# Movie Recommendation Systems: An Overview and Analysis

Laurence Stephan

September 2024

## Contents

**1 Introduction** **2**

**2 Exploring the data** **3**
2.1 Data Overview . . . 3
2.2 Exploring The Data . . . 4

**3 Data Preprocessing** **13**
3.1 Matrix Transformation . . . 13
3.2 Dimension Reduction . . . 15
3.3 Relevant Data . . . 16

**4 Models and Results** **17**
4.1 Linear Regression . . . 17
4.2 Recommender Engines . . . 23
4.3 Slope One . . . 24
4.4 Matrix Factorisation . . . 25
4.5 Ensemble Methods . . . 28
4.6 Neural Networks . . . 30

**5 Making Predictions** **34**

**6 Conclusion** **35**

**7 Recommendations** **35**

# 1 Introduction

A recommendation system employs machine learning algorithms to analyse past user data, with the aim of predicting a user's preferences for future or unobserved data. Within the context of movie recommendations, the system utilises user rating, and movie rating data to generate predictions for unseen movies or users. The primary categories of ML algorithms commonly applied in movie recommendations are content-based filtering and collaborative filtering systems.

Content-based filtering recommends similar content to a user, drawing from their past interactions. Collaborative filtering recommends content highly rated by users with comparable profiles, relying on similar past ratings for overlapping movies. Additionally, collaborative filtering algorithms are further categorised into two distinct types: User-based collaborative filtering, which seeks analogous patterns in movie preferences between the target user and others within the database, and Item-based collaborative filtering, which revolves around identifying similar items (in this case movies) that the target users have rated or interacted with.

Developing a movie recommendation system follows several critical stages. The process begins with acquiring relevant data, which can be sourced from various online platforms—either freely available, scraped from the web, or purchased from third-party providers. For this report, I utilised the MovieLens 10M and 100k data sets from GroupLens.org. Once the data was acquired, it underwent comprehensive analysis to extract meaningful insights and to understand how best to manipulate and structure the data for further refinement.

With a focus on collaborative filtering models, multiple models were trained, evaluated, and compared based on key metrics, including margin of error, training time, and storage efficiency. The best-performing model was selected for further optimisation, aiming to achieve the highest accuracy with the least resource consumption. Finally, the optimised model was used to predict a randomly selected user's rating preferences for unseen movies.

Tailoring content to match users' preferences, interests, and behaviours has become crucial for businesses seeking to offer superior service. By delivering personalised recommendations, businesses can enhance engagement and drive greater value, often outpacing their competitors. Machine learning algorithms play a key role in systems that use data to optimise their offerings. Leading companies such as Spotify, Amazon, Disney+, and Netflix employ such algorithms to provide personalised suggestions for their users. In 2006, Netflix famously launched a competition, offering a $1 million prize to the team that could most significantly improve the accuracy of their recommendation system. This competition elevated the field's profile, leading to the development and popularisation of many new algorithms in Data Science.

This report compares a variety of recommendation models, using 10 million ratings from the MovieLens data set provided by the GroupLens research lab. Notably, a neural network with a simplified dot product model and a matrix factorisation approach using stochastic gradient descent performed best, as measured by Root Mean Squared Error (RMSE) on test and training sets. However, Alternating Least Squares and linear regression models also delivered competitive results, while requiring fewer resources.

Broadly speaking, recommendation systems face two primary challenges. The first is known as the ranking problem, which involves ranking and recommending items based on known user data, such as past interactions. For instance, factors like actors, release dates, and genres that a user has previously rated favourably are used to generate recommendations. The second challenge, and the focus of this report, is the matrix completion problem. This challenge involves predicting missing data points within a matrix by analysing rating patterns across observed data. For example, if a group of users have historically shown similar rating patterns for a subset of movies they've watched, it's reasonable to infer that if the group rates a movie that one user hasn't yet seen, their ratings would likely be similar. By leveraging this insight, we can recommend highly-rated movies from users with comparable profiles to those who haven't rated the movie yet, anticipating a similar rating and potential enjoyment.

# 2 Exploring the data

## 2.1 Data Overview

```
Load libraries and data
```

```
 "training data"
```

```
Rows: 9,000,061
Columns: 6
$ userId    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, ~
$ movieId   <int> 122, 185, 231, 292, 316, 329, 355, 356, 362, 364, 370, 377, ~
$ rating    <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 5, ~
$ timestamp <int> 838985046, 838983525, 838983392, 838983421, 838983392, 83898~
$ title     <chr> "Boomerang (1992)", "Net, The (1995)", "Dumb & Dumber (1994)~
$ genres    <chr> "Comedy|Romance", "Action|Crime|Thriller", "Comedy", "Action~
```

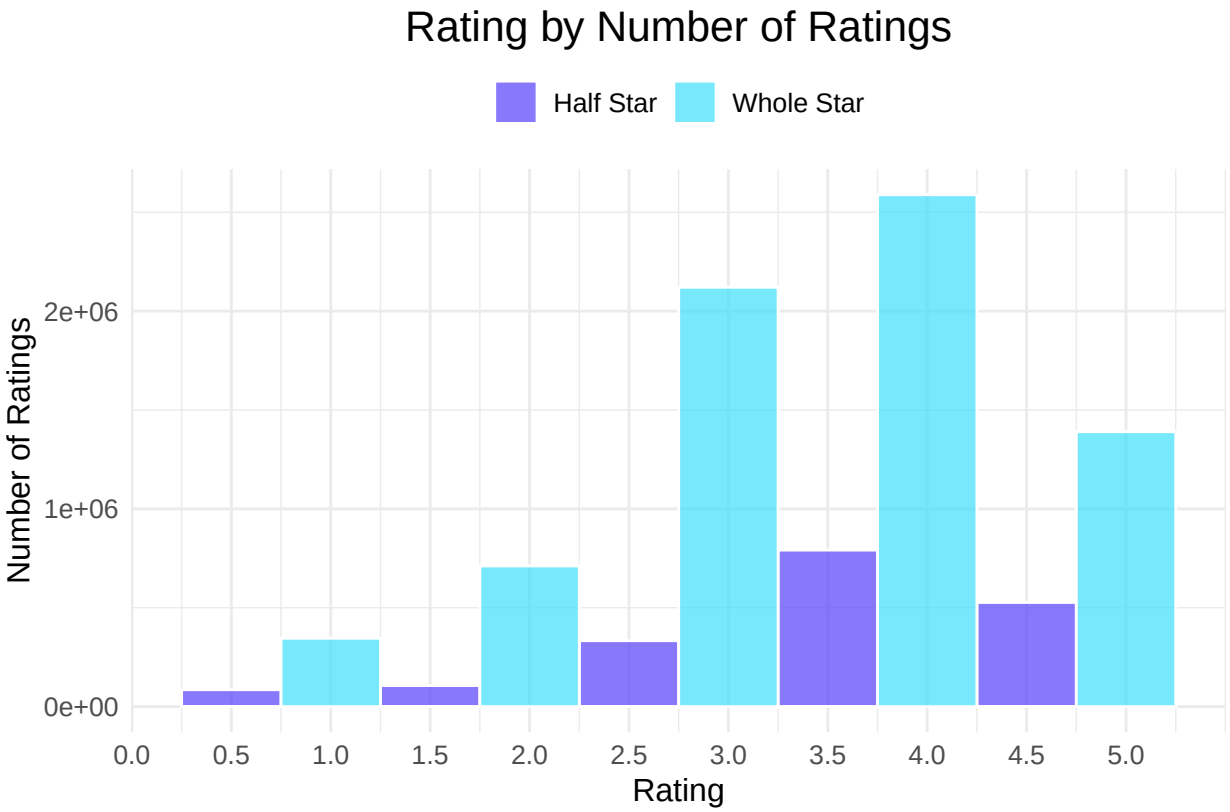
```
 "testing data"
```

```
Rows: 999,993
Columns: 6
$ userId    <int> 1, 1, 1, 1, 1, 3, 4, 4, 4, 4, 5, 5, 5, 5, 6, 7, 7, 7, 7, 7, ~
$ movieId   <int> 466, 480, 539, 589, 594, 5527, 110, 432, 434, 592, 52, 778, ~
$ rating    <dbl> 5.0, 5.0, 5.0, 5.0, 5.0, 4.5, 5.0, 3.0, 3.0, 5.0, 4.0, 4.0, ~
$ timestamp <int> 838984679, 838983653, 838984068, 838983778, 838984679, 11648~
$ title     <chr> "Hot Shots! Part Deux (1993)", "Jurassic Park (1993)", "Slee~
$ genres    <chr> "Action|Comedy|War", "Action|Adventure|Sci-Fi|Thriller", "Co~
```

The training set has 9,000,061 rows, and contains approximately 90% of the data. The testing set has 999,993 rows and contains approximately 10% of the data. Each set has 6 variables, represented by each column. The userID variable represents a unique number assigned to a specific user profile. The movieId variable represents a unique number assigned to a specific movie. The rating variable represents a score about a movie given by a user, on a rating scale of 1-5, where 1 represents a low score and 5 represents a high score. The timestamp variable is a record of the date and time a rating for a movie was submitted by a user. The title variable represents the name the movie was released under in North America, and the genre variable represents the genre of the movie as defined by the MovieLens community. Although the same unique variables may show up across multiple rows, there will only ever be one row with the same userId and movieId, representing a distinct rating for that movie by that user. The userID, movieID, timestamp and rating variables are discrete, whereas the title and genre variables are nominal. The models developed and tested in this report will focus on predicting rating as a dependent variable, with userId and movieId as the independent variables, which is typical of a collaborative filtering system. Timestamp, title and genre will not be considered when building the predictive models; however, they are datatypes that prove useful for other types of models such as content-based filtering systems, and will therefore be discussed and explored at a high level.

## 2.2 Exploring The Data

An initial investigation of each variable was performed to gain a strong understanding of the data.




A visual representation of the rating data shows that full star ratings are much more common than half star ratings. It also shows that people tend to leave higher ratings over lower ones, with the mode being 4 stars. Such findings suggest that people may be biased towards choosing whole numbers over fractions, and more likely to express positive feelings over negative ones. While this will have a small effect on a recommendation systems accuracy, it does raise the question whether rating scales that use fractions, or rating scales in general serve as a good estimators for accurately representing ratings.

Top Genres Summary

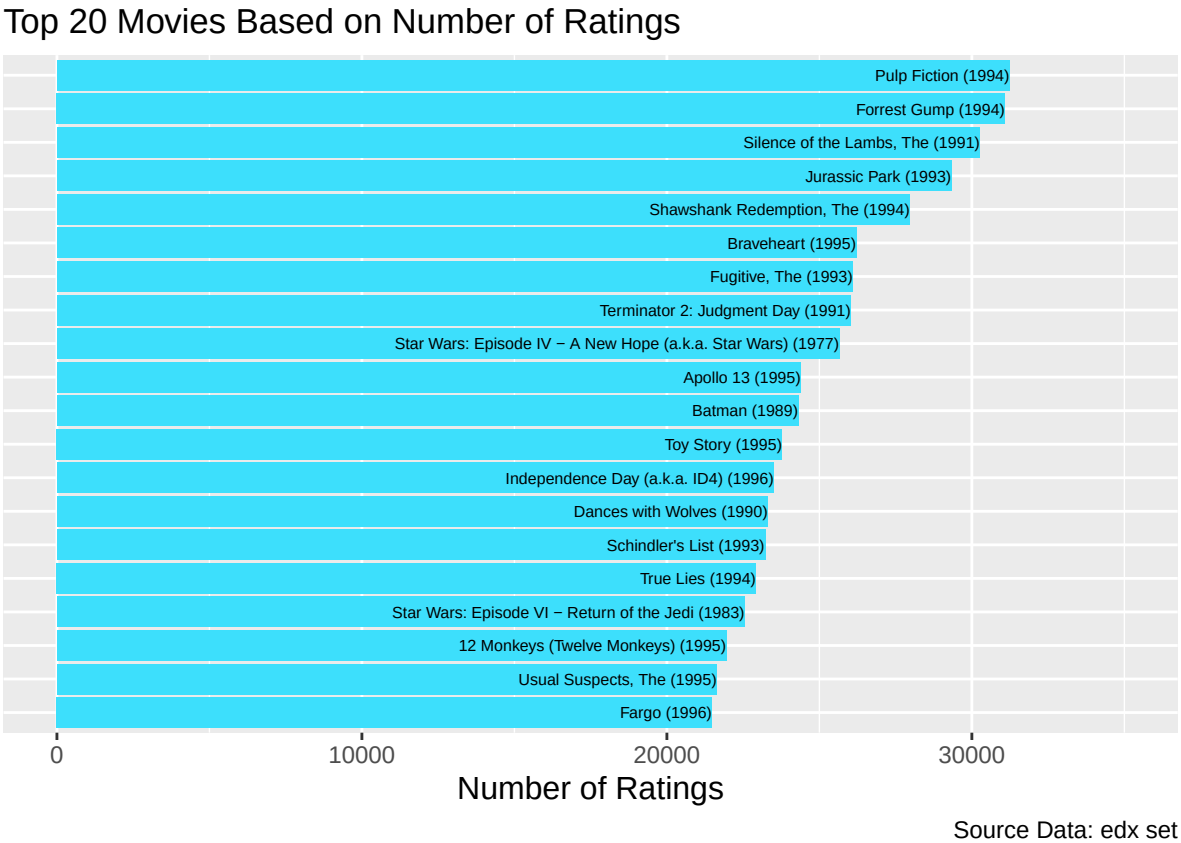
| genres | count |
|---|---|
| Drama | 3909769 |
| Comedy | 3540190 |
| Action | 2561318 |
| Thriller | 2325639 |
| Adventure | 1909695 |

Top Years Summary

| year | count |
|---|---|
| 1995 | 787124 |
| 1994 | 671688 |
| 1996 | 593262 |
| 1999 | 489917 |
| 1993 | 481311 |

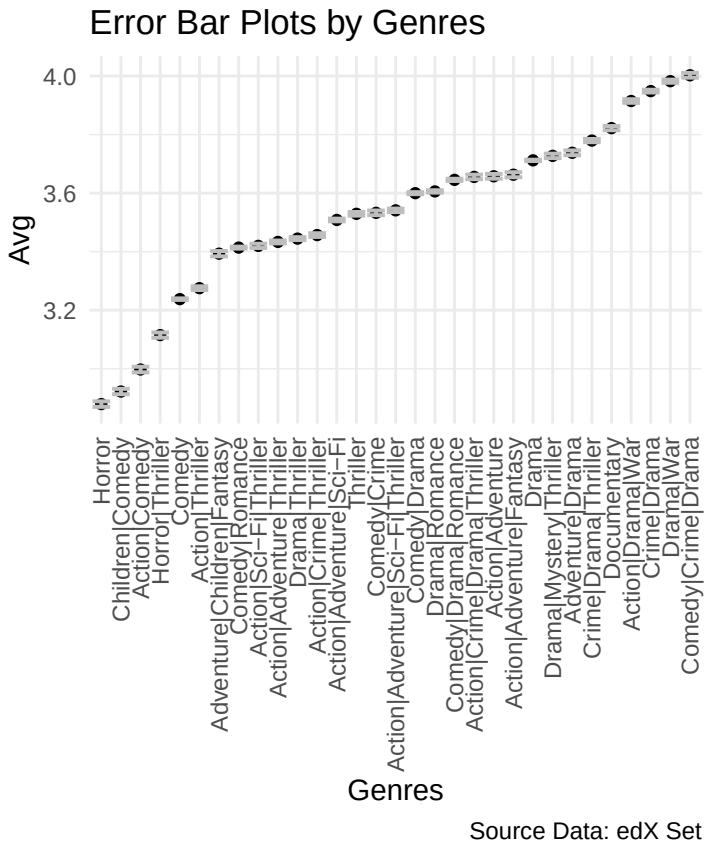
| Top Titles Summary | |
|---|---|
| title | count |
| Pulp Fiction | 31234 |
| Forrest Gump | 31084 |
| Silence of the Lambs, The | 30263 |
| Jurassic Park | 29325 |
| Shawshank Redemption, The | 27964 |

The tables group the data by genre, year, and title, and then order them in descending order based on the amount of ratings they've received. This should act as a good indicator of how many people have watched a movie with a specific title, released in a specific year, or released within a specific genre. Understanding which variables receive the most ratings gives us a heading for making decisions about the future. However, a word of caution should be added: In order to make the data more meaningful, the ratings count should be weighted by website traffic, the volume of movies released in a year, and the amount of time the movie has been available to receive ratings. This would account for a more accurate indicator of how many people watched a movie because not everyone will use MovieLens to rate a movie, meaning that the amount of ratings a movie receives might be a direct result of how popular MovieLens is at the time, compared to the movie itself. Moreover, the longer a movie has been online, the more time it's had to accrue ratings, and the amount of movies released in a year may create a bias that makes that year seem more popular, meaning the amount of ratings should be scaled relative to both the popularity of MovieLens at the time, the volumes of movies released in a year, and the amount of time the movie has been open to ratings. This analysis is beyond the scope of this project, but could serve as an insightful avenue for future research.

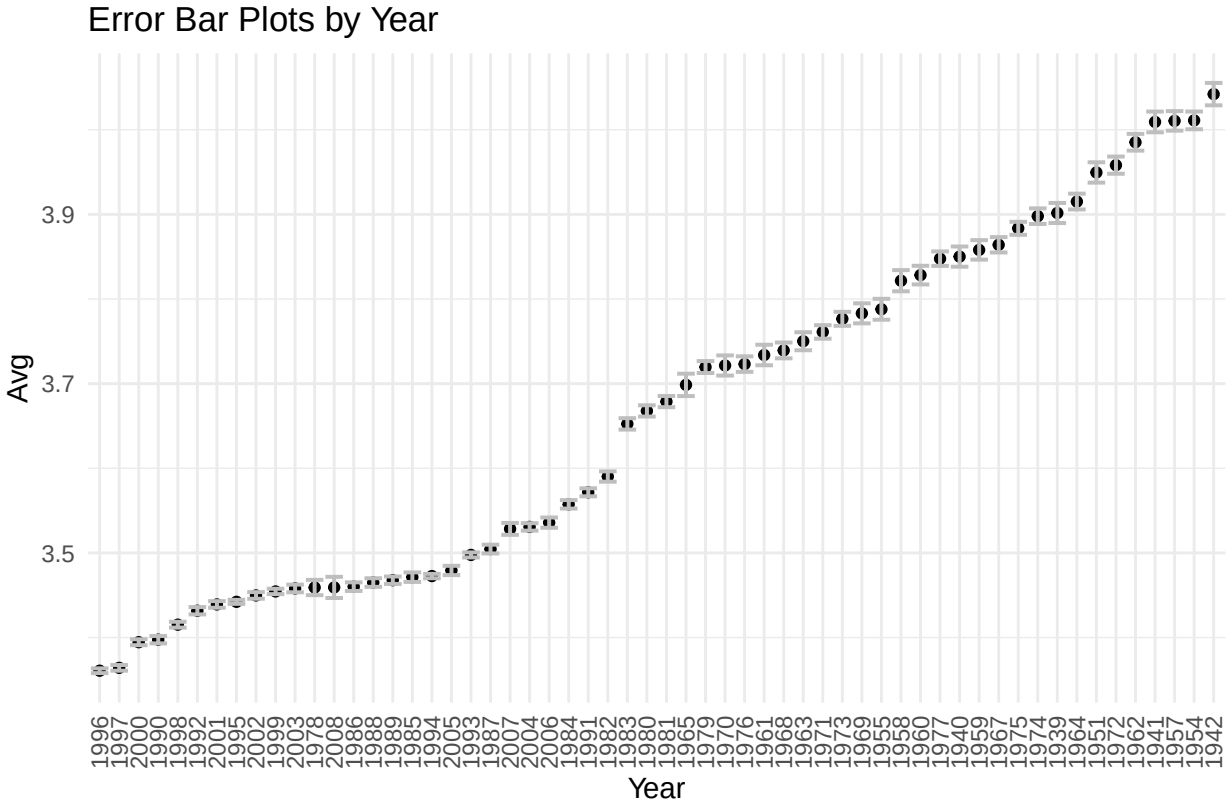



A visual representation of the titles, with their respective release years, shows a trend towards more ratings for movies released in the 90s. Furthermore, the majority of these movies fall into the most rated genres,

suggesting that a genre's popularity might be a result of a small number of heavily rated movies, as opposed to the genre being inherently more popular itself.




The bar plot illustrates the significant impact of genre on a movie's rating. Additionally, it indicates that a movie's genre is not correlated with the number of ratings it receives. Despite being comparatively poorly rated, comedy genre movies rank second in total ratings. Hence, for a platform seeking to maximise its ratings, the quality of a movie's rating holds little relevance. Furthermore, if we assume that a high number of ratings correlates with a large viewership, then a movie's rating score does not strongly influence its popularity.

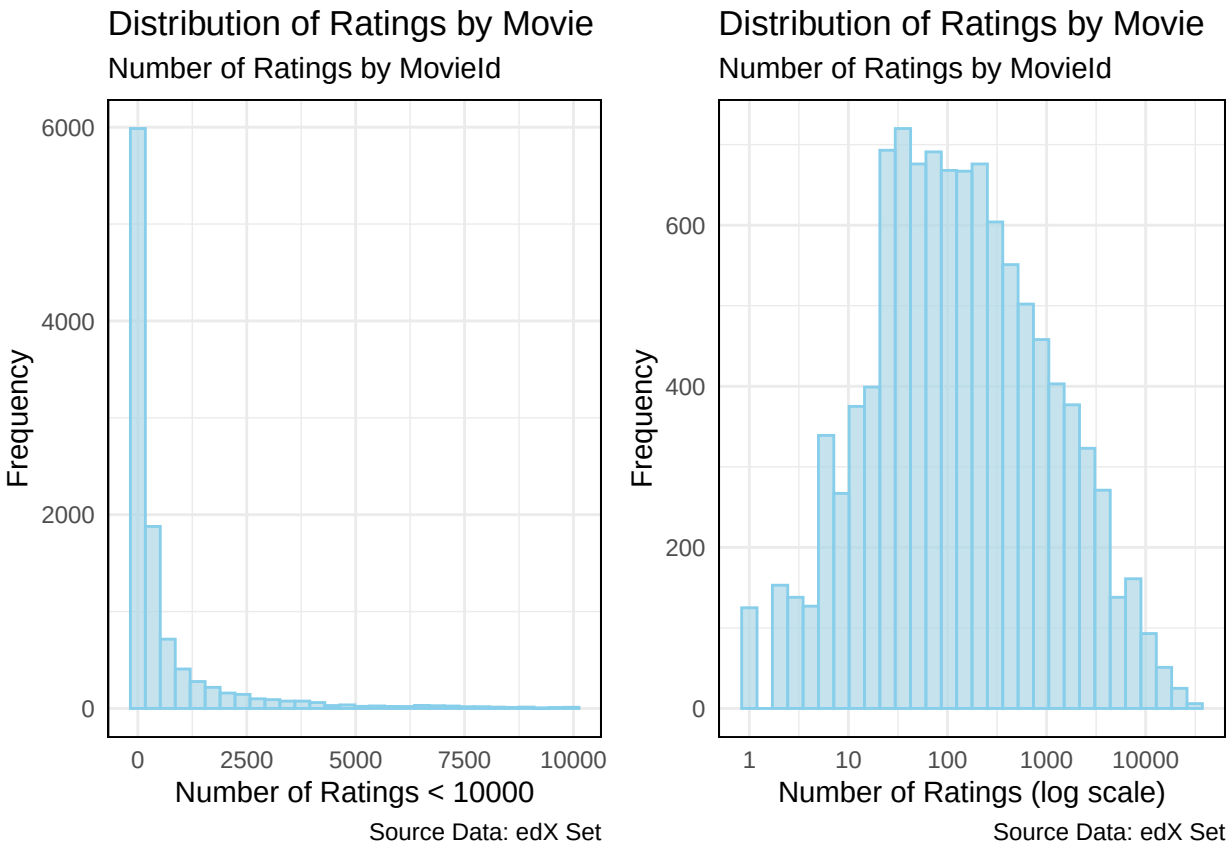



Source Data: edX Set

The release year of a movie also influences its rating. There is a noticeable trend showing that movies released before the 1980's tend to receive higher ratings. However, these older movies generally receive fewer ratings overall and exhibit slightly higher variability. To comprehend the underlying reasons for this trend, it is necessary to consider a broad spectrum of socioeconomic factors, including changes in cinema culture, economic conditions, the volume of film production, and the size of the industries market. With that being said, it seems reasonable to suggest that lower production volumes and higher barriers to entry create an environment conducive to higher average quality.

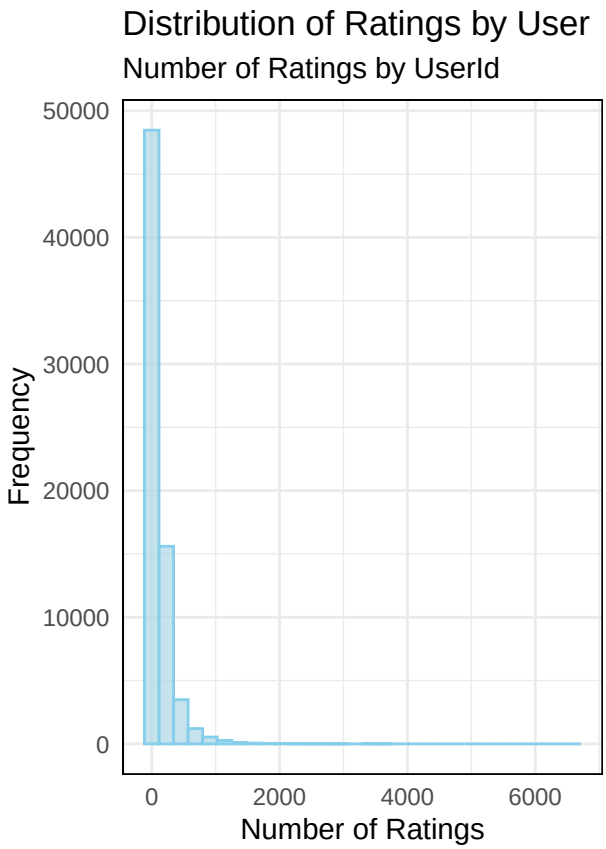
```
  n_users n_movies
1   69878    10677
```

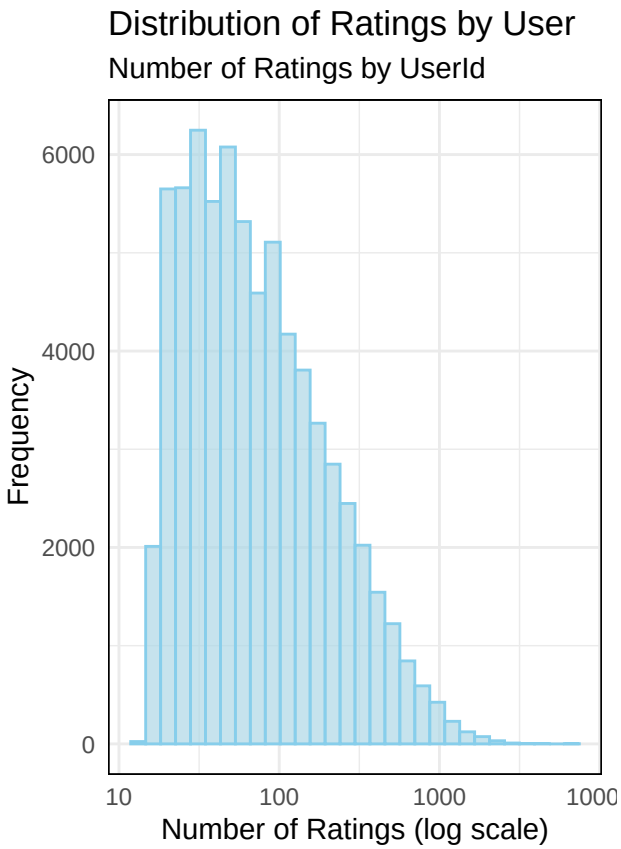
Next, I analysed the number of users who provided movie ratings (n_users) alongside the number of movies that received ratings (n_movies). Notably, the n_users figure is 6.5 times greater than n_movies, suggesting that some users have rated multiple movies. However, rating volumes are rarely uniform; it is common for a small group of "superusers" to contribute a disproportionate number of ratings, skewing averages and ratios. Furthermore, when exploring the data set for predictive purposes using matrices and multidimensional linear regression, it became evident that the data set likely exhibited significant sparsity.




The first histogram clearly shows a highly right skewed distribution. Most movies have a very low number of ratings, as evidenced by the tall bar near zero, with the frequency rapidly declining as the number of ratings increases. This aligns with the observation that the majority of movies receive very few ratings, while only a small subset of movies are rated frequently.

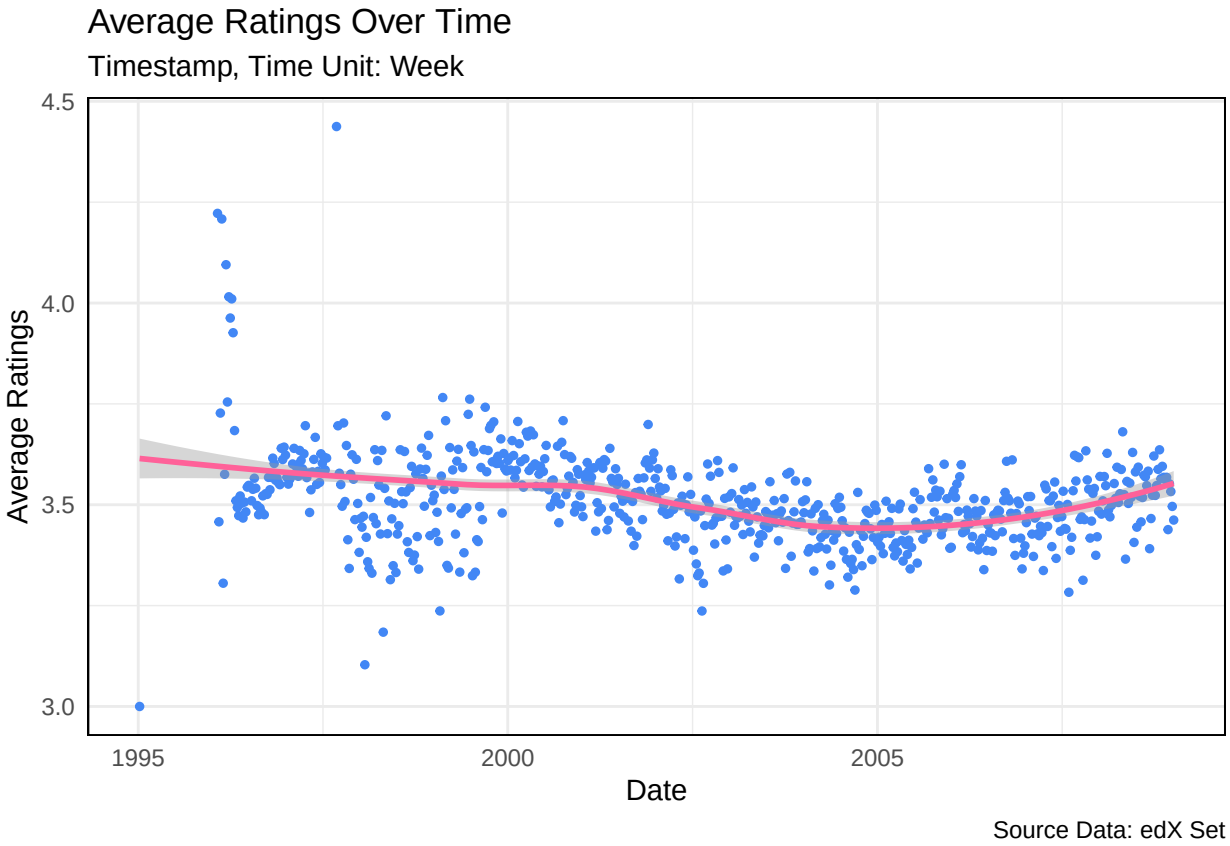
The second histogram, plotted on a logarithmic scale, provides a clearer picture of the spread across orders of magnitude. While the skewness is less apparent due to the log transformation, it still demonstrates that there are significantly more movies with low ratings than those with high ratings. This supports the conclusion about the rarity of movies with many ratings and further emphasises the sparsity issue inherent in the data set.



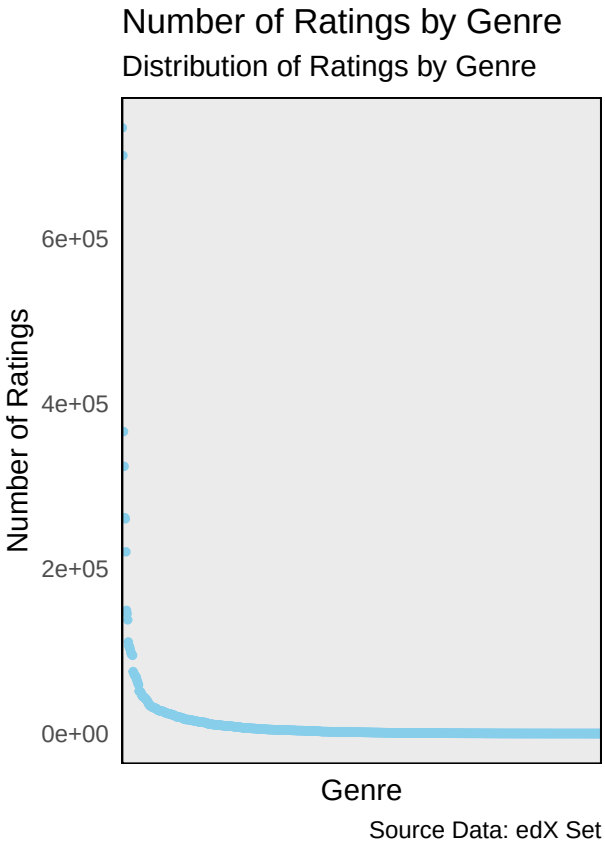

The analysis of the number of ratings per user reveals a distribution that is heavily skewed to the right, characterised by a rapid decrease in frequency as the number of ratings increases. This pattern suggests that while most users give relatively few ratings, with a select cohort of users that provide a significant number of ratings, which is unsurprising given the relative ease of creating new user profiles compared to consistently adding new movies to the database.

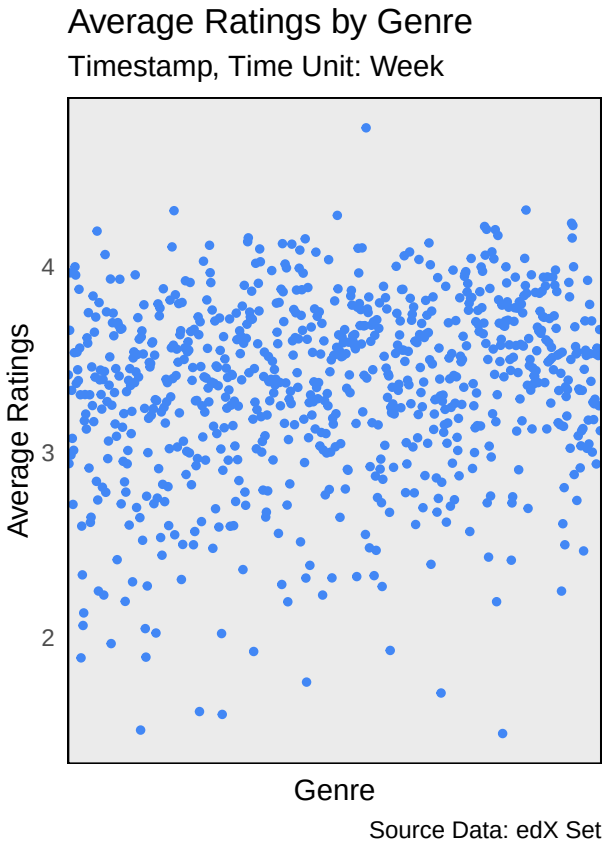
This observation further highlights the critical importance of addressing scarcity in predictive modeling. It prompts us to confront challenges such as accurately predicting user preferences when faced with limited data or when dealing with users whose interests span a wide spectrum, presenting a lack of comparable nearest neighbours for reliable predictions.



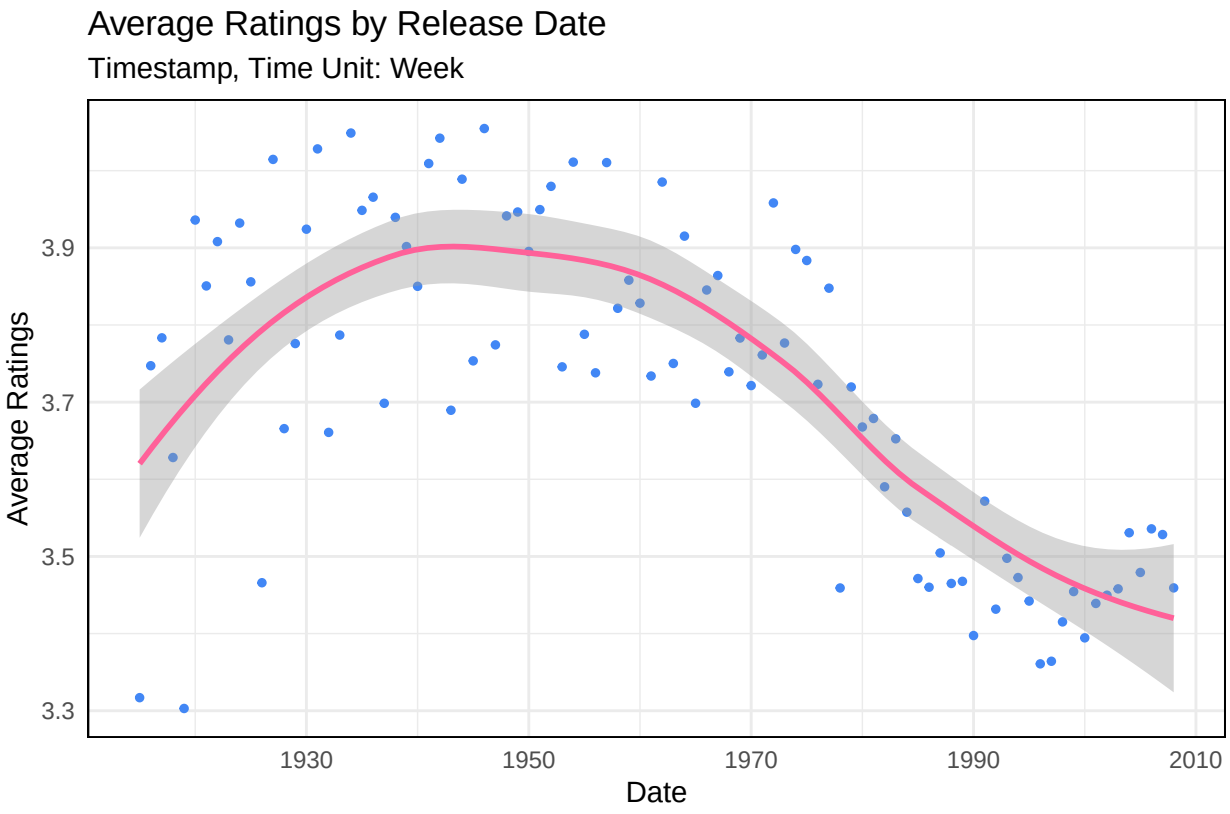

Analysing average ratings over time reveals a notable trend: in the early years of the internet, there was considerably higher variability, likely attributable to the smaller user base. However, as time has progressed, this variability has decreased, and the distribution of ratings has to normalised around the true mean.

Moreover, examining the trend line illustrates that over time, people's average ratings experience minimal change, fluctuating within a narrow range of 3.4 to 3.7. This stability suggests a consistent pattern of user behaviour, indicating a certain level of reliability in the rating system over the years. Meaning that the data should act as a reliable estimator when predicting future movie ratings.




Upon closer examination of the genre variable, it becomes evident that there is significant variability in both the number of ratings each genre accumulates and the average rating within each genre. This variability suggests that genre could strongly influence both the popularity and the rating of a movie. While a typical time series trend may not be calculable in this context, trends relating to genre popularity may emerge through further analysis, such as matrix factorisation, which goes beyond the standard categorisation measures




Analysis of average ratings according to movie release dates reveals a notable pattern: movies released between the 1930s and 1970s tend to receive higher ratings, with a steady decline observed since the 1940s. Consequently, it becomes imperative to consider normalisation of movie release years to ensure precision in recommendations, particularly for newly released films.

# 3 Data Preprocessing

Data sets created from data observed in the real world typically need processing before they can be utilised. Tasks such as cleansing, filtering, and modification make the data suitable for mathematical and machine learning models. Within this section, the emphasis will lie on data preprocessing techniques crucial for the development of recommendation system. It's important to recognise that no data preprocessing stage is flawless or exhaustive. The preprocessing process is inherently iterative and subject to evolution over time, influenced by the nature of the data at hand, advancements in mathematical research, shifting assumptions, and evolving standards.

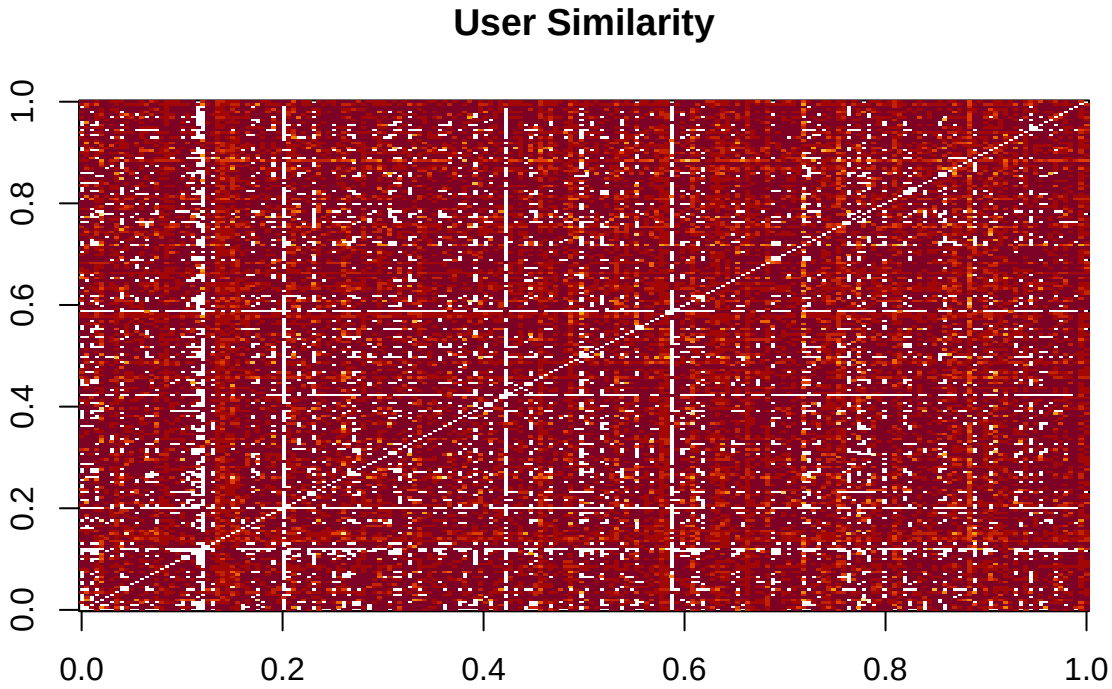
## 3.1 Matrix Transformation

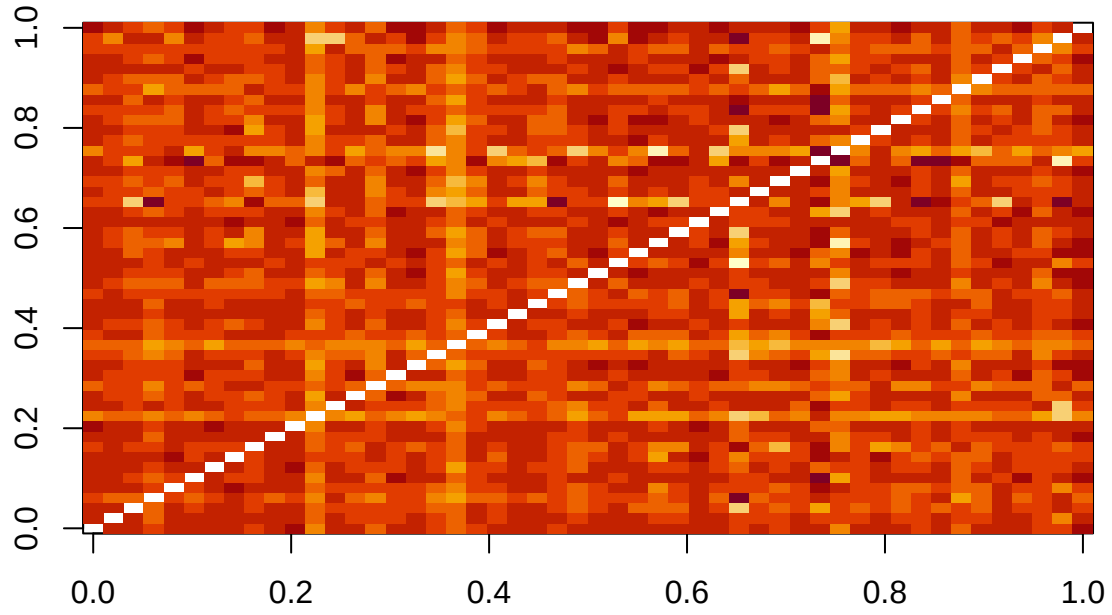
First I created a ratings matrix to start to help us reduce the sparsity problem.

```
10 x 10 sparse Matrix of class "dgCMatrix"

 [1,] . .   . . . . . . . .
 [2,] . .   . . . . . . . .
 [3,] . .   . . . . . . . .
 [4,] . .   . . . . . . . .
 [5,] 1 .   . . . . 3 . . .
 [6,] . .   . . . . . . . .
 [7,] . .   . . . . . . . .
 [8,] . 2.5 . . 3 4 . . . .
 [9,] . .   . . . . . . . .
[10,] . .   . 3 . . 3 . . .

69878 x 10677 rating matrix of class 'realRatingMatrix' with 9000061 ratings.
```

When converting datasets into matrices, numerous packages are available. However, various factors must be taken into account, including data sparsity, scarcity, compute power, memory, complexity, and runtime. Particularly for large datasets with high levels of sparsity, the Matrix.utils package provides a suitable solution with its function, sparseMatrix, which handles the data well.

During the data mining process of building a recommendation system, I needed to identify patterns that allowed me to calculate how similar people's preferences are to one another. Typically, similar preferences can be thought of in terms of distances in a multidimensional Euclidean space. One particular measure of distance that has proven efficient is the Cosine Similarity, which measures the cosine of the angle between two different vectors of an inner product space. Essentially, it aims to measure whether two vectors are pointing in the same direction. The benefits of this technique are its consistency, its adequate handling of the problems of sparsity and scarcity, and its scalability with large datasets, making it suitable for real-world applications where processing large amounts of data is necessary.




For visualisation purposes, the two matrices above were generated using a subset of the first 200 users to reduce computational load. Each row represents a user, and each column represents a movie. Consequently, each cell contains user ratings for movies, with similarity indicated by colour. Darker cells denote greater similarity between adjacent users.

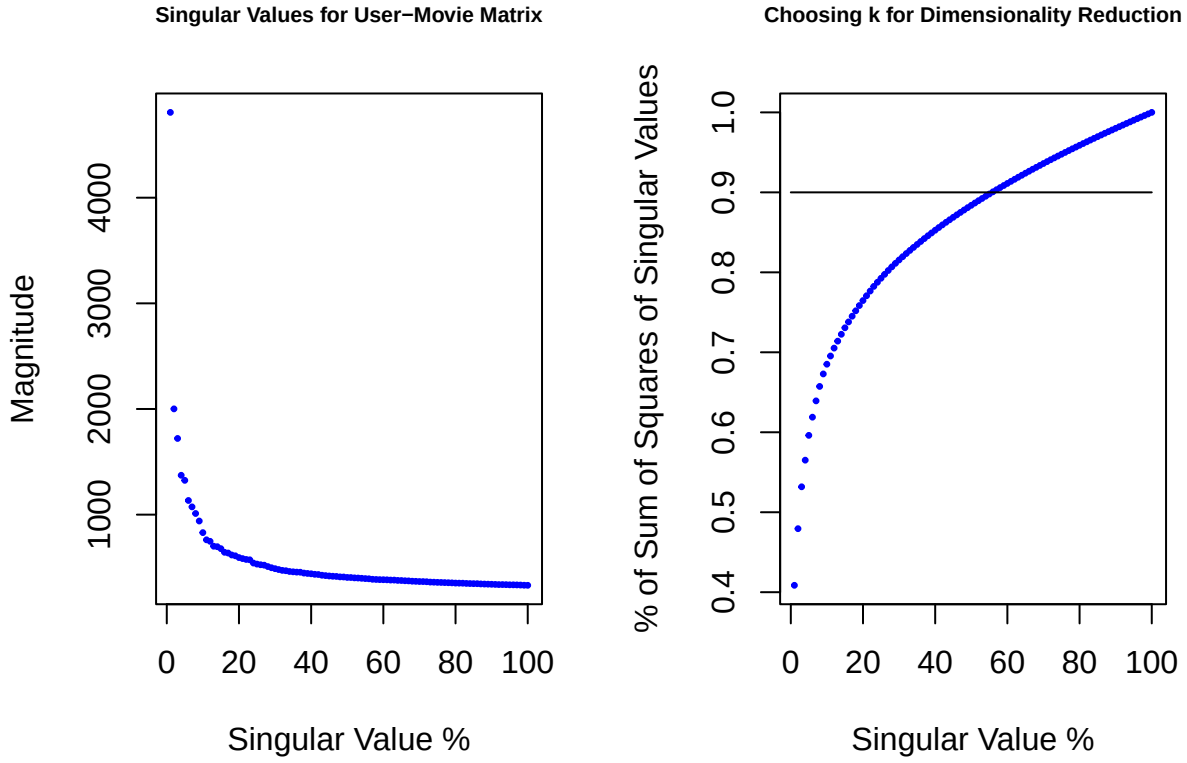
Furthermore, a significant implication of this analysis is the discovery of uncharted categories suggested by groups of similar user and movie ratings. These categories may defy traditional genres or preferences, revealing previously unrecognised patterns in human behaviour. This approach to identifying patterns offers a more organic method of defining groups in various contexts, although caution is warranted regarding potential circular reasoning. Organic groups may have formed over time due to the use of pre-existing grouping methods, causing expectation bias.

Nevertheless, uncovering novel groups and patterns yields transformative insights for industry, playing a pivotal role in prediction and recommendation systems.

### 3.2 Dimension Reduction

In the next stage of my data processing, I need to tighten up the sparsity of the data. I can do this using techniques that reduce the dimentionality of the Euclidean spaces. I can use Principal Component Analysis (PCA) to reduce the linear dimensionality of the data, and Singular Value Decomposition to factorise, rotate and rescale the data in order to extrapolate some of the more important patterns, with lower dimentionality.

As I am running these computations locally, I will use the Iterative randomised Lanczos Bidiagonalization Algorithm (IRLBA) package, which has been optimised towards low computational load by using the randomised algorithms based on the Lanczos method (a method that can quickly and accurately find the eigenvalues and eigenvectors of a large, sparse, symmetric matrix) to approximate the most important single values and vectors.




Plotting a cumulative sum of squares for the singular values reveals that the 90% threshold is achieved somewhere between 50% and 60% of the singular values. The first 6% percent of the singular values of the imputed ratings matrix explain over half of the variability, with nearly 70% of the variability explained by the first 12%, and over 75% by the first 20%. My goal is to identify the percentage of 'k' singular values whose squared sum accounts for at least 90% of the total sum of squares, as this allows us to capture the vast majority of variability while keeping the dimensionality reduction manageable. Essentially, I am prioritising the retention of the most significant patterns or features in the data. This 90% threshold strikes a balance between preserving information and reducing the complexity of the model. It's a common practice in data analysis and dimensionality reduction to set thresholds based on the proportion of variability explained, ensuring that I retain a high level of signal while minimising noise.

```
Optimal k Value: 55

Dimensions of U_k: 69878 55

Dimensions of D_k: 55 55

Dimensions of V_k: 55 10677
```

Upon observing that $k = 55$ captures 90% of the variability, I generated three matrices: $D_k$ sized 55 x 55, $U_k$ sized 69878 x 55, and $V_k$ sized 55 x 10677. The total number of numeric values required to store these component matrices amounts to $(69878 \times 55) + (55 \times 55) + (55 \times 10677) = 4,433,550$. This reflects a reduction of approximately 99.4% compared to the original 746,087,406 entries. Despite the dimensionality reduction I can still do more work to reduce the memory needed to run my models. To address this, I employ another reduction technique, selecting relevant data using the entire rating matrix.

### 3.3 Relevant Data

During the data analysis process, I observed significant left skewness in both the user rating counts and movie ratings, indicating that a substantial portion of the data holds limited predictive value. To mitigate computational load without sacrificing predictive accuracy, I propose implementing a threshold for the minimum number of ratings required for inclusion in my models, both for users and movies.

```
Minimum number of movies (90th percentile): 302

Minimum number of users (90th percentile): 564

6960 x 2669 rating matrix of class 'realRatingMatrix' with 3344239 ratings.
```

If I only include movies and users within the 90th percentile, then any users or movies that fall into the lowest 10th percentile will not be included in the analysis. As a result the dataset consists of movies with a minimum of 302 ratings and users who have left a minimum of 564 ratings. This results in a matrix of 6960 distinct movies and 2669 distinct users, with 3344239 ratings.

# 4 Models and Results

The upcoming section will delve into a range of predictive models, including Linear Regression, Slope One, Matrix Factorisation, Ensemble Models, and Neural Networks. Each model will be accompanied by an explanation of its workings, rationale for its selection, and the predictive outcomes it generated. To evaluate the efficacy of the models' predictions, I will use Root Mean Squared Error (RMSE). A lower RMSE value indicates better predictive accuracy. The objective for this report is to achieve an RMSE of under 0.85, as stipulated by the task requirements. To construct the RMSE, one must determine the residuals. Residuals are the difference between the actual values and the predicted values. I denoted them by $\hat{y}_{u,i} - y_{u,i}$, where $y_{u,i}$ is the observed value for the ith observation and $\hat{y}_{u,i}$ is the predicted value.

The residuals value can be larger or smaller than the true value, and can therefore be positive or negative. Squaring the residuals, averaging the squares, and taking the square root gives us the RMSE. Which in turn always produces a positive number. I then use the RMSE as a measure of spread between the true values and the predicted values.

The equation to calculate the RMSE can be denoted as follows:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{u,i}\left(\hat{y}_{u,i} - y_{u,i}\right)^2}$$

## 4.1 Linear Regression

To begin with, I will start with a set of simple and straight forward linear regression models, that take into account some of the effects I found during the data exploration process.

***Mu***: I employed the average rating, denoted as $\mu$, as a benchmark for making predictions on the test data. In this model, each observation's prediction is simply the global average, represented as $Y_{u,i} = \mu + \varepsilon_{u,i}$, where $\hat{Y}_{u,i}$ signifies the predicted rating for each observation $i$. More specifically, M represents the global average rating, serving as the baseline or average rating across all observations, and $\varepsilon_{u,i}$ represents the error term for observation for $i$, capturing the deviation of the actual rating from the global average. Essentially, I utilise the global average rating of 3.51 as a predictor for how users will rate other movies. This model serves as a useful benchmark for future models, as any model exhibiting a higher error rate than the average introduces complexity and yields inferior outcomes.

```
RMSE for Mu: 1.060759

Training Time: 0.088 sec

Model Size: 7.6294 MB
```

. . . . . .

***Movie Effects***: I can improve upon my original model by adding movie bias to the equation denoted by $b_i$. This entails summing the differences between each individual movie rating within a particular movie group and $mu$. As a result, I obtained an average value that indicates the extent to which the set of ratings for each movie deviates from the overall average of all movies. Therefore, the model can now be written as:

$$Y_{u,i} = \mu + b_i + \varepsilon_{u,i}$$

This process enabled me to generate predictions with slightly greater precision with an RMSE score of 0.944.

```
RMSE for Movie Effect: 0.9441859

Training Time: 0.279 sec

Model Size: 7.7525 MB
```

.....

***Movie + User Effects***: Next I can apply the same logic for User effect, meaning the model will now take into consideration the difference between ratings from a specific user depicted by $b_u$, compared to the average of all movies. Therefore, the model can now be written as:

$$Y_{u,i} = \mu + b_i + b_u + \varepsilon_{u,i}$$

The new model allows me to generate predictions with a considerable improvement in precision achieving an RMSE score of 0.866. Through the initial data exploration I identified that there were noticeable year, genre, and time effects. Therefore, the next set of models will incorporate those variables to train and gain more predictive power.

```
RMSE for Movie + User Effect: 0.8660138

Training Time: 0.213 sec

Model Size: 8.5531 MB
```

.....

***Movie + User + Time Effects***: Including time as a variable increases the predictive power minimally, lowering the RMSE from 0.867 to 0.865. The model can now be written as

$$Y_{u,i} = \mu + b_i + b_u + b_t + \varepsilon_{u,i}$$

.

```
RMSE for Movie + User + Time Effect: 0.8659272

Training Time: 0.484 sec

Model Size: 8.5648 MB
```

.....

***Movie + User + Genre Effects***: Including genre as a variable also increases the predictive power minimally, lowering the RMSE from 0.867 to 0.866. The model can now be written as

$$Y_{u,i} = \mu + b_i + b_u + b_g + \varepsilon_{u,i}$$

.

```
RMSE for Movie + User + Genre Effect:  0.8657151

Training Time: 0.209 sec

Model Size: 8.6289 MB
```

.....

***Movie + User + Year Effects***: Including release year as a variable also increases the predictive power minimally, lowering the RMSE from 0.867 to 0.866. The model can now be written as

$$Y_{u,i} = \mu + b_i + b_u + b_y + \varepsilon_{u,i}$$

.

```
RMSE for Movie + User + Year Effect:  0.8656928

Training Time: 0.412 sec

Model Size: 8.5605 MB
```

.....

***Movie + User + Time + Genre + Year Effects***: Including release year as a variable also increases the predictive power minimally, lowering the RMSE from 0.867 to 0.865. The model can now be written as

$$Y_{u,i} = \mu + b_i + b_u + b_g + b_t + b_y + \varepsilon_{u,i}$$

.

```
RMSE for Movie + User + Genre + Time + Year Effect:  0.865492

Training Time: 0.551 sec

Model Size: 8.6479 MB
```

.....

***Regularisation***: Next I experimented by adding regularisation to the models to try a smooth out some of the large outlier values, and stop overfitting the data. Regularisation mitigates overfitting by adding a penalty to large values in the dataset. Specifically, I applied regularisation to model parameters $bi$ and $bu$ in the optimisation process, controlled by the regularisation parameter $\lambda$.

Adjusting $\lambda$ allows us to strike a balance between over and under fitting the training data, while maintaining a simpler model. This balance allows us to maintain accurate pattern capture while stabilising the models behaviour, by reducing sensitivity to noise and sparse data.

Mathematically, $n_i$ represents the number of ratings made for movie $i$. As $n_i + \lambda \approx n_i$, when $n_i$ becomes large, the model tends to stabilise on it's own due to the central limit theorem, and in turn the influence of $\lambda$ diminishes. Conversely, as $n_i$ decreases and data becomes sparse, and the estimate of $\hat{b}_i(\lambda)$ converges towards 0. Therefore, larger values of $\lambda$ are needed, and lead to stronger regularisation, causing the estimated coefficients to be contracted more aggressively. This contraction is necessary because they aren't naturally smoothed out by larger samples of $n$.

The regularisation is using cross validation to achieve the best $\lambda$ value for each variable. In the first model, I group by movieId, and then by userId. This approach assumes that each user's rating deviation from the mean is primarily influenced by the movie they are rating. The equations can be shown as:

movieID:

$$\hat{b}_i(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu}\right)$$

userID:

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu} - \hat{b}_i\right)$$

.

In contrast in the second model, I grouped by userId, and then by movieId. This approach assumes that each movie's rating deviation from the mean is primarily influenced by the user rating it. The equations can be shown as:

userID:

$$\hat{b}_i(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu}\right)$$

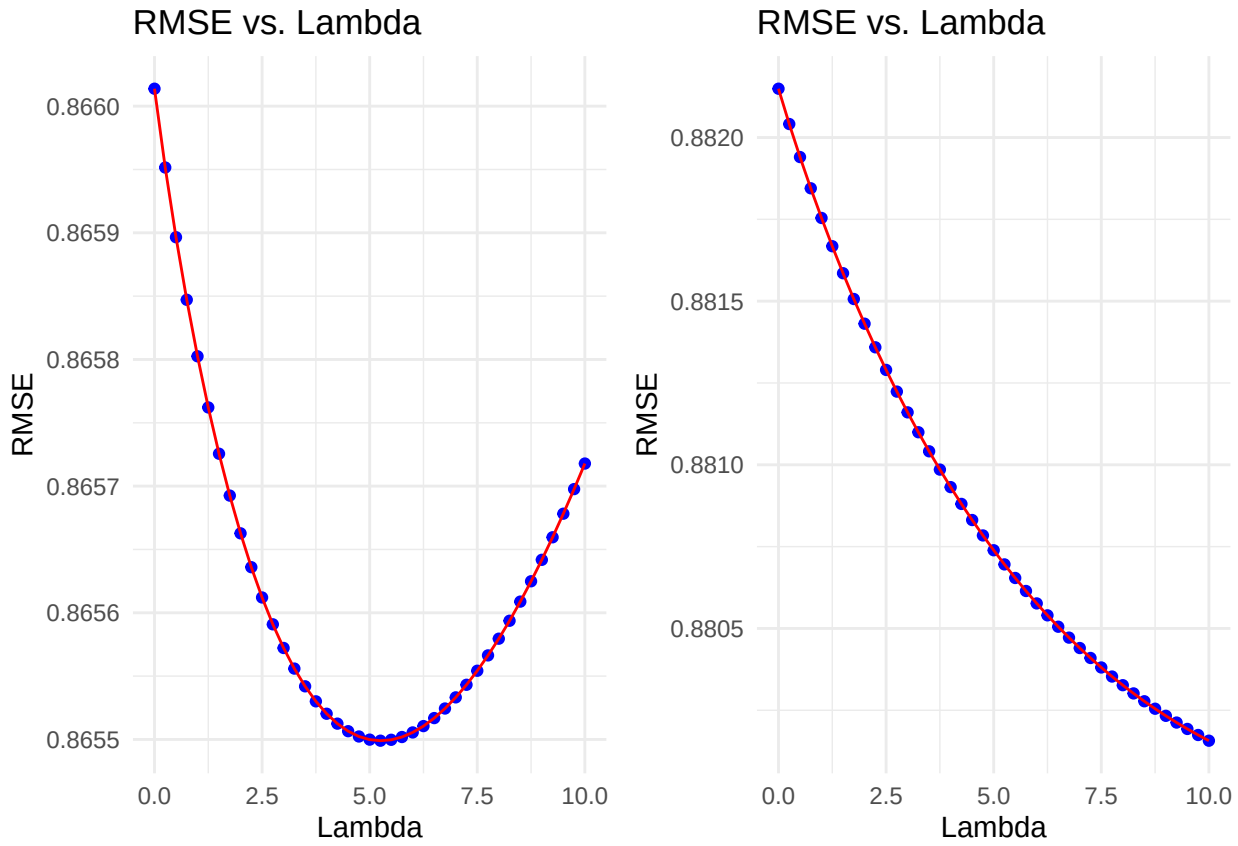
movieID:

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left( Y_{u,i} - \hat{\mu} - \hat{b}_i \right)$$

.

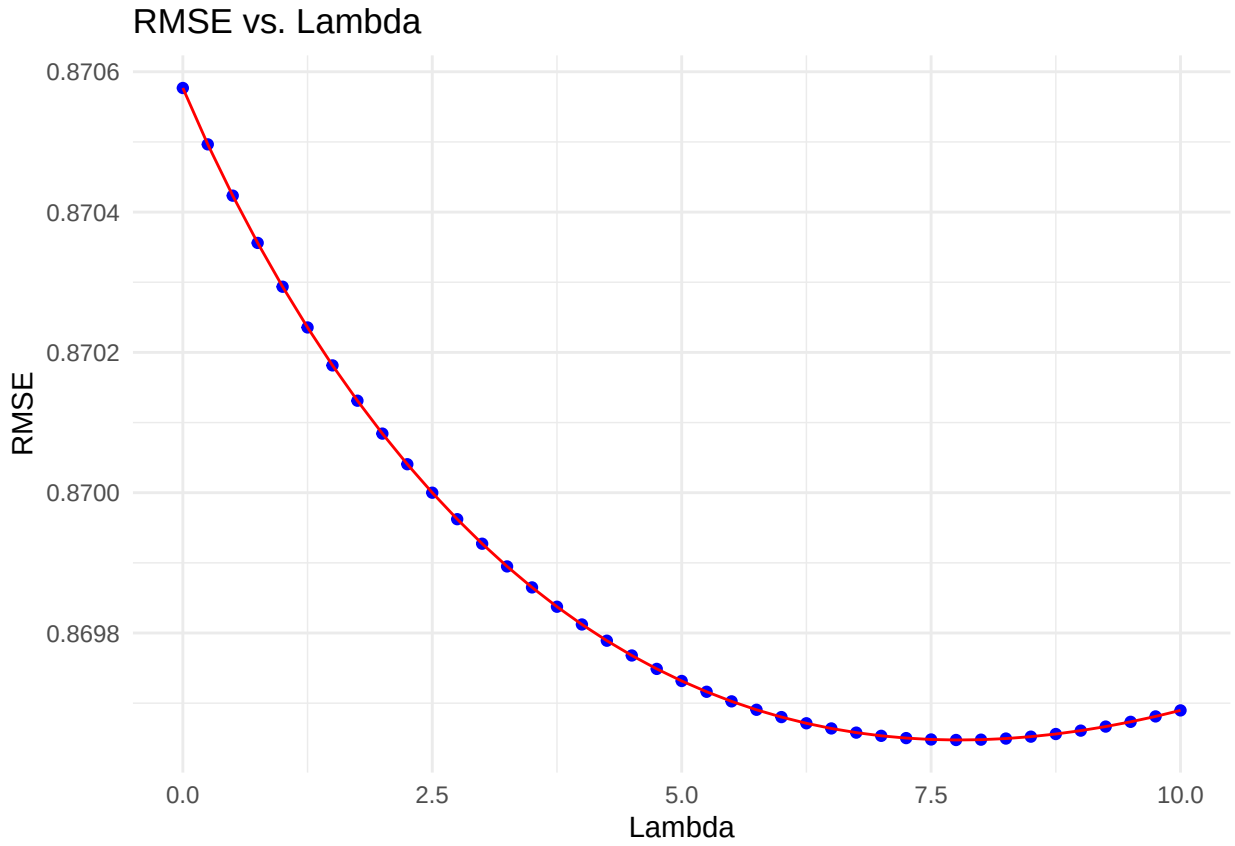
Typically, in collaborative filtering-based recommender systems, the first approach is more common, as ratings tend to vary more for different movies than for different users. Essentially, I expect to see a larger difference in ratings between different movies rated by the same user than between ratings for the same movie given by different users. For example, a user might have a strong preference for certain genres or actors, meaning their ratings will vary a lot; however, a movie tends to be less divisive, with people gravitating towards more similar, centralised ratings.

Nevertheless, I will investigate both regularisation methods for good measure, and exploratory purposes.




```
RMSE for movieId with Regularisation: 0.865499

Training Time: 99.114 sec

Model size: 8.5532 MB

RMSE for userId with Regularisation:  0.880157

Training Time: 88.172 sec

Model size: 8.5532 MB
```

I observed that applying regularisation to the movieId variables resulted in a slight decrease in RMSE from 0.866 to 0.865, and as expected regularisation to the userId increases the RMSE from 0.866 to 0.88. Moving forward, I will extend regularisation across all parameters grouped by movieId in pursuit of further enhancements.




Here I've included all the parameters grouped by movieId, userId, genres, year and then date. The equations can be shown as:

movieID:

$$\hat{b}_i(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu}\right)$$

userID:

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu} - \hat{b}_i\right)$$

genres:

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu} - \hat{b}_g\right)$$

year:

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu} - \hat{b}_y\right)$$

date:

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} \left(Y_{u,i} - \hat{\mu} - \hat{b}_t\right)$$

```
RMSE for Full Model with Regularisation:  0.8696475
Training Time: 256.222 sec
Model size: 8.648 MB
```

Including all the variables slightly increases the RMSE from 0.865 to 0.87. As the regularisation is increasing the RMSE, it suggests that the penalties are influencing the model in a way that leads to less accurate predictions on the dataset. This can occur if $\lambda$ is set too high, causing the model to underfit the data. In more simple terms, the regularisation might be too heavy, smoothing the data too much, and causing the model to become too simplistic and unable to capture the underlying patterns effectively.

Table 1: Results for Linear Regression

| Linear Regression | | | |
|---|---|---|---|
| Method | RMSE | Time (sec) | Size (MB) |
| Mu | 1.0607590 | 0.088 | 7.6294 |
| Movie Effect | 0.9441859 | 0.279 | 7.7525 |
| Movie + User Effects | 0.8660138 | 0.213 | 8.5531 |
| Movie + User + Time Effects | 0.8659272 | 0.484 | 8.5648 |
| Movie + User + Genre Effects | 0.8657151 | 0.209 | 8.6289 |
| Movie + User + Year Effects | 0.8656928 | 0.412 | 8.5605 |
| Movie + User + Time + Genre + Year Effects | 0.8654920 | 0.551 | 8.6479 |
| Regularised Movie + User Effect | 0.8654990 | 99.114 | 8.5532 |
| Regularised User + Movie Effect | 0.8801570 | 88.172 | 8.5532 |
| Regularised Movie + User + Time + Genre + Year Effect | 0.8696475 | 256.222 | 8.6480 |

Through the execution of multiple Linear Regression models, it becomes evident that user and movie effects stand out as the most influential variables for predicting ratings. Although the inclusion of additional variables and, in some instances, regularisation techniques marginally reduce the RMSE values, the magnitude of this improvement is negligible compared to the additional computational resources required. Thus, the practical benefits of incorporating these additional complexities are not justified.

## 4.2 Recommender Engines

Now I will briefly explore the recommenderlab package, which contains a set of algorithms specifically designed for making recommendations using collaborative filtering. The package doesn't accept training and test datasets outlined in this report's confines. Instead, the functions automatically split the data. However, a brief exploration and discussion of some of its features will provide valuable insight into the realm of possibilities.

```
     RMSE
0.8146398
```

First I explored the popular algorithm which is a is a non-personalised algorithm that recommends to all users the most popular items they have not rated yet. It acts as a good benchmark for assessing the performance of the personalised algorithms. The algorithm achieved an RMSE score of 0.8146, which is already better than my lowest score using linear regression.

```
     RMSE
0.7839743
```

Next I explored the packages user-based collaborative filtering algorithm, which predicts ratings by aggregating the ratings of users who have a similar rating history to the user receiving the recommendation. Center normalisation subtracts the mean rating of each user from their ratings, cosine similarity measures the cosine of the angle between two vectors in a multi-dimensional space, and "nn" specifies the number of nearest neighbors to consider when making recommendations. Choosing an appropriate number of neighbors is crucial for the performance of the recommendation system. A higher value of nn may capture more diverse preferences but can also introduce noise, while a lower value may lead to underfitting and missing out on relevant recommendations. Given my chosen parameters I managed to obtain a much better RMSE of 0.779.

```
     RMSE
0.8010077
```

Finally, I explored the item-based collaborative filtering algorithm with the same set of tuning parameters. Given my earlier exploration of linear regression models using the difference in movie ratings based on users compared to user ratings based on movies, I congruently obtained a higher RMSE score of 0.801.

### 4.3 Slope One

Next, I explored the Slope One algorithm. Slope One was introduced in a 2005 paper by Daniel Lemire and Anna Maclachlan. Unlike linear regression, which estimates a model using equations like $f(x) = ax + b$, Slope One utilises a simpler form of regression with a single free parameter represented as $f(x) = x + b$. This simplicity avoids the need for complex parameter estimation as in Linear Regression, meaning it scales more easily, and can be less susceptible to overfitting due to capturing too much noise in the training data. Slope One's simplicity can help mitigate this issue by not introducing unnecessary complexity into the model. In some instances, Slope One has been shown to be much more accurate than linear regression, especially in sparse datasets, and it requires half the storage or less compared to more complex models.

Slope One requires specific data formatting, therefore, the following steps are necessary for training the model.

```
RMSE for Slope One:  0.8637956
```

```
Training Time: 6860.541 sec
```

```
Model size: 2485.63 MB
```

Using the Slope One algorithm I was able to obtain an RMSE score of 0.864. Which is only slightly better than the Linear Regression model. Furthermore, the training time was substantially longer, with much more space taken in memory. However, the RMSE score was based on only 50% of the dataset due to my machines RAM limitations Therefore, there should be a note of caution, as the entire dataset might produce significantly better results.

## 4.4 Matrix Factorisation

In the context of recommender systems, matrix factorisation is a widely used technique to predict user-item ratings. The basic idea is to approximate the original rating matrix $R$ with the product of two lower-dimensional matrices: $P$ and $Q$.

Let $p_u$ denote the latent factors of user $u$, represented by the $u$-th column of matrix $P$, and $q_v$ denote the latent factors of item $v$, represented by the $v$-th column of matrix $Q$. The predicted rating given by user $u$ on item $v$ is then computed as the dot product of $p_u$ and $q_v$: $p'_u q_v$.

One common approach to determine the matrices $P$ and $Q$ involves solving a regularisation problem, which can be formularised as:

$$\min_{P,Q} \sum_{(u,v)} f(r_{u,v}, p'_u q_v) + \mu_P ||P||_F^2 + \mu_Q ||Q||_F^2 + \lambda_P ||P||_F + \lambda_Q ||Q||_F$$

Here, $(u, v)$ represents the observed entries in matrix $R$, $r_{u,v}$ is the observed rating, $f$ is the loss function, and $\mu_P$, $\mu_Q$, $\lambda_P$, $\lambda_Q$ are regularisation parameters to prevent overfitting.

In this optimisation problem, the objective is to minimise the difference between observed ratings and predicted ratings while penalising the complexity of matrices $P$ and $Q$ to avoid overfitting.

To predict the rating for item $i$ by user $u$, I simply compute the dot product of the transpose of $p_u$ and $q_i$: $p'_u q_i$.

First I started with recosystems recommendation engine, that uses stochastic gradient descent for optimisation of an objective function's optimum value. The package comes with a set of tunable parameters:

dim specifies the dimensions of the latent feature space. It's a vector containing three values. These values represent the number of latent factors used by the model. Changing this parameter directly influences the model's capacity to represent the underlying structure of the data. Increasing the dimensionality can enhance the model's ability to capture intricate patterns but may also increase the risk of overfitting.

lrate represents the learning rate used during training. It's a vector containing two values that control the step size during optimisation and affect how quickly the model learns. Changing this parameter impacts the model's training dynamics. Adjusting it too high may lead to unstable convergence or overshooting of the optimal solution, while setting it too low may result in slow convergence or getting stuck in local minima.

nthread defines the number of threads to be used during training. Changing this parameter affects the degree of parallelism employed during training. Utilising multiple threads can significantly speed up training on multi-core processors but may also introduce overhead and contention in resource-limited environments.

niter represents the number of iterations or epochs used during training. This parameter determines how many times the entire training dataset is passed through the model during training. Changing this parameter directly impacts the duration and depth of the training process. Increasing the number of iterations may improve the model's convergence and generalisation performance, but it also incurs higher computational costs and longer training times.

For my first model, I kept the variables stored in my random access memory (RAM). RAM is fast, but potentially more volatile.

```
RMSE for RAM Matrix Factorisation: 0.7838998

Training Time: 5178.734 sec

Model size: 160.2317 MB
```

Using Matrix Factorisation while storing the dataset in RAM, I was able to achieve a RMSE score of 0.784.

Next I decided to investigate using a different method for storing the datasets. Storing the dataset on disk and later reloading it before model training adds an extra step to the data processing pipeline. However, for large datasets that push a processing unit's memory to capacity, this step might become essential to facilitate

the operation. Moreover, by offloading data to disk, the model can access the entire dataset without memory limitations, potentially enhancing model accuracy. For example, by using disk space over RAM, I can reduce memory-related glitches during training, improve noise, and enable faster data retrieval and processing, thereby potentially enhancing model performance.

```
RMSE for Disk Matrix Factorisation:  0.7833761
```

```
Training Time: 2155.302 sec
```

```
Model size: 793.4668 MB
```

By using matrix factorisation with data read from disk memory, I improved the RMSE score slightly to 0.783. This improvement may be due to differences in how the algorithms utilise available resources and round floating-point values. Additionally, the data loading process might vary slightly. Matrix factorisation algorithms often involve random initialisation of latent factors, and even with the same random seed, implementation differences can lead to variations in these initial values. Furthermore, the stochastic nature of gradient descent can cause slight variations in data processing and updates, especially if the data is chunked differently for disk-based processing.

Another notable finding pertained to the variance in training time. Traditionally, RAM outpaces disk access in speed. However, the disk-based algorithm exhibited notably quicker performance despite the larger model size. This discrepancy may stem from scenarios where the dataset surpasses available RAM, triggering memory swapping or suboptimal memory handling. In memory swapping, inactive portions of the dataset are moved from RAM to disk, potentially impeding the training process by necessitating subsequent retrieval.

Conversely, the disk-based approach operates by streaming or processing data in smaller, more manageable portions instead of loading the entire dataset into memory at once, as in RAM. This methodology effectively reduces the memory footprint and mitigates the necessity for swapping, thus contributing to enhanced efficiency.

Next I decided to investigate the other commonly used algorithm in Matrix Factorisation, Alternating Least Squared Means (ALS). ALS is particularly effective for large-scale recommendation problems because it can be parallelised easily. By alternating between updating the user and item factors, ALS gradually improves the quality of the latent factor representations until convergence.

I also decided to investigate using Apache Spark which is an open-source unified analytics engine for large-scale data processing. It automatically analyses the jobs operations and distributes the work across the multiple nodes in the cluster.

Spark achieves parallelism through a concept called Resilient Distributed Datasets (RDDs). RDDs represent distributed collections of objects across the cluster, and automatically partition and distribute themselves across the nodes in the cluster, allowing parallel processing of data. RDDs are the fundamental data structures in Spark, meaning I must initially convert the dataframes into a format that Spark can work with. Additionally, Spark leverages in-memory computing and caching to minimise data movement across the cluster, further improving performance by reducing disk I/O (input/output) and network overhead.

```
RMSE for Alternating Least Square Means:  0.8445612
```

```
Training Time: 90.368 sec
```

```
Model size: 361.5343 MB
```

Using ALS combined with spark, I was able to obtain an RMSE of 0.845. However, this used only one fifth of the data, due to the resource constraints of my machine. With the adequate hardware this number could be significantly lower, with a very fast computation time.

Next I ran another Alternative Least Square Algorithm, using the recommednerlab package. I kept the partition of data the same as the spark version, to allow me to compare and contrast the two.

```
RMSE for Alternating Least Square Means:  0.9202998
```

```
Training Time: 11818.33 sec
```

```
Model size: 8737.141 MB
```

Using ALS means from the recommenderlab package, I was able to obtain an RMSE score of 0.96.

Table 2: RMSE Results for Matrix Factorisation

| Matrix Factorisation | | | |
|---|---|---|---|
| Method | RMSE | Time (sec) | Size (MB) |
| SlopeOne | 0.8637956 | 6860.541 | 2485.6302 |
| Matrix factorisation using RAM | 0.7838998 | 5178.734 | 160.2317 |
| Matrix factorisation using Disk | 0.7833761 | 2155.302 | 793.4668 |
| Alternating Least Squares using Spark | 0.8445612 | 90.368 | 361.5343 |
| Alternating Least Squares using Recommender | 0.9202998 | 11818.335 | 8737.1408 |

Among all the matrix factorization algorithms I explored, the disk-based Matrix Factorization delivers the best results. However, these results are somewhat dependent on the hardware used. The Alternating Least Squares and SlopeOne algorithms only utilise a subset of the data, therefore I can conclude that the results obtained may only be optimal on a machine with highly limited resources.

## 4.5 Ensemble Methods

Next I investigated a selection of ensemble methods, which combine the predictions of multiple individual models to improve overall performance. The basic idea behind ensemble methods is that I can merge the predictions of multiple weaker models into one stronger and more robust model.

In addition, I explored the capabilities of H2O's open-source distributed in-memory machine learning platform, renowned for its linear scalability, facilitating the processing of exceptionally large datasets. This platform is equipped with a plethora of libraries specifically designed for executing ensemble methods, each serving distinct purposes. One such method is Gradient Boosting Decision Trees (GBDT), characterised by sequential training where each subsequent tree aims to rectify errors made by its predecessors.

Another notable technique offered by the platform is Random Forests, which operate independently, utilising a random subset of features and a bootstrapped sample of the training data. These trees grow to their maximum depth without pruning, contributing to their robustness.

Finally, the platform supports stacked ensembles, a sophisticated approach that amalgamates predictions from diverse base models. These base models, including GBDT, Random Forests, Support Vector Machines (SVM), and Neural Networks, are trained independently. Their predictions are then harmonized using a meta-learner, which essentially functions as a model that connects all the base layers.

First, I investigated a gradient boosted decision tree (GBDT). The initial step involved training two different models. The first model was configured with 50 trees, a maximum tree depth of 5, a learning rate of 0.1, and 3-fold cross-validation. The features used for this model were movieId, userId, n.movies_byUser, and n.users_bymovie.

The second model was similarly configured with 50 trees, a maximum depth of 5, a learning rate of 0.1, and 3-fold cross-validation. However, this model included a random seed for reproducibility, kept cross-validation predictions, and set fold assignment to random. The features for this model were limited to movieId and userId.

After comparing the RMSE of the two models on the training set, I found that the second model exhibited a lower RMSE Therefore, I evaluated the second model on the test set.

```
RMSE for Gradient Boosting:  0.9869315

Training Time: 373.611 sec

Model size: 10.0513 MB
```

Using the GBDT method I was able to achieve a RMSE score of 0.987.

Next I investigated the performance of random forest models in the same manner. The initial random forest model was configured with 50 trees and a maximum tree depth of 20. The features used for this model were movieId, userId, timestamp, n.movies_byUser, and n.users_bymovie.

Subsequently, a second random forest model was trained with a similar configuration of 50 trees and a maximum depth of 20, but with additional parameters for 3-fold cross-validation, a random seed for reproducibility, cross-validation predictions enabled, and random fold assignment. This model used only movieId and userId as predictors.

After comparing the RMSE of the two models on the training set, I found that the second model had a lower RMSE. Consequently, I proceeded to evaluate the second model on the test.

```
RMSE for Random Forest:  0.9521627

Training Time: 1219.974 sec

Model size: 10.019 MB
```

Using the Random Forest ensemble method I was able to achieve a RMSE score of 0.953.

Finally, in an attempt to further enhance the accuracy of movie rating predictions, I implemented a stacked ensemble model, which combined the strengths of the previously developed gradient boosted decision tree and random forest models.

The ensemble model was trained using movieId and userId as predictors. The stacked ensemble was created with the model_id "my_ensemble_auto" and included the gradient boosted decision tree and random forest models as base learners.

This approach of using a stacked ensemble leverages the strengths of multiple learning algorithms, potentially leading to improved predictive performance. By combining the gradient boosted decision tree and random forest models, the ensemble model aimed to provide a more robust and accurate prediction of movie ratings compared to individual models.

```
RMSE for Stacked Ensemble:  0.9514228
```

```
Training Time: 16.797 sec
```

```
Model size: 10.061 MB
```

Using the Stacked Ensemble method I was able to achieve a RMSE score of 0.952, which was ever so slightly better than the base models on their own.

Table 3: RMSE Results for Ensemble Methods

| Ensemble Methods | | | |
|---|---|---|---|
| Method | RMSE | Time (sec) | Size (MB) |
| Gradient Boosting | 0.9869315 | 373.611 | 10.0513 |
| Random Forest | 0.9521627 | 1219.974 | 10.0513 |
| Stacked Ensemble | 0.9514228 | 16.797 | 10.0610 |

Looking across all the different ensemble methods, the GBDT model demonstrated an RMSE of 0.987, which indicates a moderate level of predictive accuracy. The training time for this model was 301.393 seconds, and the model size was 10.0507 MB. While the GBDT model was relatively quick to train, its predictive performance was slightly lower compared to the random forest and stacked ensemble models. The random forest model achieved a lower RMSE of 0.953, suggesting better predictive accuracy compared to the GBDT model. However, the improvement in accuracy came at the cost of a longer training time of 1069.031 seconds. The model size was comparable to the GBDT model, at 10.0200 MB. This indicates that while the random forest model provides better predictions, it requires more computational resources and time to train. Finally, the stacked ensemble model exhibited the best predictive performance with an RMSE of 0.952. Notably, the training time for the ensemble model was only 22.192 seconds, making it the fastest model to train. The model size was slightly larger at 10.0613 MB, but this increase is negligible.

Out of all the models, the ensemble model emerged as the most effective approach, achieving the lowest RMSE and it was the fastest to train, significantly reducing computational time compared to both the gradient boosting and random forest models. The slight increase in model size is a minor trade-off for the substantial gains in accuracy and efficiency.

## 4.6 Neural Networks

Finally, I explored the effectiveness of Neural Networks. In order to run a neural network model, I needed to do some work to wrangle the data into the correct format for processing.

To understand the latent representations within my models, I bound the data back together into one dataset, and created a dense column comprising of the movieIDs. The embedding layers map these categorical variables to dense vectors in a continuous space, where similar movies have similar vector representations. Without a dense representation of movie IDs, the embedding layers would not be able to effectively learn meaningful representations and patterns within the movie data. Furthermore, the model expects sequential mapping of inputs. In the context of the movieId's, a sequential representation ensures that the model can learn meaningful relationships between consecutive IDs, which might correspond to similar or related movies. This is why we must recombine the data into a single set before creating dense layers, and subsequently divide it back into a test and training set later in the process. This is opposed to creating dense layers after the data has been divided, which results in inconsistencies in the dense data layer, which prevents neural networks from finding meaningful patterns.

To do this I created indices for the training set. I selected rows from the ratings dataset based on the sampled indices to create the training dataset. I selected rows not included in the training indices to create the validation dataset. I divided the relevant variables into training and validation matrices for processing, and finally, I set the dimensionality of the embedding layer.

The next steps involved building out the model.

I used the Keras package from TensorFlow. Originally, Keras was a separate library, but it was integrated into TensorFlow 1.4 as its official high-level neural networks API. I need to use the keras_model_custom, because I'm using more than one variable. Furthermore, because TensorFlow has been written in Python, R must use reticulate to convert the code into Python for it to be processed. Therefore, I used an objected orientated approach to build the model.

First an embedding layer is defined for users and movies. Embedding layers are used to map users and movies to dense vectors of fixed size outlined using the embedding_dim. These vectors represent users and movies in a lower-dimensional space, capturing latent features that are learned during training. Latent factors are essentially hidden patterns in the data, which might represent aspects such as genre, style, cast, or even abstract qualities like emotional tone or pace. This initial step helps in finding patterns and similarities between users and movies.

Then bias embeddings are added for users and movie, which account for individual user and movie biases that can influence ratings. For example, some users might generally give higher ratings while some movies might consistently receive lower ratings regardless of the user. Adding bias embeddings helps in capturing these tendencies and improving the accuracy of predictions.

Then I created dropout layers to prevent overfitting. The dropout layers randomly set a fraction of input units to zero during training. This helps prevent overfitting by ensuring that the model does not become too reliant on specific neurons, promoting the learning of more robust features that generalise better to new data.

Next, the lambda layers are defined for calculating dot product and adding biases. Lambda layers are used to implement custom operations within the model. The first lambda layer computes the dot product of user and movie embeddings, representing the interaction between a specific user and movie. The dot product is a mathematical operation that takes two equal-length sequences of numbers (in this case, the embeddings of users and movies) and returns a single number. It is calculated by multiplying corresponding elements of the vectors and then summing these products. For example, if the user embedding is $[u_1, u_2, ..., u_n]$ and the movie embedding is $[m_1, m_2, ..., m_n]$ the dot product is computed as $u_1\ m_1 + u_2\ m_2 + ... + u_n,\ m_n$. This result quantifies the similarity or interaction strength between the user and the movie based on their latent features. The second lambda layer adds the user and movie biases to this dot product, refining the prediction by incorporating these additional influences. Bias terms adjust the baseline predictions for individual users and movies, accounting for their inherent tendencies (e.g., a user's general preference for higher ratings or

a movie's overall average rating). This adjustment helps produce more accurate and personalised rating predictions.

Then another lambda layer is defined to scale the output to the desired rating range. The final lambda layer scales the output of the previous layer to the desired rating range (e.g., 1 to 5). This ensures that the predicted ratings are within a realistic and expected range, making the predictions more meaningful and comparable to the actual ratings.

Now that the model has been defined, it needs to be compiled, which involves specifying the loss function and the optimiser. Mean Squared Error (MSE) is chosen as the loss function because it measures the average squared difference between predicted and actual ratings, providing a clear measure of prediction accuracy. MSE is particularly effective in regression tasks like this one, where the goal is to predict a continuous output (movie ratings). By squaring the differences, MSE penalises larger errors more heavily, encouraging the model to make accurate predictions. This can be contrasted with Mean Absolute Error (MAE), which measures the average absolute differences. While MAE is robust to outliers and provides a linear penalty for errors, it does not penalise large errors as strongly as MSE, making it less generalised, potentially leading to less precise predictions on unseen data sets.

The Adam optimizer was selected for its efficiency and effectiveness in training deep learning models. Adam (short for Adaptive Moment Estimation) combines the benefits of two other popular optimisers: AdaGrad and RMSProp. It adapts the learning rate for each parameter individually by computing adaptive learning rates from estimates of first and second moments of the gradients. This makes Adam particularly well-suited for problems with sparse gradients or noisy data. Compared to the standard Stochastic Gradient Descent (SGD), which uses a single learning rate for all parameters and requires careful tuning, Adam is more robust and often converges faster. Another alternative, RMSProp, also adapts learning rates but does not incorporate momentum, which can slow down convergence in some cases. Adam's combination of adaptive learning rates and momentum often results in faster convergence and better performance.

Training the model involves using the training data to learn the parameters (embeddings and biases) that minimise the loss function. Validation data is used to monitor the model's performance on unseen data, ensuring that it generalises well. Early stopping is a callback that stops training when the validation performance stops improving, preventing overfitting and saving computational resources.

I then stored the variables in a single object (history) and run the fit function to fit the model to the data. Specifically, I stored information about the training process, including the loss and metric values for each epoch for both the training and validation data. This information can be used to analyse the model's learning curve and diagnose potential issues like overfitting or underfitting.

The x_train and y_train variables are the training data inputs and their corresponding labels. x_train contains the user-movie pairs, and y_train contains the actual ratings given by users to movies. Each epoch is one complete pass through the entire training dataset. Setting epochs = 10 means the model will iterate through the training data 10 times. Multiple epochs allow the model to learn and adjust its parameters incrementally, improving prediction accuracy.

The batch size specifies the number of samples per gradient update. Instead of updating the model parameters after each training sample (which can be noisy and slow) or after all samples (which can be slow and may not fit in memory), mini-batch gradient descent updates the model parameters after every 32 samples. This approach balances efficiency and performance, helping to smooth out updates and speeding up the training process.

Increasing the batch size can lead to more stable and accurate gradient estimates since each update is based on a larger sample of data. This can result in faster convergence and improved utilisation of hardware such as GPUs, which are optimised for parallel processing. However, larger batch sizes require more memory and can potentially lead to poorer generalisation as the model might overfit to the training data more quickly.

On the other hand decreasing the batch size results in noisier updates, since each update is based on fewer samples. This can act as a form of regularisation, potentially improving generalisation and reducing the risk of overfitting. Smaller batch sizes also require less memory, making them suitable for training on hardware

with limited resources. However, training with very small batch sizes can be slower and might lead to less stable convergence. Choosing an appropriate batch size is crucial for balancing memory usage, training speed, and model generalisation. I chose batch 32 after I ran multiple iterations and observed performance and available computational resources.

The x_valid, y_valid data is used to evaluate the model's performance during training. By providing validation data, the model can be evaluated on unseen data at the end of each epoch. This helps monitor overfitting and model generalisation. x_valid contains validation user-movie pairs, and y_valid contains the actual validation ratings.

I added a callback function to stop training if the performance did not improve after a specified number of epochs. This prevents overfitting and saves computational resources by not continuing to train when no improvement is seen. Finally, predictions are made on the validation set using the trained model. Root Mean Squared Error (RMSE) is calculated between predicted ratings and actual ratings (y_valid).

```
RMSE for Complex Dot Model:  0.83176
```

```
Training Time: 207988.9 sec
```

```
Model size: 8.1048 MB
```

Using the Complex Dot Model I was able to achieve a RMSE score of 0.8302.

Next I experimented by simplifying the custom lambda layers. The biases were customised using a lambda layer that also applied a sigmoid activation function to the sum of the dot product and biases. In the simplified model I used the layer_add, which simplifies the architecture and does not include the sigmoid activation. Furthermore, I added batch normalisation for user and movie embeddings before applying dropout to help in stabilising and accelerating the training process by normalising the activations.

```
RMSE for Simple Dot Model:  0.802482
```

```
Training Time: 179077.8 sec
```

```
Model size: 1.761 MB
```

Using the Simplified Dot Model I was able to achieve a RMSE score of 0.8025.

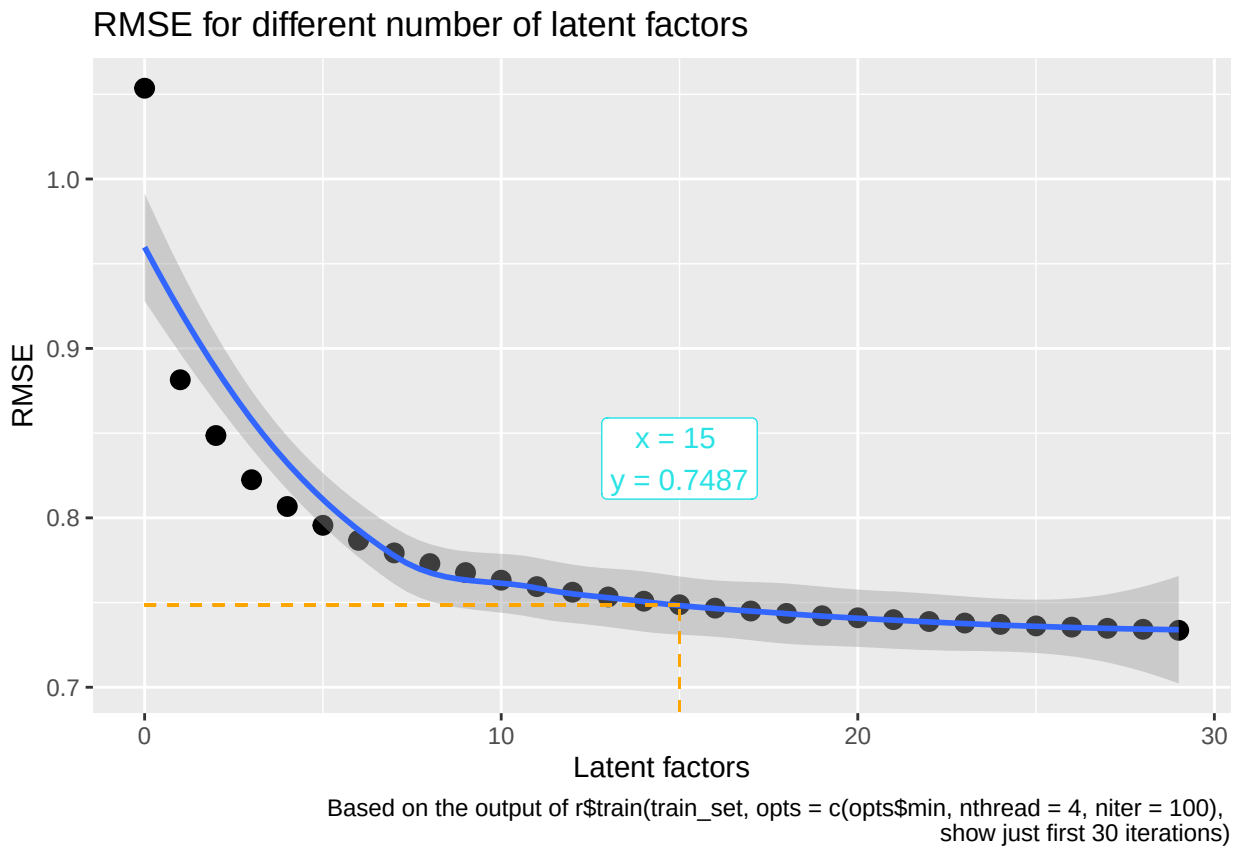
Table 4: RMSE Results for Neural Networks

| Neural Networks | | | |
|---|---|---|---|
| Method | RMSE | Time (sec) | Size (MB) |
| Complex Dot NN | 0.831760 | 207988.9 | 8.1048 |
| Simplified Dot NN | 0.802482 | 179077.8 | 1.7610 |

Ultimately, neural networks provide a large amount of tunable parameters, and therefore it stands to reason that a lower RMSE could be possible with a different set of settings. However, it's high level of customisation and complexity poses problems. Iterating through different model settings is time consuming, and makes understanding exactly how the models are working very difficult. This has been described as the "black box problem", which refers to the lack of transparency and interpretability of AI algorithms. It is often difficult to describe mathematically what criteria the systems are actually using to make their predictions, which makes it challenging to identify and rectify errors or biases. Moreover, scaling neural networks is resource heavy, time consuming and potentially volatile given the lack of transparency.

Now that I've reviewed multiple models, I can choose the most efficient one to make predictions and understand its performance in more detail using cross-validation. This technique trains and tests the model on subsets of the data, called folds. The process is repeated multiple times, each time using a different fold for evaluation, and the results are recorded over time to obtain the model's performance as it trains. This way, I can obtain RMSE values throughout the training process to understand where I'm getting the most

value for resource.




The figure above shows the amount of latent factors needed to achieve a subset of RMSE values. The more latent factors used, the more computationally expensive the models are. I can see that there is a sharp decline in RMSE values up until around 0.8. Then there is a progressively slower decrease in RMSE values, compared to the amount of resource needed. To obtain an RMSE of below 0.8 I only need 15 latent factors. Therefore, I could look to optimise my model to achieve a much more efficient RMSE, using far less resource.

| Top Picks for User 26489 | | | | |
|---|---|---|---|---|
| Movie ID | Predicted Rating | Title | Genre | Year |
| 7140 | 5.40 | Legend of Leigh Bowery, The | Documentary | 2002 |
| 6691 | 5.30 | Dust | Drama\|Western | 2001 |
| 64275 | 5.24 | Blue Light, The (Das Blaue Licht) | Drama\|Fantasy\|Mystery | 1932 |
| 65133 | 5.19 | Blackadder Back & Forth | Comedy | 1999 |
| 32657 | 5.18 | Man Who Planted Trees, The (Homme qui plantait des arbres, L') | Animation\|Drama | 1987 |
| 33264 | 5.15 | Satan's Tango (Sátántangó) | Drama | 1994 |
| 61037 | 5.13 | Silent Light (Stellet licht) | Drama | 2007 |
| 5194 | 5.12 | Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) | Comedy | 1980 |
| 58808 | 5.11 | Last Hangman, The | Drama | 2005 |
| 8536 | 4.91 | Intended, The | Drama\|Thriller | 2002 |

| Bottom Picks for User 26489 | | | | |
|---|---|---|---|---|
| Movie ID | Predicted Rating | Title | Genre | Year |
| 5837 | 0.59 | Legion of the Dead | Comedy\|Horror | 2000 |
| 5805 | 0.63 | Besotted | Drama | 2001 |
| 8394 | 0.64 | Hi-Line, The | Drama | 1999 |
| 7120 | 0.83 | DarkWolf | Horror | 2003 |
| 3437 | 0.83 | Cool as Ice | Drama | 1991 |
| 64999 | 0.88 | War of the Worlds 2: The Next Wave | Action | 2008 |
| 5724 | 0.93 | Creature Wasn't Nice, The (a.k.a. Naked Space) (a.k.a. Spaceship) | Comedy\|Horror\|Musical\|Sci-Fi | 1981 |
| 61348 | 0.96 | Disaster Movie | Comedy | 2008 |
| 5271 | 0.97 | 30 Years to Life | Comedy\|Drama\|Romance | 2001 |
| 4255 | 0.97 | Freddy Got Fingered | Comedy | 2001 |

# 5 Making Predictions

Finally, I used my best model to predict and display the top and bottom 10 movies for a specific user, to show the recommendation model in an applied setting. These results could be displayed on a webpage, or sent in an email, providing real value to an end user.

# 6 Conclusion

My report on recommendation algorithms examines various models, including Linear Regressions, Matrix Factorisation, Ensemble Methods, and Neural Networks. I also explored several libraries, infrastructural frameworks, and services to optimise model training.

Model accuracy was assessed using Root Mean Squared Error (RMSE), a common metric for its interpretability and its ability to penalise larger errors more severely. Infrastructural load was analysed based on training time and memory usage. Time reflects computational resources, which are essentially tied to energy consumption. Although all models were tested locally, tools like Spark and H2O create clusters to distribute computational resources across multiple nodes, speeding up model training and data processing. Therefore, time should be adjusted based on the hardware used to provide a more accurate assessment of energy consumption. Moreover, time itself is a valuable resource, and can be analysed in isolation as shorter training times are desirable.

# 7 Recommendations

Based on the analysis, the following models are recommended:

Linear Regression with Movie + User + Genre Effects: This model achieved an RMSE of 0.8657, a training time of 0.209 seconds, and used 8.6289 MB of memory. Despite its simplicity, it provides reasonably accurate movie rating predictions with very fast training times and minimal memory usage.

Matrix Factorisation using Disk Memory: If accuracy is the primary concern, this model is the best choice with the lowest RMSE of 0.7834. However, it requires a long training time of 2155.302 seconds and uses 793.4668 MB of memory.

Alternating Least Squares (ALS) using Spark: This model offers a balanced option with an RMSE of 0.8446, a training time of 90.368 seconds, and a memory usage of 361.5343 MB. Due to hardware constraints, only a subset of the data was used, and a more accurate RMSE may be achieved with additional resources. Furthermore, there is potential for ALS to outperform Matrix Factorisation in all metrics given its current performance and resource usage.

The limitations of this report are primarily related to hardware constraints. The study would benefit from being conducted on a more powerful machine. Additionally, there is a lack of resources to measure computational and energy consumption, which are crucial for large-scale model training. A more detailed analysis of hardware and energy requirements, and how these scale with hardware and data, would further enhance the findings.

Another limitation to consider is the tendency for recommendation systems to overgeneralise with widely admired or widely disliked movie titles. Social norms can influence individual ratings; for instance, if a movie is well-respected, people might rate it more highly due to its reputation, leading to its more frequent recommendation. This can skew recommendations towards popular movies, while potentially overlooking more personalised suggestions.

To address this issue, one approach could be to introduce a subcategory for recommendations, such as "Movies We Think You'll Like, That Everyone Loves." This allows for the inclusion of highly-rated films while also leaving room for further subcategorised recommendations that are more tailored to the user's individual preferences and perceived uniqueness.