalisation techniques to an average predicted rating.

Background research into modern-day datasets reveals that real-life data is often sparse and consists of an unbalanced number of users and items (Sarwar et al., 2001), which relates to the issue of scalability mentioned in Section 2.4.2. To simulate a real-life dataset, we defined a script to initialise a 1000x100 ratings matrix that would enable us to control a percentage of the missing ratings (see code listing 3). The matrix consisting of 1000 users and 100 items has been initialised with 75% of the ratings declared as *NaN* (missing). We then use methods derived from machine learning to create a set of randomly sampled test users to generate a loss value by computing the MAE (see Equation 4). To do this, we deconstructed our ratings matrix to a list of 250 user ratings (Table 5) who will act as our test set via the function `generate_test_users` (see code listing 4). Each test user acts as the active user and once all users have a predicted rating, we can then use the statistical accuracy metric MAE to assess the algorithm's performance. It is important to note that the test set can only be composed of non-missing ratings which are hidden before a prediction is made. The size of our

Table 5: Generated test set with predicted rating using `user_based_prediction`.

userId	itemId	actual	predicted
616	92	4	4.370975
444	32	3	5.213319
463	8	2	4.885852
571	86	4	5.449237
741	40	1	4.775622
…	…	…	…

Dataframe totals to 250 unique test users.

neighbourhood k has been set to 40 to match the algorithms we investigate further in the report.

Table 6 shows the results of our experiment which we repeated five times for an average prediction and the normalised predictions; where each iteration generates a separate set of test users and movies. By obtaining the mean and median of the MAE results, we can observe that the normalisation of our output significantly reduces the loss value of our predicted ratings in comparison to a weighted averaged prediction.

Table 6: Measuring the accuracy of normalisation techniques.

	weighted_average[a]	mean_centred	z_score
	2.414656	0.941080	0.938260
	2.503713	0.934943	0.934274
MAE:	2.472077	1.027406	1.026625
	2.485447	0.948069	0.947970
	2.448766	1.026837	1.025826
mean	2.464932	0.975667	0.974591
median	2.472077	0.948069	0.947970

[a] average prediction which has incorporated the weights of each user.

3.2. Introduction to Surprise: kNN

Officially, the memory-based CF algorithm reproduced in Section 3.1 is known as the *k-Nearest Neighbour* (kNN) algorithm. The previous section breaks down the kNN algorithm to provide insight into the workings of recommender algorithms, however, as it is an already well-established algorithm, we opted to use a Python library known as *Surprise*[8] for our remaining experiments. Per the documentation, Surprise is a 'Python scikit[9] for building and analyzing recommender systems that deal with explicit rating data' (Hug, 2020). It offers a range of prediction algorithms popular in recommendation research including 'the baseline algorithms, neighborhood methods, matrix factorization-based (SVD, PMF, SVD++, NMF), and many others.'

Using the Surprise implementation of the kNN algorithm which incorporates Z-score normalisation, `KNNWithZScore`, we repeated our experiment from Section 3.1.2 using the 1000x100 ratings matrix for our dataset. A prerequisite to using Surprise is to appropriately format our data which we achieved using the pandas `.stack()` function which reshapes the ratings matrix into a dataframe composed of a list of ratings with corresponding columns:

`[userId::itemId::rating].`

Alongside the `Reader` object which parses the dataframe, we invoke the Surprise function `Dataset.load_from_df` to create our own custom dataset (see code listing 5).

We must first fit the algorithm to the formatted dataset which outputs a similarity matrix consistent for all users. Using the same test set shown in Table 5, we then generated a predicted rating for each test user using `.estimate()`; a method from the base class `AlgoBase` which the documentation states is where all Surprise algorithmic implementations derive from (see code listing 6). Evaluation of the test results (Table 7) using the `KNNWithZScore` algorithm with a neighbourhood size defaulted to 40 outputs an MAE of 0.923, demonstrating our own implementation of the kNN algorithm works similarly to the scikit toolkit.

[8] *https:// surpriselib.com/*

[9] A Python library specialising in the implementation of machine learning models and statistical modelling.

Table 7: Test set with predicted ratings for each test user-generated using the kNN algorithm.

userId	itemId	actual	predicted
616	92	4	1.774518
444	32	3	2.729728
463	8	2	2.466678
571	86	4	2.742062
741	40	1	2.076350
...	...	...	...

Composed of the same test users as Table 5.

4. Implementation and Evaluation

4.1. Choosing a Dataset

GroupLens[10] describe themselves as 'a research lab in the Department of Computer Science and Engineering ... specializing in recommender' and other systems. GroupLens Research provides a large collection of rating datasets obtained from MovieLens[11] who use collaborative filtering technology to provide movie recommendations. From the list, we chose an older dataset released in 1998 titled *MovieLens 100K Dataset* (Harper and Konstan, 2015), consisting of 100,000 user ratings using a 5-point rating scale provided by 943 users across 1682 movies. MovieLens 100K (alias ml-100k) is divided into multiple files which include extra information regarding the participants and genres of the movies which we will investigate further in the report, but our main focus is the rating data. The first five instances of ml-100k shown in Table 8 reveal a series of records that follow the same structure as our custom dataset in Section 3.2; however, ml-100k also includes the `[timestamp]` column indicating the time 'represented in seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970' (TensorFlow, 2022). As we wish to keep the dataset as simplistic as possible and cannot predict the effect `[timestamp]` will have upon our algorithms, we decided to remove the feature

[10] *https:// grouplens.org/*
[11] *https:// movielens.org/*

column[12].

Table 8: Head of the dataset titled *MovieLens 100K Dataset*

	userId	movieId	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596
...	...	...	...	...

A bar plot representing the dataset's rating distribution can be seen in Figure A.1.

The README.txt file associated with ml-100k asserts that each user has rated at least 20 movies which we confirmed through a histogram shown in Figure A.2. This plot can also be represented by a *long tail plot* (Figure 3) which we instantiated using another Python library known as `recmetrics`[13]; a library of metrics for assessing recommender systems. The purpose of this plot is to provide a visual representation of the dataset's distribution which has a cutoff (red line) where both the head and tail are equal. The head contains items with a large accumulation of ratings in comparison to the tail and a "good" recommender should be able to balance sparsity requirements with *popularity bias*[14]; where popular items are repeatedly suggested, 'often ignoring long-tail (i.e., niche) ones, even if the latter would be more interesting for individuals' (Yalcin and Bilge, 2022).

We calculated the percentage of rated items within ml-100k to be $\approx 6.73\%$ which we pictured using a sorted heatmap (Figure A.3). Surprise offers a range of built-in datasets including ml-100k which we opted not to use as it would not enable us to manipulate the data further into the project if necessary.

[12] *Feature engineering* is the term given for selectively choosing specific features from our data as a means of optimising the performance of our algorithms and ML models (Chen et al., 2020).

[13] https://github.com/statisticianinstilettos/recmetrics

[14] https://github.com/statisticianinstilettos/recmetrics

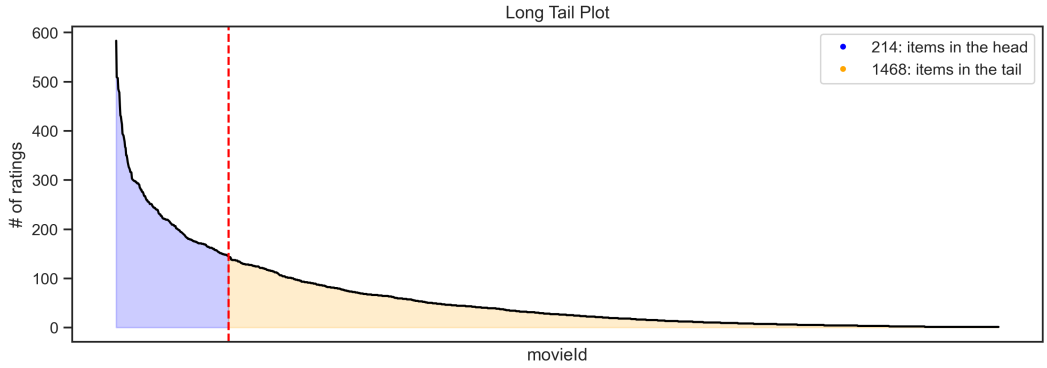


Figure 3: Long tail plot representing the distribution of user ratings.

4.1.1. kNN Investigation

With the establishment of a dataset collected from a real-life source, our next experiment was to investigate the effects of adjusting the value of k, on top of analysing the performance of two other similarity measures (Ning et al., 2015; Hug, 2020). For a user-based filtering system, the first measure titled *Mean Squared Difference* (MSD) similarity can be defined as:

$$MSD(u,v) = \frac{|I_{uv}|}{\sum (r_{ui} - r_{vi})^2} \tag{12}$$

where I_{uv} denotes the set of items rated by both the active user u and a normal user v. The second similarity measure *Cosine Vector*:

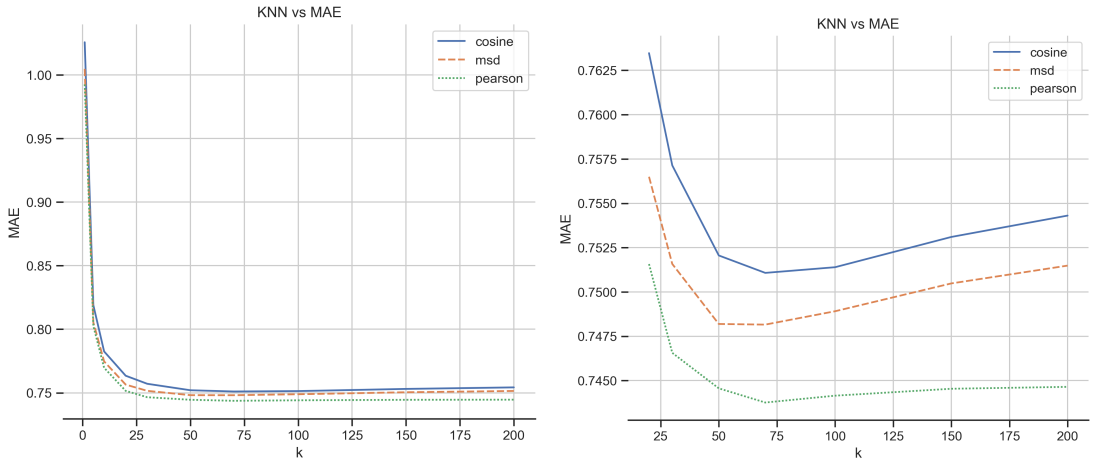
$$cos(\vec{A},\vec{B}) = \frac{\vec{A} \bullet \vec{B}}{||\vec{A}|| \cdot ||\vec{B}||} \tag{13}$$

has been translated to the context of recommender systems and considers each user u,v to be a vector:

$$cosine\_sim(u,v) = \frac{\sum r_{ui} \cdot r_{vi}}{\sqrt{\sum r^2{}_{ui}} \cdot \sqrt{\sum r^2{}_{vi}}}. \tag{14}$$

We have so far only used the Pearson correlation coefficient and a neighbourhood size of 40 to measure the linear association between users/items, and so the aim is also to see if there will be a reduction to our MAE using a proper dataset as opposed to using an artificial example; as there should be more meaning to the ratings which have been collected from actual individuals.

Figure 4 shows the results of our experiments using the user-based `KNNWithZScore` filtering algorithm evaluated using a train/test ratio of 0.75 i.e., 75% of our data was used to train the model while the remaining 25% was used to generate our final loss value. Figure 4a show a steep decrease in the MAE which levels off around a k value of 35 and initially appears to remain between approximately 0.745 and 0.755 for all similarity measures; however, Figure 4b shows a rise in the MAE after a k value of 75 for all similarity plots. This is to be expected on the basis that as we increase the number of potential users in our neighbourhood, we increase the population of those who may have a low correlation (i.e., are dissimilar in terms of their likes and dislikes) to our active user's. Our plot also verifies the Pearson's correlation coefficient to be the most effective at generating our neighbourhood of users as it computes the lowest MAE score of ≈ 0.7440.



(a) Zoomed out.

(b) Zoomed in.

Figure 4: Investigation of the performance of the kNN algorithm at various k values and similarity measures.

Our investigation points to an optimal neighbourhood size at k = 75 as opposed to the default value of 40 provided by the Surprise implementation which lists the benchmark MAE for the algorithm using the same dataset at an average of 0.774; calculated using a five-fold cross-validation approach. Cross-validation is a resampling technique closely tied to ML where the data is split into k partitions known as folds. $k-1$ folds are used as the training set while the remaining data is used to validate the results and each fold generates a loss value which can be aggregated to obtain a final loss value to

evaluate the performance of our algorithms. Where our algorithmic process differs is we opted to use a single train/test split for each evaluation due to constraints pertaining to compilation time when we initially attempted to use a cross-validation approach. The results also generated a lot of noise within our graph due to the small subset of k values we used which would have further increased compile time if we attempted to meet the optimal experimental conditions i.e., increasing the length of k variations. While it is more accepted to perform multiple tests on a single dataset via cross-validation or other methods and aggregate the results[15], we are still able to verify literature claims regarding the Pearson correlation coefficient being the optimal similarity metric; as well as visualise the importance of choosing a suitably sized neighbourhood (Herlocker et al., 1999) for our application. Ideally, we would also repeat our experiments using multiple train/test split ratios which may yield different results as shown by Sarwar et al. (2001) who performed a similar experiment on top of adjusting the train/test ratio for both user-user and item-item filtering methods. A final point we are also able to observe is a reduction in the error in comparison to our artificial example, possibly due to hidden intrinsic meaning in the data which we mention in the following section.

4.1.2. Dataset Investigation

During the implementation of our ML-based algorithm, the question arose as to how we could verify if the ratings from the ml-100k dataset were artificial or collected from a real-life source; and if the similarity computation between our items (movies) was valid for the kNN algorithm we covered. Previously, the similarity or weight between movies was represented as a value between -1 and 1. As we only used an index value to title each movie (e.g., $movie_1$) and each movie's features is comprised only of their given rating, we have no details regarding what types of movies are present within a neighbourhood; and essentially what movies would influence the recommendations to an active user for an item-item based filtering system. Fortunately, the ml-100k separates itself from others as it consists of an index file containing extra information pertaining to each movie and user in the form of files detailing special characteristics such as a movie's title and genres, and user occupations. We can use the information of each movie (i.e., their title and genre) to give us a better idea of how well the kNN algorithm computes

[15] This is to avoid *selection bias* where bias is introduced 'toward a specific subgroup of the target population' (Nikolopoulou, 2022).

the similarity between movies, as well as assess whether the ratings provided have a structure to them or if they are randomised; as a randomised dataset should have little to no consistency in the types of movies within an items neighbourhood.

Using the `KNNWithZScore` algorithm we used for our experiments in Section 3.2 we created an item-item similarity matrix as opposed to a user-user matrix by defining the relevant similarity measure configuration. To be able to obtain the neighbours of a movie, Surprise offers two functions (see code listings 7 and 8) which initially convert the id of each movie, referred to as raw ids, to inner ids which are stated to be 'more suitable for Surprise to manipulate'[16]. A second function is then used to compute the top k neighbours of the movie *Toy Story (1995)* which we are using for our investigation which gives us the output seen in Figure 5 when k is set to 10.

```
Out[9]: ['Line King: Al Hirschfeld, The (1996)',
         'Inkwell, The (1994)',
         'Stars Fell on Henrietta, The (1995)',
         'Critical Care (1997)',
         'Old Lady Who Walked in the Sea, The (Vieille qui marchait dans la mer, La) (1991)',
         'Jerky Boys, The (1994)',
         'Scarlet Letter, The (1995)',
         'Stranger, The (1994)',
         'Newton Boys, The (1998)']
```

Figure 5: The 10 most similar movies to *Toy Story* based upon their computed weightings.

Our assessment of how well the kNN algorithm performs at computing movie similarity will be defined by the top genres of the movies within our selected movies neighbourhood which we would expect to mimic *Toy Story*. To obtain the genres of each movie, the *u.genre* file located within the ml-100k index provides a sorted list of movie genres (Figure 6) which translates to the encoded genres column within *u.item*; the index file containing the characteristics of each movie such as their title, year of release, genre and more (Table 11).

The genres column within *u.item* has been encoded using a similar method to *one-hot encoding*; a technique prominent in ML used to reduce categorical attributes to a binary variable; however, it differs in the fact that some movies consist of multiple genre representations. We defined a script to first decode the genre columns for each movie into their string representations based upon the list of genres (see code listing

[16] *https://surprise.readthedocs.io/en/stable/FAQ.html#what-are-raw-and-inner-ids*

```
Out[8]: 0          unknown
        1           Action
        2        Adventure
        3        Animation
        4      Children's
        5           Comedy
        6            Crime
        7      Documentary
        8            Drama
        9          Fantasy
        10       Film-Noir
        11          Horror
        12         Musical
        13         Mystery
        14         Romance
        15          Sci-Fi
        16        Thriller
        17             War
        18         Western
        dtype: object
```

Figure 6: List of genres available for movies within the ml-100k dataset

9). The `groupby` clause provided by the pandas library can then be used alongside the aggregate function `.count()` to obtain a count for each genre within the movies neighbourhood which we visualised using a `matplotlib` pie chart. What we were able to deduce from our experiments in Section 4.1.1 is that the size of k impacts the type of movies present within our neighbourhood and ultimately the final error value. With consideration for k, we repeated the investigation by altering the number of neighbours in each iteration and selecting the top 10 genres present within each neighbourhood (see code listing 10). Our results illustrated by Figure 7 would indicate that *Toy Story* is a comedy or drama film as these two genre categories dominate the pie chart for each k iteration, however, the output from our decoded genres columns states *Toy Story* is comprised of the genres *Animation, Children's, and Comedy*.

We could not determine whether the poor results were due to how the `KNNWithZScore` algorithm uses the user ratings to compute the weight between movies or if the data lacks meaning in the form of similarity between users/items, so we aimed to see if we could improve the neighbourhood corpus by utilising another variant of the kNN algorithm titled `KNNBaseline`. What differentiates `KNNBaseline` to our previous kNN implementation is the addition of a baseline estimate b_{ui} or *bias* integrated into an item-based prediction via the formula:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum sim(i,j) \cdot (r_{uj} - b_{ui})}{\sum sim(i,j)}. \tag{15}$$

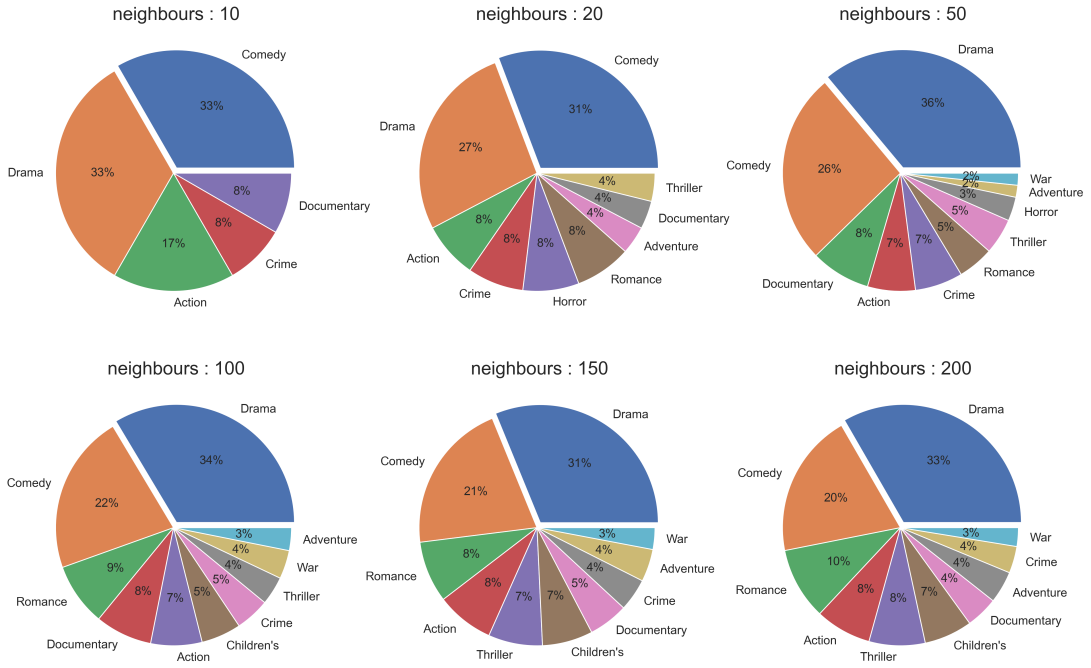


Figure 7: Depiction of the genre distribution present within the neighbourhood corpus for *Toy Story* at various k intervals. Note that the distribution results may not always sum to 10.

Koren (2010) explains that the baseline estimate takes into account the overall average rating μ and is calculated by observing deviations from the averages associated with a user b_u and item b_i:

$$b_{ui} = \mu + b_u + b_i \tag{16}$$

By repeating our experiment with Surprise's `KNNBaseline` algorithm while also using the `Pearson baseline` similarity configuration parameter recommended by the Surprise documentation, our results shown in Figure 8 indicate a more positive outcome. In comparison to Figure 7, the distribution of genres within each pie chart is more evenly spread and dominated primarily by the *"Comedy"* genre for all charts alongside the *"Action"* and *"Drama"* categories for larger neighbourhood sizes. The 10 nearest neighbours consist of all the genres listed for *Toy Story*, however, from what we know regarding serendipity, the incorporation of other genres as present within our larger neighbourhood sizes can often be more favourable for real-life solutions; as they introduce users to a greater selection of items while not constricting them to only their

interests.

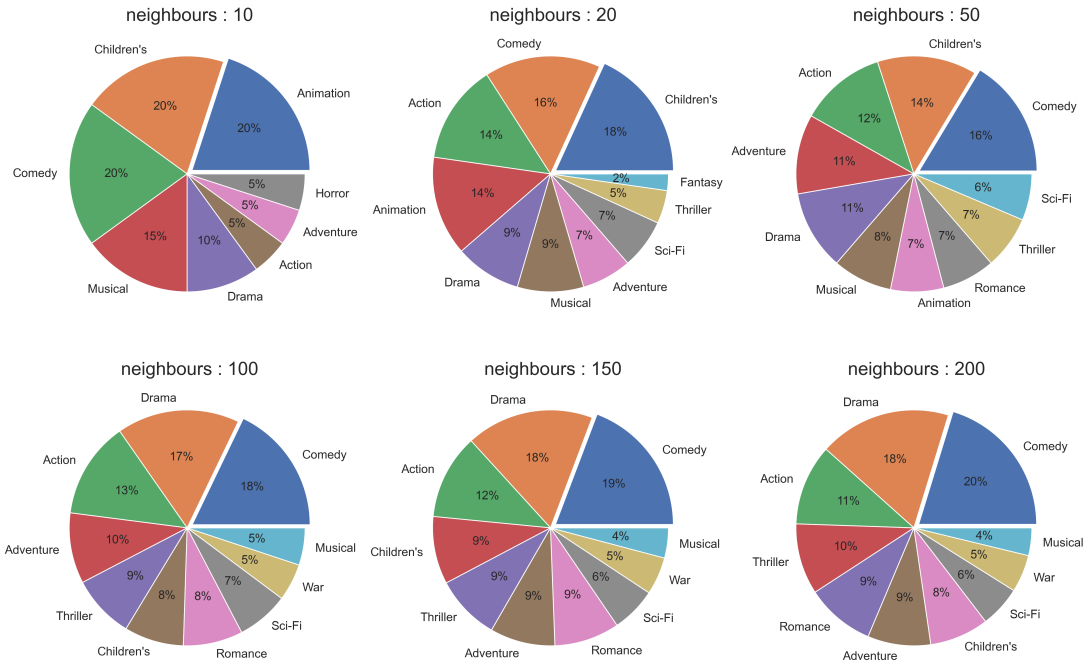


Figure 8: Genre distribution present within the neighbourhood corpus for *Toy Story* at various k intervals using the `KNNBaseline` algorithm.

4.2. Model-Based Filtering: Matrix Factorisation

So far, the algorithms we implemented require a similarity matrix to be manipulated in order to generate a predicted rating. Section 2.4.2 of our background research describes singular value decomposition (SVD) as a matrix factorisation (MF) technique which aims to reduce the dimensionality of the user-item ratings matrix through factorisation (hence the name):

$$\begin{bmatrix} \hat{r}_{11} & \cdots & \hat{r}_{11} \\ \vdots & \ddots & \vdots \\ \hat{r}_{n1} & \cdots & \hat{r}_{nm} \end{bmatrix} = \begin{bmatrix} p_1^T \\ \vdots \\ p_n^T \end{bmatrix} \begin{bmatrix} q_1 & \cdots & q_m \end{bmatrix}.$$

Our decomposed ratings matrix allows us to rewrite Equation 1 to:

$$\hat{r} = p^T q \tag{17}$$

where the user matrix p is $k \times n$ and the item matrix q is $k \times m$ where k denotes latent[17] features. The resulting dot product $\hat{r}$ is the full $n \times m$ ratings matrix consisting of all user-item interactions (i.e., rating values). The default Surprise implementation of the SVD algorithm modelled after the one used in the Netflix prize competition incorporates bias into the formula which we mentioned is to keep each user on a similar rating scale. This addition means that our decomposed ratings matrix formula can be rewritten as:

$$\hat{r} = \mu + b_u + b_i + p^T q \tag{18}$$

where we are training our model to minimise the regularised squared error (Hug, 2020; Koren et al., 2009):

$$\min_{q*,p*,b*} \sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 \right) \tag{19}$$

and ultimately estimate the unknown ratings matrix values.

4.2.1. Hyperparameter Tuning

As outlined in Section 2.2, learning rate γ and regularisation rate λ are two prominent hyperparameters in machine learning which Surprise's `SVD` algorithm enables us to control. Additionally, we are able to alter another hyperparameter known as the number of *epochs*. Each epoch constitutes a single pass of the training data which progressively minimises the model parameters:

$$b_u \leftarrow b_u + \gamma(e_{ui} - \lambda b_u) \tag{20}$$

$$b_i \leftarrow b_i + \gamma(e_{ui} - \lambda b_i) \tag{21}$$

$$p_u \leftarrow p_u + \gamma(e_{ui} \cdot q_i - \lambda p_u) \tag{22}$$

$$q_i \leftarrow q_u + \gamma(e_{ui} \cdot p_u - \lambda q_i) \tag{23}$$

using stochastic gradient descent (SGD). Our plot analysing the performance of the default `SVD` algorithm at varying epochs (Figure 9) mimics the line plot presented in the article by Mnih and Salakhutdinov (2007) for 'a simple SVD' model. As illustrated, there is a sharp decrease in the error value of the algorithm before it gradually increases at an optimal 20 iterations. This is as a result of the model becoming too complex when fitted to the training data (also known as *overfitting*) leading to poor results on unseen

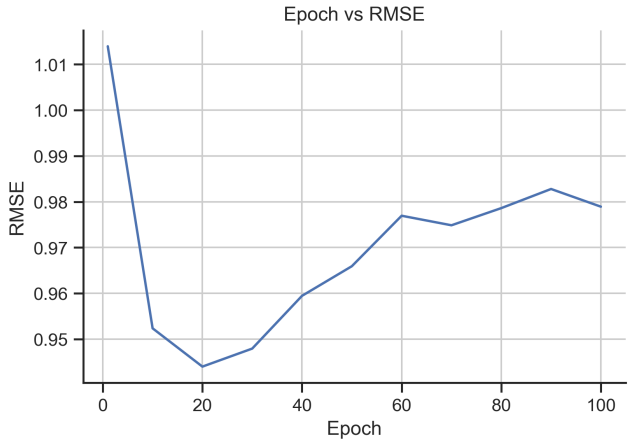


Figure 9: Visualisation of the performance of the `SVD` model at various epoch iterations.

data, and it is the role of a researcher to deduce the optimal parameter configuration to achieve the lowest error score.

Regularisation keeps a model's complexity in check while the learning rate controls the *step size* at each epoch. In relation to stochastic gradient descent (SGD), step size refers to how much a model's parameters are updated in an attempt to reach a global minimum point on our loss curve assuming the curve is convex[18]. Using the same epoch values we used in Figure 9, we altered the learning and regularisation rates to visualise how these hyperparameters would affect the SVD algorithm. The boundaries we used constituting high and low, learning and regularisation rates were inspired by Google's *Advanced Course on Recommendation Systems*[19].

Figure 10 shows that a low learning rate of 0.00005 does little to reduce the model's loss value at each epoch as the step size for reaching the global minimum increases slowly; meaning the model would require an increased number of passes of the training data. In contrast, a high learning rate of 0.05 initially increases the model's error computation before the model gradually reduces the RMSE. A Low regularisation value of 0.0002 appears to follow the same trend as our plot in Figure 9 except the optimal number of epochs is at 10 and the RMSE is at its lowest at a score of approximately 0.952. A high regularisation rate of 2 prevents the model from effectively fitting to the

[17] Not directly observed.

[18] *https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate*

[19] *https://developers.google.com/machine-learning/recommendation*

training set, meaning when it is evaluated using the unseen test data, it *generalises*[20] poorly.

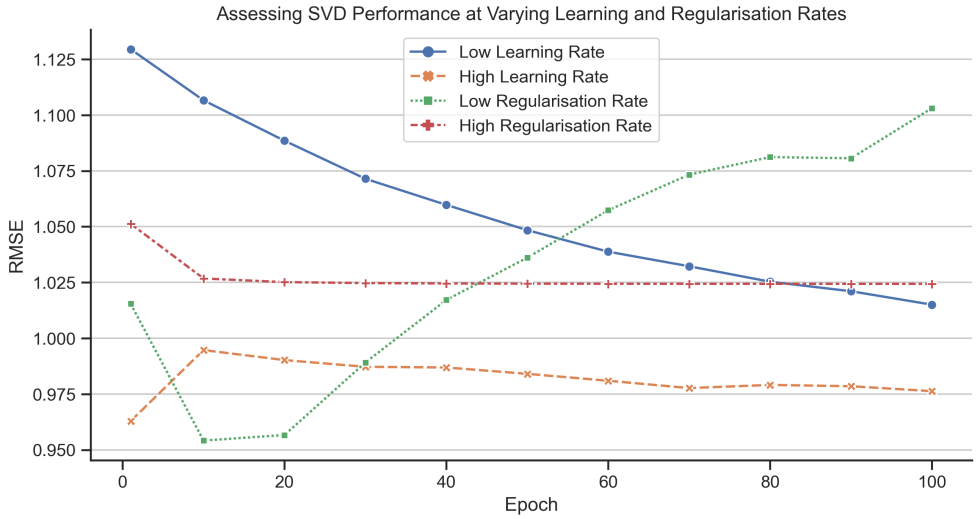


Figure 10: Performance of the `SVD` algorithm at various learning and regularisation parameter rates.

To finalise our investigation of the best hyperparameter configuration for our matrix factorisation-based model, we performed a grid search using `GridSearchCV` to perform an 'exhaustive search over [the] specified parameter values' (Pedregosa et al., 2011). Lines 9 and 10 of our script below show all the parameter configurations tested by the grid search algorithm which uses cross-validation to evaluate each configuration.

```
1  from surprise import SVD
2  from surprise.model_selection import GridSearchCV
3
4  # dataset
5  reader = Reader(line_format="user item rating", sep="\t")
6  data = Dataset.load_from_file("ml-100k/u.data", reader=reader)
7
8  param_grid = {
9      'lr_all' : [0.00005, 0.0005, 0.005, 0.05, 0.5],
10     'reg_all' : [0.002, 0.002, 0.02, 0.2, 1]
```

[20] How the model adapts to new, unseen data.

```
}
gs = GridSearchCV(SVD, param_grid)
gs.fit(data)
```

Listing 1: Grid search for the best-performing learning rate and regularisation hyperparameters.

Our grid search results support Surprise's default parameter combination of a learning rate equivalent to 0.005 and a regularisation value set to 0.02; producing an optimal RMSE score of 0.937 which was close to the documentations benchmark score of 0.934.

4.2.2. Top-N Recommendations

To give us a sense of the rating values computed from testing our model, Figure 11 features a histogram and boxplot to showcase the predicted rating distribution present after the `SVD` model is fit to a train-test ratio of 0.75. For this example, the lower and upper quartiles of the boxplot are 3.146 and 3.995 respectively which surround a median value of 3.594. An interesting observation is that the upper whiskers of the boxplot do not consider rating values of 5 to be outliers compared to ratings below ≈ 1.9, represented by fliers preceding the boxplot's lower whisker. One could argue that the extreme bounds of a perfect rating is the reason why 5 is still within the boxplot's range, and we can see it is closer to the peak of the distribution curve while also being more frequently used in contrast to lower rating scores between 1 and 2.

It should be noted that the use of a train-test split means the rating distribution for our test set sums to 25000 individual predicted ratings (Table 9) out of the full dataset which totals to $100,000$ observable user ratings. As our experiments involve us comparing each of our algorithm's predictions with the actual rating, it makes sense that we only make predictions on rating values that are known; as we can then use the accuracy and other evaluation metrics to determine the overall performance of our algorithms.

The predicted ratings on our test set allow us to move on to recommending movies to users by determining the top-n recommendations for an active user. For this, we first need to determine the "useful" movies of a user to then compare with their recommended movies. Using our list of predicted test-user ratings, we grouped the index of each user's rated movies into a single list for each test user. We then reduced each list to what we would classify as useful for this investigation; that being an arbitrary N value of the 10 highest-rated movies by a test user (see code listing 11). Finally, we repeated this

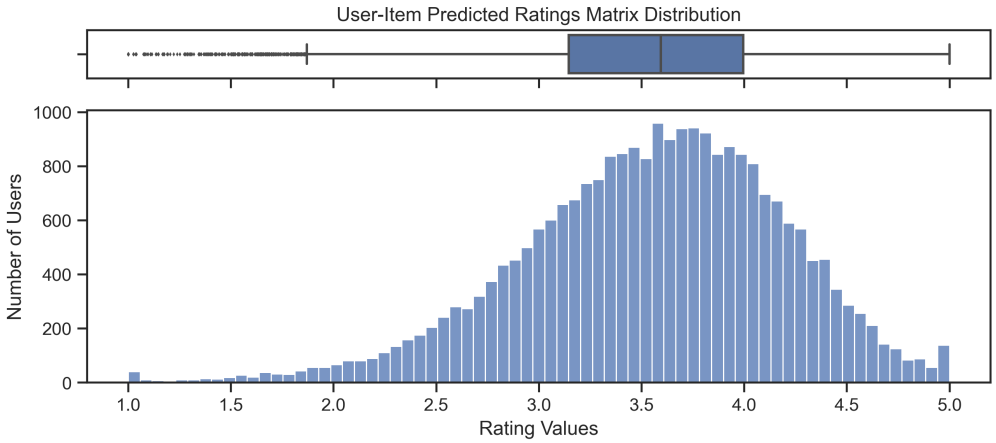


Figure 11: Centrally aligned histogram and boxplot distribution charts of the predicted ratings output generated from matrix factorisation.

Table 9: Actual and predicted ratings for test users extracted from ml-100k.

	userId	movieId	actual	predicted
0	345	715	4.0	3.541539
1	92	998	2.0	2.905483
2	934	195	4.0	4.050726
3	586	423	2.0	3.932764
4	336	383	1.0	2.056852
...	...	...	...	...

process for the recommendation list corresponding to each test user by sorting the values of their predicted ratings; giving us the dataframe represented by Table 10. As each test user within our dataframe has been randomly sampled from the full dataset, some test users may have a large variety of movie ratings while others may have few; however, our script which outputs the recommendation list (see code listing 12) ensures the length is equivalent to each test user's useful list.

The results of recommending an item to a user are represented by the four quantities

Table 10: Index list of useful movies alongside list of recommended movies computed using the `SVD` model.

userId	useful	recommendations
1	[1, 113, 114, 124, 152, 169, 172, 181, 19, 196]	[172, 114, 169, 209, 156, 262, 208, 79, 135, 181]
10	[100, 133, 183, 211, 223, 285, 483, 498, 602, ...	[513, 483, 285, 183, 498, 606, 529, 651, 707, ...
100	[315, 272, 344, 347, 690, 751, 898, 270, 286, ...	[272, 347, 315, 270, 751, 344, 333, 286, 326, ...
101	[117, 405, 50, 546, 866, 924, 225, 280, 471, 597]	[50, 7, 117, 845, 405, 1093, 471, 866, 111, 924]
102	[101, 195, 202, 307, 510, 98, 121, 154, 176, 235]	[313, 98, 79, 202, 82, 176, 522, 510, 7, 230]
...	...	...

shown in Table 1 which we calculated using the following formulae:

$$TP = \text{useful items} \cap \text{recommended items} \tag{24}$$

$$FP = \text{useful items} - \text{recommended items} \tag{25}$$

$$FN = \text{useful items} - \text{recommended items} \tag{26}$$

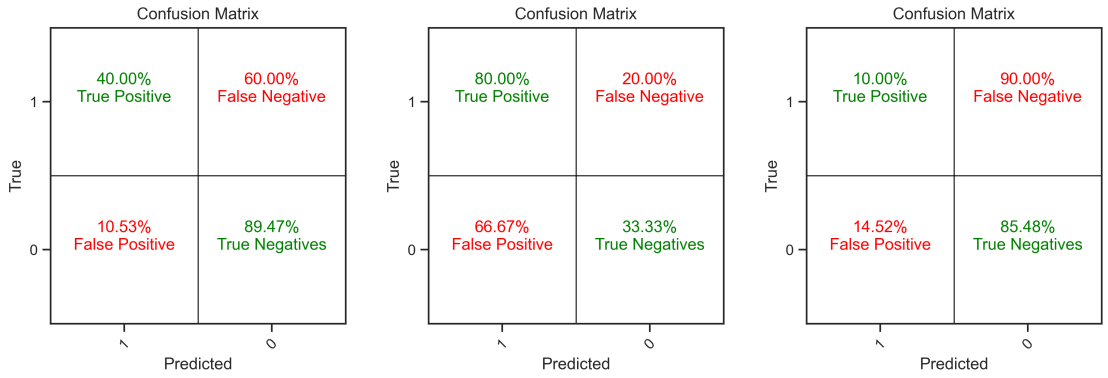
$$TN = (\text{all items} - \text{useful items}) \cap (\text{all items} - \text{recommended items}) \tag{27}$$

Figure 12 presents the confusion matrix results for three active users using another method defined by the `recmetrics` library which represents the quantity distributions using percentages. True positive and true negative characterise an outcome where the model correctly predicts the positive and negative classes respectively. Conversely false positive and false negative are instances where the model predicts each class wrong.

4.2.3. Precision and Recall

Each of our active user's confusion matrix results depicted in Figure 12 consists of varying distributions for each quantity, so we can use the precision and recall accuracy metrics (see Equations 5 and 6) to evaluate the `SVD` model's performance. Using the Surprise function, `precision_recall_at_k`, we are able to calculate both accuracy metrics for a list length of N. On top of passing in our list of predictions alongside their actual rating (Table 9), the function provides two additional arguments enabling us to control the threshold rating of what constitutes a useful item and the length of the recommendation list.

Formally, the rating scale ml-100k uses (value between 1 and 5) is known as a *likert scale* which Jamieson (2004) describes as a common method for measuring a person's



(a) $user_1$ (b) $user_{623}$ (c) $user_{280}$

Figure 12: Confusion matrix results for three randomly selected test users.

attitude within an 'ordinal level of measurement' The article conveys that the 'intervals between values cannot be presumed equal' which informs us that the difference between ratings on a scale of 1 to 2 is inconsistent with those from 4 to 5. This means that a statistical quantity (i.e., the median or mode) should be used to determine a 'measure of central tendency'. For our own problem, the mean rating value of our population is 3.529860 while the median is 4 so we settled on a threshold of 4 which would classify movies rated at or above this value to be placed within each list for every test user. The length of each useful movies list is composed of all movies that meet the threshold as opposed to the recommendation list which is fixed to a maximum size of N when using the `precision_recall_at_k` function. As there are no constraints on the application domain we are working within, we can experiment with lengths of varying sizes and investigate how both precision and recall change at various lengths; enabling us to plot both metrics.

We defined a script which utilises the `precision_recall_at_k` function by calculating and averaging both accuracy metrics at list lengths starting at 1 and ending at 49; incremented by 2 for each cycle; all the while each iteration is repeatedly averaged using a 3-fold cross-validation approach (see code listing 13). The result is a list of precision and recall values at various list lengths which we plotted using a standard `matplotlib` scatter plot (Figure 13).

We initially chose the precision-recall curve as our evaluation plot for the `SVD` algorithm due to Section 2.6 of our background research which highlights it is best suited for applications where the recommendation list length is not fixed; however, when it

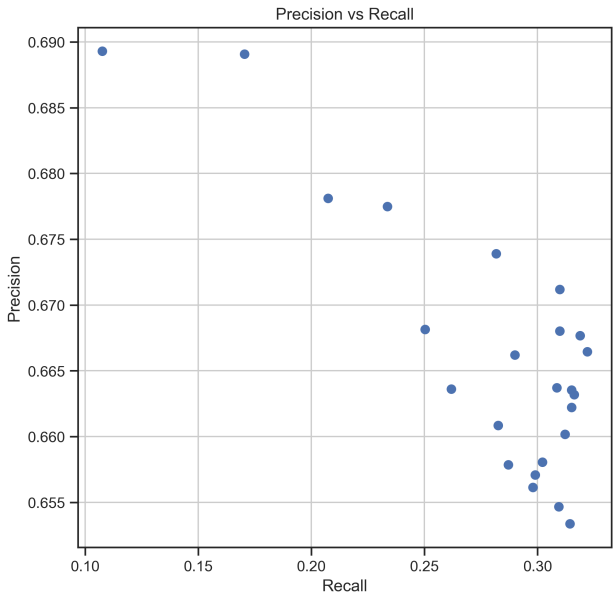


Figure 13: Precision-recall scatter plot for `SVD` model at various list lengths.

came to using the libraries to display our curve, the functions required a probability distribution over a set of classes. This is the same for the alternative ROC curve where the probability distribution is referred to as a *threshold* value calculated from the confidence the model has in its predictive ability. This is typically found in binary classifiers and instead of plotting the averaged precision and recall scores across a range of list lengths, it is more accepted to analyse the precision and recall scores across a variety of thresholds which is not provided by the Surprise library.

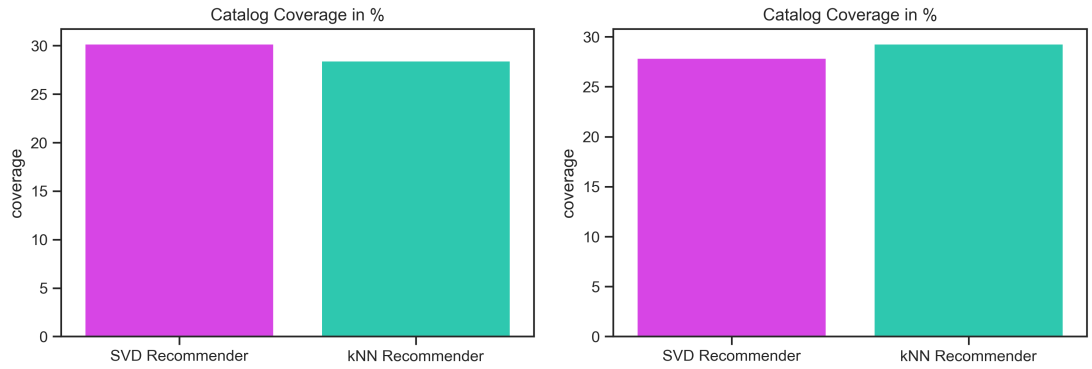
While our scatter plot is not a standard representation of a precision-recall curve, we can still infer an overlapping observation which references the inverse relationship between the two metrics; where increasing precision adversely affects a model's recall; likewise prioritising a high recall score reduces a model's precision. For our particular problem, Figure 13 shows the model is more precise for a smaller recommendation list meaning more relevant items are selected, while larger lists lead to an increase in the recall; therefore there is a greater 'fraction of relevant items in all relevant items' (Wang, 2022). The tension between both metrics prevents us from maximising each at a time so we can use the harmonic mean of precision and recall; also known as F-measure or F_1-*score* (7) to summarise the model's performance at each list length. Shown in Table 12 are the precision-recall values used to display our previous scatter plot alongside their

respective F-score. The score ranges from 0 to 1 where 1 describes a model with perfect predictive capability while 0 is the 'worst possible score' (Allwright, 2022). What we can observe from our table is that increasing the number of recommended movies for our test users correlates to a greater F_1-score; which is true up to a length of ≈ 27 where the score hovers around 0.42. Nevertheless, for a real-life scenario, it would be left to the researchers to determine which size is most appropriate for their business needs based on the domain they operate in.

4.2.4. Catalogue Coverage

Coverage often referred to as *catalogue coverage* is a measure of 'the percentage of all items that can ever be recommended' (Gunawardana and Shani, 2015). Recommender systems with a low coverage score will only recommend items that are more popular or have a greater number of ratings which relates back to the cold-start problem discussed in Section 2.5, as items lacking in this area will be forgotten. To calculcate the `SVD` model's coverage score, we reused our code for obtaining the useful movies indices list (see code listing 11) and in contrast to selecting the 10 highest-rated movies using the `.nlargest()` funcion, we used the `pandas .query()` method to define a threshold $>= 4$; meaning movies rated at or above this value are added to the useful movies list. Our script defined by refactoring the example within the `recmetrics` repository (see code listing 14) produces the output shown in Table 13 which ensures the recommender matches the useful movie indices list's length. To make a comparison, the table includes the recommendations outputted from the `KNNBaseline` algorithm which has also been fitted to a 0.75 train-test ratio.

Figure 14 shows the coverage percentages for each of the recommenders represented by a bar plot. Multiple figures are included as each time we re-run the code, comparing the two algorithms show the greater overall coverage score is not consistent and varies between 27% and 32%. We theorised that due to each algorithm falling under the CF methodology and both incorporating a bias value into their training formula, it would make sense that their outputs would be on a similar scale. Our results could also be a result of us selecting 4 as the threshold value which limits the number of movies that can be recommended at a time from our test set, however, without feedback from real users and knowledge of the domain we are working in, it is difficult to assess each algorithm's coverage scores.



(a) Greater `SVD` coverage score.

(b) Greater `KNNBaseline` coverage score.

Figure 14: Analysis of the coverage scores for both the `SVD` and `KNNBaseline` algorithm's.

5. Conclusion and Future Work

Our research into recommendation systems first covered a memory-based filtering algorithm which computes a confusion matrix to determine the similarity between users within our self-initialised ratings matrix. While successful in being able to generate a predicted rating for an active user by taking the mean of similar users' ratings, we improved the algorithm by incorporating normalisation into the calculation. To evaluate the effect normalisation had on our output, we initialised a larger ratings matrix that would enable us to control the percentage of ratings present within the matrix, thus introducing sparsity; and used a technique derived from machine learning to generate a set of test users to evaluate the refined algorithm. Alongside verifying our own implementation, the memory-based algorithm (officially titled k-nearest neighbour) was a good entryway to the introduction of the Surprise library which offers a variety of pre-built recommender models we used to progress our experiments.

Moving onto the implementation and evaluation portion of the report, we provided a visual and statistical analysis of the MovieLens 100k dataset composed of real ratings made within the movie domain. This would be the data used for our remaining experiments so we investigated both literature claims regarding the performance of Surprise's kNN algorithm and the validity of the dataset to ensure the ratings were not artificial. Our results confirmed Pearson's similarity measure to be the optimal similarity configuration for our problem while also confirming the reliability of the dataset.

The concluding report segments focused on the matrix factorisation technique, SVD, specifically chosen due to its notoriety in the collaborative filtering field. So far, we had only calculated the predicted ratings for each active user in our test sets. As a result, we generated a personalised recommendation list for each test user that we then compared to the list of items they had individually highly rated. The output enabled us to utilise the precision and recall accuracy metrics as well as the model's coverage score to evaluate the system, however, in relation to serendipity, these kinds of problems have no right answer and are dependent on the domain being researched.

An important point to consider is that our algorithmic implementations and evaluations were conducted offline using datasets that aim to simulate real-life user behaviour. This allows researchers to 'compare a large range of candidate algorithms at a low cost' (Gunawardana and Shani, 2015) but provides evaluations that only answer a narrow range of questions. It is therefore often preferable to conduct online tests or *user studies* which garner feedback from live users in the form of their general behaviour and qualitative answers. Live user interactions would also have been useful for extending the investigation covered in Section 4.2.2 to also include an analysis of the effects of item positioning which comes under the heading of *ranking measures*.

As it was a necessity to first explore the concepts associated with our research at a surface level, future work could also spend less focus on the individual systems and more on a generalised outlook of the large scale of algorithms to perform comparisons of each. This could include investigating hybrid systems which had one of the least number of studies conducted on it according to research by Portugal et al. (2018); next to content-based methods.

References

Allwright, S. (2022). What is a good f1 score and how do i interpret it?

Amatriain, X. and Basilico, J. (2015). Recommender systems in industry: A netflix case study. In *Recommender systems handbook*, pages 385–419. Springer.

Bennett, J., Lanning, S., et al. (2007). The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York.

Burke, R. (2002). Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370.

Chen, R.-C., Dewi, C., Huang, S.-W., and Caraka, R. E. (2020). Selecting critical features for data classification based on machine learning methods. *Journal of Big Data*, 7(1):52.

Dias, M. B., Locher, D., Li, M., El-Deredy, W., and Lisboa, P. J. (2008). The value of personalised recommender systems to e-business: a case study. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 291–294.

Do, M.-P. T., Nguyen, D., and Nguyen, L. (2010). Model-based approach for collaborative filtering. In *6th International conference on information technology for education*, pages 217–228.

Google Developers (n.d.). Machine Learning Glossary | Google Developers — developers.google.com. *https://developers.google.com/machine-learning/glossary*. [Accessed 05-May-2023].

Gunawardana, A. and Shani, G. (2015). Evaluating recommender systems. In *Recommender systems handbook*, pages 265–308. Springer.

Harper, F. M. and Konstan, J. A. (2015). The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4).

Herlocker, J. L., Konstan, J. A., Borchers, A., and Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 230–237.

Hug, N. (2020). Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174.

Isinkaye, F. O., Folajimi, Y. O., and Ojokoh, B. A. (2015). Recommendation systems: Principles, methods and evaluation. *Egyptian informatics journal*, 16(3):261–273.

Jamieson, S. (2004). Likert scales: How to (ab) use them? *Medical education*, 38(12):1217–1218.

Kaminskas, M. and Bridge, D. (2016). Diversity, serendipity, novelty, and coverage: a survey and empirical analysis of beyond-accuracy objectives in recommender systems. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 7(1):1–42.

Koren, Y. (2010). Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(1):1–24.

Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37.

Mahesh, B. (2020). Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386.

Mnih, A. and Salakhutdinov, R. R. (2007). Probabilistic matrix factorization. *Advances in neural information processing systems*, 20.

Nikolopoulou, K. (2022). What Is Selection Bias? | Definition & Examples — scribbr.com. *https://www.scribbr.com/research-bias/selection-bias/*. [Accessed 05-May-2023].

Ning, X., Desrosiers, C., and Karypis, G. (2015). A comprehensive survey of neighborhood-based recommendation methods. In *Recommender systems handbook*, pages 37–76. Springer.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Portugal, I., Alencar, P., and Cowan, D. (2018). The use of machine learning algorithms in recommender systems: A systematic review. *Expert Systems with Applications*, 97:205–227.

Ricci, F., Rokach, L., and Shapira, B. (2015). Recommender systems: Introduction and challenges. In *Recommender systems handbook*, pages 1–34. Springer.

Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. (2000). Application of dimensionality reduction in recommender system-a case study. Technical report, Minnesota Univ Minneapolis Dept of Computer Science.

Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295.

Strömqvist, Z. (2018). Matrix factorization in recommender systems: How sensitive are matrix factorization models to sparsity?

Su, X. and Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009.

TensorFlow (2022). movielens | TensorFlow Datasets — tensorflow.org. *https://www.tensorflow.org/datasets/catalog/movielens*. [Accessed 22-May-2023].

Wang, B. (2022). Ranking evaluation metrics for recommender systems.

Xin, Y. et al. (2015). *Challenges in recommender systems: scalability, privacy, and structured recommendations*. PhD thesis, Massachusetts Institute of Technology.

Yalcin, E. and Bilge, A. (2022). Evaluating unfairness of popularity bias in recommender systems: A comprehensive user-centric analysis. *Information Processing & Management*, 59(6):103100.

Zahrawi, M. and Mohammad, A. (2021). Implementing recommender systems using machine learning and knowledge discovery tools. *Knowledge-Based Engineering and Sciences*, 2(2):44–53.

A. Figures

Figure A.1: Further analysis of the ml-100k dataset revealing the distribution of the number of users within each rating increment.

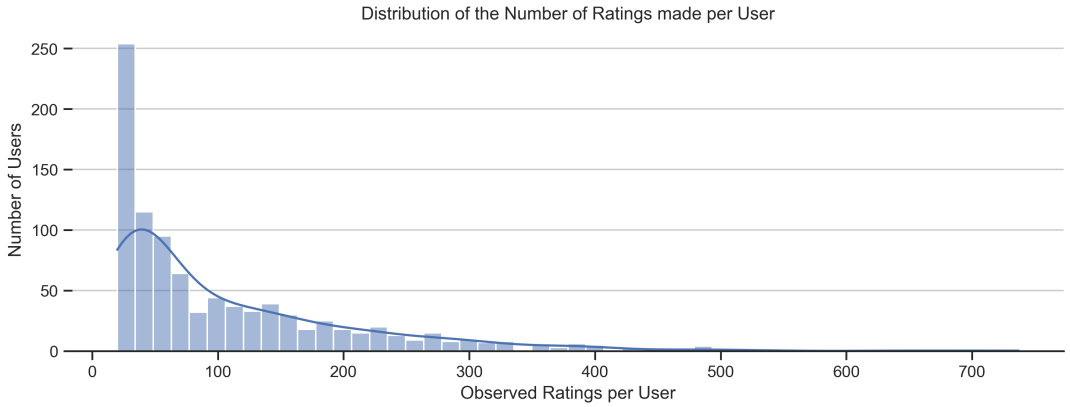


Figure A.2: Histogram with kernel density estimate line representing the distribution of user ratings.

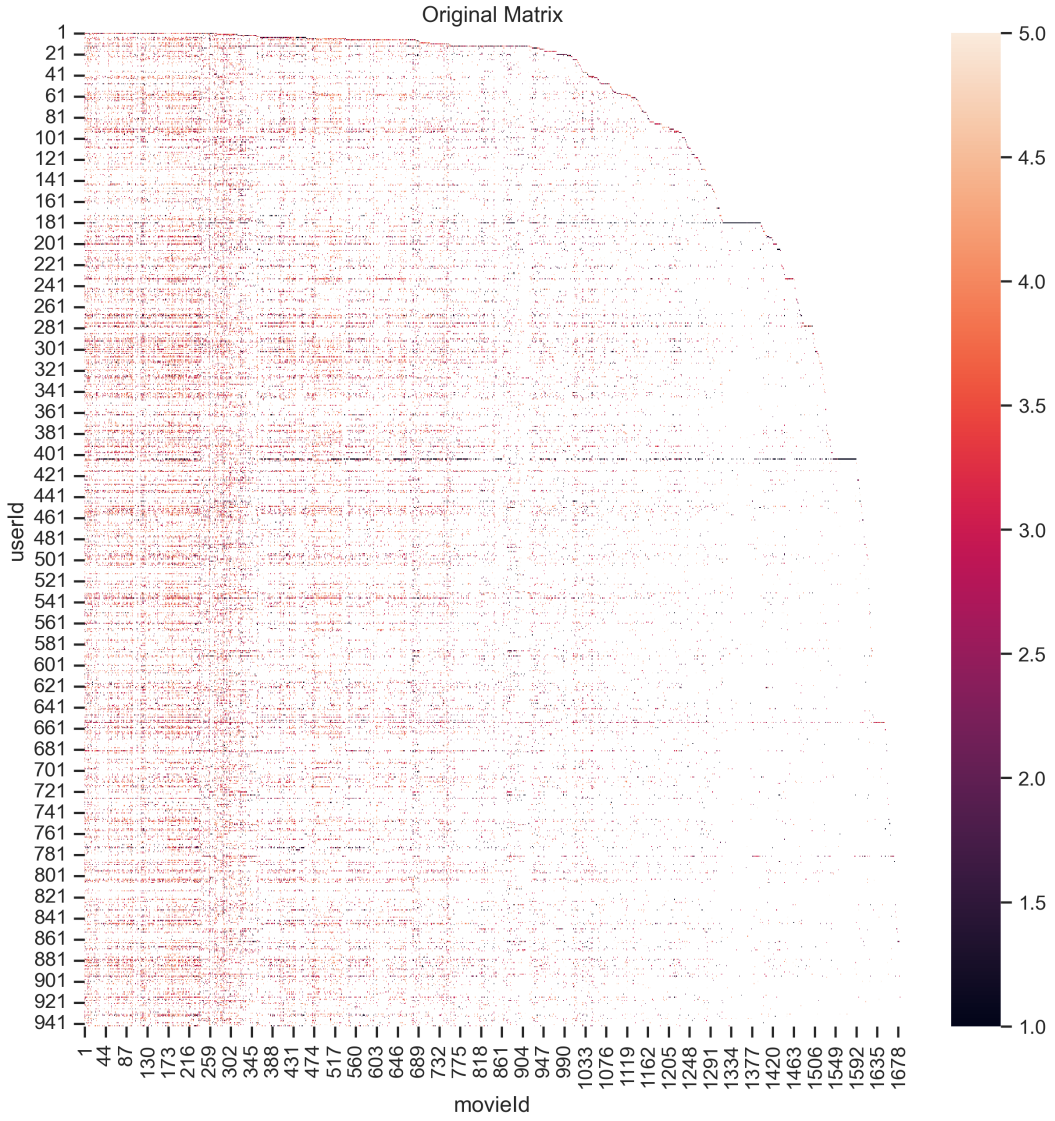


Figure A.3: Visualisation of the sparsity of a ratings matrix where the colour scale represents the rating value between 1 and 5.

B. Tables

Table 11: ml-100k *u.item* index file with encoded genres columns.

movieId	movie_name	year	url	5	6	7	8	9	...	15	16	17	18	19	20	21	22	23
1	Toy Story (1995)	01-Jan-1995	...	0	0	0	1	1	...	0	0	0	0	0	0	0	0	0
2	GoldenEye (1995)	01-Jan-1995	...	0	1	1	0	0	...	0	0	0	0	0	0	1	0	0
3	Four Rooms (1995)	01-Jan-1995	...	0	0	0	0	0	...	0	0	0	0	0	0	1	0	0
4	Get Shorty (1995)	01-Jan-1995	...	0	1	0	0	0	...	0	0	0	0	0	0	0	0	0
5	Copycat (1995)	01-Jan-1995	...	0	0	0	0	0	...	0	0	0	0	0	0	1	0	0

Table 12: Precision, recall and f1 score results from our experiments in Section 4.2.3.

list length	precision	recall	f1	list length	precision	recall	f1
1	0.708024	0.063947	0.117300	27	0.665162	0.306965	0.420072
3	0.687404	0.142024	0.235410	29	0.658432	0.305566	0.417417
5	0.685071	0.188486	0.295633	31	0.666521	0.310197	0.423363
7	0.673164	0.220404	0.332080	33	0.656980	0.308897	0.420217
9	0.674010	0.243893	0.358178	35	0.663852	0.310951	0.423523
11	0.668004	0.260199	0.374517	37	0.657100	0.312123	0.423218
13	0.665491	0.271472	0.385634	39	0.661027	0.313645	0.425431
15	0.670342	0.277465	0.392478	41	0.660686	0.319033	0.430288
17	0.670331	0.290324	0.405168	43	0.664211	0.315006	0.427342
19	0.663501	0.291384	0.404936	45	0.664235	0.314880	0.427231
21	0.668705	0.298197	0.412464	47	0.661226	0.316410	0.428009
23	0.666248	0.301945	0.415558	49	0.663281	0.315180	0.427310
25	0.667580	0.302608	0.416445				

Table 13: Useful and recommendation movie lists used to assess the `SVD` and kNN algorithm's coverage scores.

userId	actual	svd_recommendations	knn_recommendations
1	[1, 113, 114, 124, 135, 152, 156, 160, 163, 16...	[172, 114, 169, 209, 156, 262, 208, 79, 135, 1...	[169, 172, 114, 251, 9, 181, 185, 156, 198, 20...
10	[1, 100, 133, 160, 161, 174, 183, 186, 211, 21...	[513, 483, 285, 183, 498, 606, 529, 651, 707, ...	[285, 513, 174, 483, 100, 651, 498, 705, 432, ...
100	[272, 315, 344, 347, 690, 751, 898]	[272, 347, 315, 270, 751, 344, 333]	[272, 315, 690, 286, 270, 344, 348]
101	[117, 405, 50, 546, 866, 924]	[50, 7, 117, 845, 405, 1093]	[50, 117, 7, 1093, 111, 845]
102	[101, 195, 202, 307, 510, 98]	[313, 98, 79, 202, 82, 176]	[313, 195, 98, 176, 181, 154]

C. Code Listings

```
def user_based_prediction(ratings_matrix, userId, k,
    movie_predict):
    """
    :type ratings_matrix: pd.DataFrame()
    :type userId: int
    :type k: int
    :type movie_predict: str
    :rtype: int
    """
    userId = ratings_matrix.index.get_loc(userId) # index of
        user_i

    # remove users with no ratings of the movie
    ratings_matrix = pd.concat([ratings_matrix.iloc[[userId]],
        ratings_matrix[ratings_matrix[movie_predict].notna()]])

    # similarity matrix
    distance_matrix = ratings_matrix.T.corr(method='pearson')
    print(distance_matrix, end="\n\n")

    # similarity list for user_i
    user_weights = distance_matrix[userId].drop(userId)
    user_weights = user_weights.sort_values(ascending=False)
    print(user_weights, end="\n\n")

    # indices of the k most similar users
    most_similar_users = user_weights.iloc[:k].index
    print(most_similar_users, end="\n\n")

    _similar_user_ratings = []

    # obtain k most similar users ratings
    temp = ratings_matrix.loc[most_similar_users]
```

```
31    for index, row in temp.iterrows():
32        _similar_user_ratings.append(row[movie_predict])
33
34    return np.mean(_similar_user_ratings) # average
          neighbourhood rating
```

Listing 2: User-user filtering algorithm.

```
1  # matrix dimensions
2  number_of_movies = 100
3  number_of_users = 1000
4
5  # we use item indexes as opposed to regular strings to
       represent movies as it is easier for computation later
6  ratings_matrix =
       pd.DataFrame(np.random.randint(1,5,size=(number_of_users,
       number_of_movies)), columns=range(0,
       number_of_movies)).rename_axis('userId')
7
8  # randomly add specific percentage of missing values
9  percentage = 0.75
10 index = [(row, col) for row in range(ratings_matrix.shape[0])
       for col in range(ratings_matrix.shape[1])]
11 for row, col in random.sample(index, int(round(percentage *
       len(index)))):
12    ratings_matrix.iat[row, col] = np.nan
```

Listing 3: Initialises a 1000×100 ratings matrix with a controllable percentage of missing ratings.

```
1  def generate_test_users(n):
2      """
3      :type n: int
4      :rtype: pd.DataFrame()
5      """
6      test_users = ratings_matrix.sample(n) # randomly sampled
```

```
        test users
7
8     series = []
9     for index, row in test_users.iterrows():
10       while True:
11          item_rating = row.sample(axis=0) # get random rating
               value
12          if (item_rating.values == item_rating.values): #
               ensure rating is not missing
13             ratings_matrix.at[item_rating.name,
                  item_rating.keys()[0]] = np.nan # hide rating
14
15             s = pd.Series([item_rating.name,
                  item_rating.keys()[0], item_rating.values[0]])
16             series.append(s)
17             break
18
19    # dataframe representing test users
20    return pd.DataFrame(series).rename(columns={0: 'userId', 1:
         "itemId", 2: "actual"}).astype('int').set_index('userId')
```

Listing 4: Generates a set of test users from a ratings matrix by selecting a predetermined number of users, saving their ratings, and hiding them to then compute a prediction and measure the accuracy score.

```
1 # convert ratings matrix to list of ratings
2 df = ratings_matrix.reset_index().set_index('userId').stack()
3 df = df.reset_index()
4 df.columns = (["userId", "itemId", "rating"])
5
6 # convert dataframe to surprise object dataset
7 data = Dataset.load_from_df(df[["userId", "itemId",
     "rating"]], Reader(rating_scale=(1, 5)))
8 trainset = data.build_full_trainset() # full dataset
```

Listing 5: Creates a custom Surprise dataset object.

```
# initialise and train knn algorithm
sim_options = {
    "name": "pearson", # similarity method
    "user_based": True, # user-based or item-based
}
algo_user_based = KNNWithZScore(sim_options=sim_options)
algo_user_based.fit(trainset)

# generate predicted ratings
predicted_ratings = []
for index, row in testset.iterrows():
    pred = algo_user_based.estimate(row.name, row.itemId)[0]
    predicted_ratings.append(pred)

testset['predicted'] = predicted_ratings
```

Listing 6: Uses Surprise's kNN implementation to generate a set of predicted item ratings.

```
def read_item_names():
    file_name = get_dataset_dir() + "/ml-100k/ml-100k/u.item"

    rid_to_name = {}
    name_to_rid = {}
    with open(file_name, encoding="ISO-8859-1") as f:
        for line in f:
            line = line.split("|")
            rid_to_name[line[0]] = line[1]
            name_to_rid[line[1]] = line[0]

    return rid_to_name, name_to_rid
```

Listing 7: Return two mappings to convert raw ids into movie names and movie names into raw ids

```
def get_movie_neighbours(movie_name, k, algo):
    """
    :type movie_name: str
    :type k: int
    :type algo: surprise.KNNWithZScore
    :rtype: List[str]
    """
    # Read the mappings raw id <-> movie name
    rid_to_name, name_to_rid = read_item_names()

    # Retrieve inner id of the movie
    movie_raw_id = name_to_rid[movie_name]
    movie_inner_id = algo.trainset.to_inner_iid(movie_raw_id)

    # Retrieve inner ids of the nearest neighbors of Toy Story.
    neighbors = algo.get_neighbors(movie_inner_id, k=k-1)

    # Convert inner ids of the neighbors into names.
    neighbors = (
        algo.trainset.to_raw_iid(inner_id) for inner_id in
            neighbors
    )
    neighbors = (rid_to_name[rid] for rid in neighbors)

    k_neighbours = []
    for movie in neighbors:
        k_neighbours.append(movie)

    return k_neighbours
```

Listing 8: Obtains the k nearest neighbours of a specified movie title.

```
# read items file (movie properties)
movies = pd.read_csv('ml-100k/u.item', sep='|',
    encoding="ISO-8859-1", header=None)
movies.rename(columns={
```

```
    0:'movieId', 1:'movie_name', 2:'year', 3:np.nan, 4:'url'},
    inplace=True
)

# movies df with relevant genre column names
columns = ['movieId', 'name', 'year', None, 'url']
columns.extend(genres.array)
movies = pd.read_csv('ml-100k/u.item', sep='|', names=columns,
    encoding="ISO-8859-1", header=None).set_index('movieId')

genres_df = movies.iloc[:,4:].replace(1,
    pd.Series(movies.columns, movies.columns)) # replace 1 with
    genre column name

genres_arr = []

# columns representing genres are then converted to a single
    string for each row representing their genres seperated by
    commas
for index, row in genres_df.iterrows():
    genres_arr.append(", ".join(list(filter(None, row.values))))

genres_df = movies.iloc[:,0:1] # movieId, name
genres_df['genres'] = genres_arr
genres_df
```

Listing 9: Decodes the *u.item* index file columns representing the genres for each movie present within ml-100k.

```
"""Count the number of unique genres from a list of movie
    titles"""
def genre_count(k_neighbours):
    """
    :type k_neighbours: List[str]
    :rtype: pd.Series
    """
```

```
    # select movies and their corresponding genres (converted to array)
    arr = genres_df.loc[genres_df['name']
    arr = arr.isin(k_neighbours)]['genres'].values

    temp= []
    for i in range(len(arr)):
        temp.append(arr[i].split(", ")) # seperate strings consisting of multiple genres

    df = pd.DataFrame(list(chain.from_iterable(temp)), columns=['genres']) # dataframe with flattened 2D array

     # count of each genre
    return df.groupby('genres')['genres'].count().sort_values(axis=0, ascending=False)

k_val = [10, 20, 50, 100, 150, 200]

fig, ax = plt.subplots(figsize=(20,20))

for n, k in enumerate(k_val):
    # get top 10 genres listed for the movie
    k_neighbours = get_movie_neighbours('Toy Story (1995)', k, algo_item_based)
    temp = genre_count(k_neighbours).to_frame(name='count')
    temp = temp.reset_index().loc[:9]

    # plot genre distribution
    ax = plt.subplot(3, 3, n+1)

    largest_genre = temp.loc[temp['count'].idxmax()]['genres']

    plt.pie(data=temp, x='count', labels=temp['genres'], autopct='%.0f%%',
```

```
36          explode=(temp['genres'] == largest_genre).values *
               0.05,
37          textprops={'fontsize': 13}
38         )
39    ax.set_title(' neighbours : ' + str(k), fontsize=20)
```

Listing 10: Performs a count for the genres present within the neighbours of the movie *Toy Story*.

```
1  n_useful_items = 10 # number of useful items (arbitrary value)
2
3  # group together items corresponding to each unique userId
4  df_agg =
       results.groupby(['userId','movieId']).agg({'actual':sum})
5  temp = df_agg['actual'].groupby('userId', group_keys=False)
6
7  # obtain highest rated items for each user
8  df = pd.DataFrame(temp.nlargest(n_useful_items))
9
10
11 # for each user, convert grouped items to list of item id's
12 arr = []
13
14 for X, new_df in df.groupby(level=0):
15    temp = []
16    for i in range(len(new_df.index.values)):
17       temp.append(new_df.index.values[i][1])
18
19    arr.append(temp)
20
21 # userId as dataframe index
22 useful_items =
       pd.DataFrame(df.index.get_level_values(0).unique())
23
24 # initialise column
25 useful_items['actual'] = arr
```

```
useful_items = useful_items.set_index('userId')
useful_items.head()
```

Listing 11: Appends a list of the 10 highest-rated movie indices for each test user within a dataframe.

```
def get_users_predictions(user_id, model):
    """
    :type user_id: str
    :type model: int
    :rtype: List[str]
    """
    recommended_items = pd.DataFrame(model.loc[user_id])
    recommended_items.columns = ["predicted_rating"]
    recommended_items =
        recommended_items.sort_values('predicted_rating',
        ascending=False)
    return recommended_items.index.tolist()

ratings_matrix = results.pivot(index='userId',
    columns='movieId', values='predicted').fillna(0)
cf_recs = [] = []

for user in useful_items.index:
    actual_length =
        len(useful_items['actual'].loc[useful_items.index ==
        user][0])

    # recommendations of size equal to actual list
    cf_predictions = get_users_predictions(user,
        ratings_matrix)[0:actual_length]
    cf_recs.append(cf_predictions)

useful_items['recommendations'] = cf_recs
useful_items
```

Listing 12: Refactored function from the recmetrics library which obtains the indices of the highest-rated movies based on their predicted ratings from our test set.

```
results = []

for k_val in range(1, 50, 2):
    p = []
    r = []

    for trainset, testset in KFold(n_splits=3).split(data):
        algo.fit(trainset)
        predictions = algo.test(testset)
        precisions, recalls = precision_recall_at_k(predictions,
            k=k_val, threshold=4)

        # Precision and recall can then be averaged over all
            users
        p.append(sum(prec for prec in precisions.values()) /
            len(precisions))
        r.append(sum(rec for rec in recalls.values()) /
            len(recalls))

    results.append([np.mean(p), np.mean(r)])
    print(f'[Precision, Recall] at Recommendation List Length
        {k_val}: {[np.mean(p), np.mean(r)]}')
```

Listing 13: Calculates the precision and recall evaluation metrics at various recommendation list lengths.

```
# svd training and testing
svd_algo = SVD()
svd_predictions = svd_algo.fit(trainset).test(testset)

svd_results =
```

```
    pd.DataFrame(svd_predictions).drop(columns=['details'])
svd_results.columns=['userId', 'movieId', 'actual',
    'predicted']
svd_results.head()

ratings_matrix = svd_results.pivot(index='userId',
    columns='movieId', values='predicted').fillna(0)
svd_recs = [] = []

# compute index list
for user in useful_items.index:
    actual_length =
        len(useful_items['actual'].loc[useful_items.index ==
        user][0])

    # recommendations of size equal to actual list
    svd_predictions = get_users_predictions(user,
        ratings_matrix)[0:actual_length]
    svd_recs.append(svd_predictions)

useful_items['svd_recommendations'] = svd_recs
useful_items
```

Listing 14: Computes list of recommended movies using the `SVD` model and the previously defined `get_users_predictions()` function to then assess the model's coverage score.