Recommendation System Using Collaborative filtering

Asim Osman [1]

Abstract

This project presents the development of a collaborative filtering-based recommender system using Alternating Least Squares (ALS), with latent factor embeddings for users, movies alongside with their biases. Also, we address the cold start problem and improve recommendation accuracy by incorporating feature embeddings for movie genres in the last version of the model using an efficient and fast algorithm that updates these embeddings in a matter of seconds. We utilized multiple MovieLens datasets, starting with the small 100k ratings for development and testing, and eventually we trained the final model on both the 25m ratings and the 32m ratings datasets. Our implementation involves a custom ALS framework that alternately updates the user, movie, and genre factors and biases, optimizing for regularized negative log likelihood function and Root Mean Square Error (RMSE). The final model demonstrates an effective approach to recommendation with 0.77 RMSE on the 32m dataset, balancing computational efficiency and accuracy. Github

1. Introduction:

1.1. Problem Statement and Objective

In streaming and digital content, most movie-viewing patterns follow a power law distribution. This means that a handful of popular movies attract the majority of views and ratings, while lesser-known films receive minimal engagement and are not being exposed as a result of being at the tail of that distribution. Without personalized recommendations, users face limited diversity in suggestions, primarily seeing widely viewed content rather than films suited to their specific tastes. Hence the need for systems that models user-movie interaction and give recommendations that are particularly tailored to the taste and preferences of users and give more exposure to less known movies, leading to more user satisfaction and engagement. The aim of this work is to to develop a personalized movie recommendation system that learns meaningful and accurate embeddings for the movies and the users using the user's-movies interaction patterns via explicit feedback. Hence learning the specific preferences of every user and the properties of each movie that will help provide great recommendations.

1.2. Background and past work

There are two main types of recommender systems strategies, content based filtering, which suggests items to a user based on the characteristics of the items they have previously interacted with, using metadata or descriptive features associated with each item. It builds a user profile that summarizes their information and their preferences and then searches the database to identify similar items for this user. This approach requires external information about the movies and users, such as movie genres or the actors in the movie and users demographics, age and gender information that might not be easily available or easy to collect. This would be great for music recommendations, as this kind of information can be found, however, content-based filtering has a significant drawback in that it relies solely on item features to generate recommendations, which may not accurately reflect a user's preferences. For example, if a user enjoys the movie "Gaslight," the system might suggest other films by the same director or featuring the same actor, but these may not align with the user's specific interests, such as particular plot elements or production styles. This limitation means that the system cannot effectively differentiate between a user's likes and dislikes if the relevant features are not captured in the item profiles. Consequently, recommendations tend to be similar and may not fully satisfy the user's tastes.

An alternative approach that doesn't require external metadata is collaborative filtering, which relies on past user behavior, such as watch history and movies ratings, rather than explicit user profiles. because collaborative-based methods draw recommendations from a pool of users who have similar likes to one given user, they can often recommend items

[1]African Institute for Mathematical Sciences (AIMS) South Africa, 6 Melrose Road, Muizenberg 7975, Cape Town, South Africa. Correspondence to: Asim Osman <Asim@aims.ac.za>.

that a user may have not considered, appears with different features than a user's previously liked items but that retain a some unrepresented element that appeals to a user type this approach can capture complex data aspects that content filtering may miss. However, collaborative filtering faces the "cold start problem," making it less effective for new movies and users, where content filtering performs better.

The two primary areas of collaborative filtering are the neighborhood methods and latent factor models. In neighborhood methods, instead of using the content features of items to determine what to recommend, it finds similar users and recommends items that they like, this method has two versions, user-user version and the item-item version. In the user-user version, to estimate a user's x rating of an item, we will find all the users that are similar to x who rated these items and calculate estimates based on their ratings for that item, and the item-oriented approach predicts a user's preference for an item based on ratings of similar items by the same user.

The latent factor models - in a general sense- tries to embed all users and items as vectors in an n dimensional space where similar items and users will appear close in that space, and hence the recommendations are made based primarily on distance between user vectors and items vectors, these vectors capture complex relationship among users and movies as they try to to explain the ratings by characterizing both items and users. One of the best implementations of latent factors methods is Matrix Factorization (Koren et al., 2009). this method maps the user-item interaction in a sparse matrix, then decomposes this matrix to user factors and movie factors.

Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) (Lü et al., 2012) were some of the first matrix decomposition methods applied to recommender systems, used to reduce the dimensionality of user-item matrices while capturing key patterns. SVD, however, has limitations in handling sparse data and cold starts.

Alternating Least Squares (ALS) emerged as a solution to overcome some limitations of SVD. ALS decomposes the user-item matrix into user and item factors, iteratively optimizing each set of factors to minimize reconstruction error. Research like Koren et al. (2009) popularized ALS in large-scale settings, notably during the Netflix Prize competition

1.3. ALS in Large-Scale Recommender Systems

The use of ALS in large-scale systems was further explored by Hu, Koren, and Volinsky (2008) in their implicit feedback model, where they adapted ALS for implicit data, like clicks or views. This work was crucial because it allowed ALS to be used in cases where users don't provide explicit ratings, making it more versatile for real-world applications. it is also used on a large scale in Xbox Recommender System, which relies on implicit feedback, and runs on a large scale, serving tens of millions of daily users.

2. Methodology

2.1. Dataset

I conducted experiments using widely used recommendation systems benchmark datasets: MovieLens-100K (ML-100K), MovieLens-25M (ML-25M), and MovieLens-32M (ML-32M) (Harper & Konstan, 2015), the 100K ratings version was used in the first stages of the model, and finally the 25M and 32M versions were used to build the final version of the recommender system. This dataset describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. It contains 32,000,204 ratings and 2,000,072 tag applications across 87,585 movies with additional genre information about every movie. Users were selected at random for inclusion. All selected users had rated at least 20 movies. The dataset is of a sparse nature because of the limited number of ratings per user and item, typical in real-world recommendation datasets,

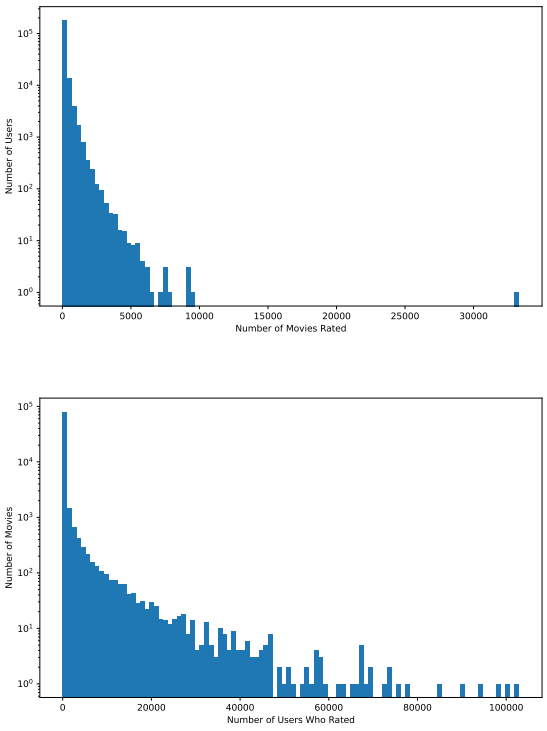

Figure 1: Histogram of User Degrees (Number of Movies Rated per User) and Movie Degrees (Number of Use/rs per Movie)

The analysis for this dataset in the above Figure 1 shows a

typical power law distribution in both user activity (ratings given) and movie popularity (ratings received) with a few popular movies and highly active users accounting for most of the interactions. This distribution motivates the need for collaborative filtering to provide recommendations for the long-tail items. Figure 2 shows better illustration of this behavior on a log-log scale.

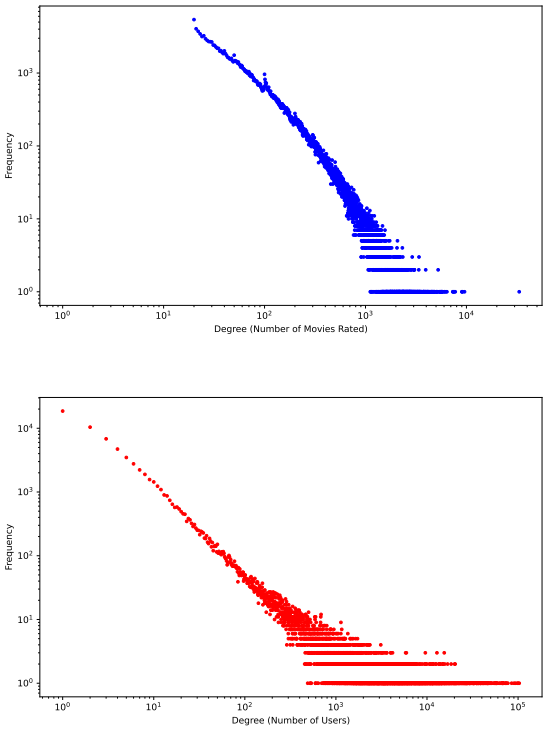

Figure 2: Log-log scale distributions showing degree patterns of user and movie interactions. This pattern is typical in collaborative filtering datasets and highlights the need for a recommendation system that addresses both frequent and infrequent participants.

Further Exploration of the MovieLens Dataset shows intresting user tendecies as ratings distributions tend to be relativly high, which is expected it movies platforms as users tend to explicitly rate the movies they liked, and ignore the movies the did not like. Figure 3 shows that certain ratings—typically around the higher end, such as 4 or 5—are likely given more frequently. This indicates a bias toward higher ratings.

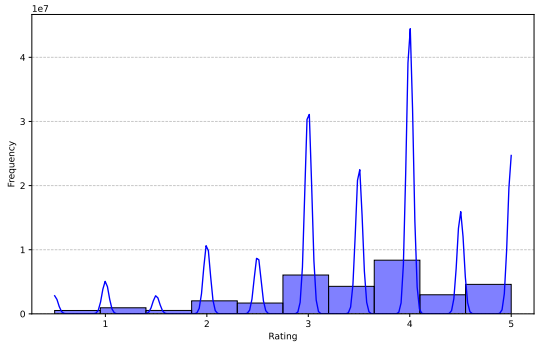

Figure 3: Distribution of Movie Ratings

2.2. Data Preprocessing

Dataset Characteristics: This is a sparse and high dimensional dataset, with the most important objective being fast retrieval and efficient structure in terms of clarity and use of memory. To handle this, I implemented a specialized data-structures and indexing maps for the ratings and genre information that ensure memory efficiency and computational speed.

Efficient Data Structures for User and Movie Data: The speed at which each data-point is retrieved and the type of data structure used to store it are the to main factors deciding the speed of the algorithm, I found Numpy object arrays to be the most efficient, and I created two primary data-structures:

- **User-Indexed Structure:** This array is organized by user index (instead of ID) for efficient access during user factor updates. Each entry contains a list of movies rated by the user, as well as corresponding ratings. For example, the first element corresponds to user index 0 and contains all the movies rated by that user.
- **Movie-Indexed Structure:** This array organizes ratings by movie index, with each entry representing a list of users who rated that specific movie, along with the ratings they provided.

These structures rely on two dictionaries—userId_to_index and movieId_to_index-which map original user and movie IDs to array indices, making data retrieval easy and efficient.

Genre Information Encoding and Mapping: The Genre information for each movie was provided as text, and since there is a limited number of genres, I encoded each one of them in a dictionary, each genre was assigned a unique index (Action = 0, Adventure = 1, etc...). Then, for each movie, a list of genres was retrieved from the metadata,

converted to genre indices, and stored in movie_genres. Using Numpy object arrays allowed varying length elements, as each movie could have a unique number of associated genres.

Advantages of Numpy Object Arrays: This data-structure allowed for a very fast retrieval of information, as there are libraries that can parallelize operations when the data is stored in the form of only Numpy arrays leading to enhanced performance, supporting quick updates of user and movie factors during model training. In addition, this method allowed for efficient memory usage as sparse data is stored compactly, reducing memory requirements. .

2.3. The Recommendation Problem:

If we can represent a movie with a latent vector Vn and a user with alatent vector Um, we can think of the recommendation problem as an estimation of the rating that user m would give to movie n, the simple likelihood [1].

$$p(r_{mn} \mid \mathbf{u}_m, \mathbf{v}_n) = \mathcal{N}\left(r_{mn}; \mathbf{u}_m^T \mathbf{v}_n +, \lambda^{-1}\right) \quad (1)$$

The Regularized Alternating Least Squares (ALS) method aims to minimize the error of the observed ratings while including regularization terms to avoid overfitting. The objective function for regularized ALS can be expressed as in equation [2]:

$$L(U,V) = \frac{\lambda}{2} \sum_{m=1}^{M} \sum_{n \in \Omega(m)} \left(r_{mn} - \mathbf{u}_m^T \mathbf{v}_n\right)^2 + \frac{\tau}{2} \left(\sum_{m=1}^{M} \|\mathbf{u}_m\|^2 + \sum_{n=1}^{N} \|\mathbf{v}_n\|^2\right) \quad (2)$$

Here:
- $\mathbf{u}_m$ and $\mathbf{v}_n$ are the user and item latent factor vectors, respectively.
- r_{mn} represents the observed rating of user m for item n.
- λ is the regularization parameter that penalizes the complexity of the latent vectors to prevent overfitting.
- τ regularizes the magnitude of the user and item factors.

The ALS method alternates between fixing the user factors to solve for item factors and vice versa, using a closed-form solution at each step. This process continues iteratively until convergence.

when trying to minimize this negative log likelihood, (ALS) is one of the most effective ways to do that, The ALS algorithm is particularly effective in handling the sparsity of user-movie interaction data by decomposing the ratings matrix into lower-dimensional latent factors.it is significantly faster then gradient descent, and handle the large scale of the data very well. In this work, I implemented a series of ALS models to perform collaborative filtering, below are the key models I implemented:

Bias-Only ALS Model:

The Bias-Only ALS model is a simplified variant that incorporates user and movies biases into the predictions without using latent factors. The model accounts for global trends and effects, such as average user ratings and item popularity. By introducing biases, the model can adjust for systematic differences in user behavior and item characteristics, improving the overall performance in situations with inherent rating tendencies. This model captures the inherent biases in user and item ratings without using any latent factors. It predicts the ratings based solely on user-specific and item-specific biases.

ALS Model With User and Movie Latent Factors: This model extending the Bias-Only model, this approach incorporates latent factors for both users and movies to better capture complex interactions between them. The predicted rating is now modeled as the dot product between the user and movie factors, in addition to the user and movie biases. This model improves upon the Bias-Only model by allowing for personalized recommendations. The regularized ALS algorithm optimizes the user and movie factors iteratively to minimize the error. the exact formula for updating users, movies and biases will be detailed in Appendex A. I also trained this model on the 25m and 32m dataset, delivering impressive resullts in terms of the quality of recommendations. details of the derivations of this models update equations are in appendex B.

ALS with Genre Features: Building upon the previous model, this version incorporates genre information as additional features. Each genre is represented by a latent factor vector, which influences the movie factor representation. The model assigns a set of genre factors to each movie, which are combined with the movie's latent factors. This allows for a more nuanced understanding of movie characteristics. The prediction now includes terms that account for the genre features, adding complexity but also potentially increasing recommendation accuracy by leveraging content-based information. Also more detailed derivation for the update equations for this model willbe in Appenex C.

3. Experiments Setup and Implementation:

I primarily utilized Google Colab for the initial development of this work. Subsequently, I transitioned to my local machine, which is equipped with 8 cores, and used VSCode to further enhance the speed and efficiency of the algorithm.

Train Test Split: The first phase of the implementation involved preparing the data for model training. In this step, a randomized approach was adopted to partition 10% of the dataset into training and testing sets. Specifically, I iterated through the 'ratings.csv' file row by row, and for each row, I used a random variable to determine the assignment. If the random variable's value was less than 0.9, the data point was allocated to the training set; otherwise, it was placed in the test set.

Before partitioning, I preprocessed the 'ratings' and 'movies' DataFrames to ensure that each movie was associated with a unique index, which remained consistent even if the movie or user appeared only in the test set. This was accomplished by first creating a mapping between each movie and a unique index, as well as a similar mapping for users. I then used these mappings to add additional columns to the DataFrame, allowing the indexing to be preserved throughout the data splitting process.

Bias-Only Model Implementation

I implemented my initial algorithm, the Bias-Only ALS model, as described in Algorithm 1. This model was trained on the 100K, 25M, and 32M datasets. To determine the optimal parameters λ and γ, I conducted a grid search. The best parameter set was found to be $\lambda = 1$ and $\gamma = 0.01$.

When applied to the 32M dataset, this same parameter configuration exhibited efficient convergence behavior. However, given that the model only needed to learn two variables, there was limited complexity to capture, causing the algorithm to converge within two or three epochs. The Root Mean Square Error (RMSE) metric was used to evaluate the performance and the goal was to minimize the error between the observed and predicted ratings.

Algorithm 1 shows the working of bias only ALS:

Algorithm 1 Biases Only ALS Model

Initialize $user\_biases = \text{np.zeros}(M)$
Initialize $item\_biases = \text{np.zeros}(N)$
repeat
  Loop over users:
    Update user bias
  Loop over items:
    Update item bias
until Convergence

I used the naive approach with only for loops which was enough to do many experiment using the small dataset, but it took a considerable amount of time when using the 32m dataset, so, it was clear that there was a need for more fast implementation of the algorithm.

Users and Movies with Bias Model

In the initial stage of my implementation, I developed the bias-only Alternating Least Squares (ALS) algorithm, as illustrated in Algorithm 1. This model was trained using datasets of varying sizes, specifically the 100k, 25M, and 32M MovieLens datasets. Through grid search, I determined the optimal regularization parameters to be $\lambda = 1$ and $\gamma = 0.01$. The bias-only model, despite its simplicity, demonstrated rapid convergence, completing in just two to three epochs due to the limited complexity of the parameters involved.

Subsequently, I extended the model to incorporate user and movie latent trait vectors, adopting a parallelized approach to update these factors alternately, as described in Algorithm 2. To enhance computational efficiency, I utilized the Numba library, leveraging its Just-In-Time (JIT) compilation capabilities for optimized numpy operations. This optimization significantly reduced the training time from 15 minutes per epoch to just 13 seconds on the 32M dataset, allowing for an extensive and efficient hyperparameter search.

In this advanced model, I found that setting the dimensionality of the latent vectors to 25 or 30 yielded the best results on the 32M dataset. Using a higher dimensionality risked overfitting, while lower dimensionalities negatively impacted the quality of recommendations. This balance ensured the model's robustness and maintained a high standard of predictive accuracy. The model was trained for 100 epochs, although there was no significant improvement in the RMSE, but the model still leaarns as the negative log likelihood decreases and model predictions become more accurate with more epochs.

The Algorithm 2 pseudo code below shows how ALS works by alternating between the users, movies and biases updates until convergence

Algorithm 2 Structure of ALS Updates For Users, Movies and Biases

repeat
  Loop over users in parallel:
    Update user bias
    Update user trait vector $\mathbf{u}$
  Loop over Movies in parallel:
    Update Movie bias
    Update Movie trait vector $\mathbf{v}$
until Convergence

Model with added Features

In the final stage of my project, I implemented the ALS model by incorporating genre-based feature vectors, making it a more robust and accurate in predictions. This model, built upon the previously described user and movie factors,

further integrated genre information to capture additional contextual relationships, as illustrated in Algorithm **??**.

To represent genres effectively, I initialized a set of latent vectors for each of the 19 possible genres. For every movie, I associated its corresponding genres using a one-dimensional array. This allowed the model to account for genre influences in the recommendation process. During training, I alternately updated the user, movie, and genre factors in parallel, leveraging the efficient Numba library for Just-In-Time (JIT) compilation. This optimization not only maintained the reduced training time but also facilitated efficient exploration of the parameter space enabling for training to go for 100 Epochs.

Extensive grid search revealed that the inclusion of genre features improved the model's predictive accuracy without significantly increasing the risk of overfitting. The best configuration involved latent vectors of 25 dimensions, similar to the user and movie factors. This comprehensive approach ensured that the model delivered high-quality, personalized recommendations while effectively capturing the complexity inherent in user-movie-genre interactions. Algorithm 3 Psudo code illustrates how ALS works by updating a set of factors and features :

Algorithm 3 ALS with User, Item, and Feature Factors

Initialize user biases $b^{(u)}$, item biases $b^{(i)}$, user vectors $\mathbf{u}$, item vectors $\mathbf{v}$, and feature vectors $\mathbf{f}$
repeat
  for each user m in 0 to $M-1$ **do**
    Update user bias $b_m^{(u)}$
    Update user vector $\mathbf{u}_m$
  end for
  for each item n in 0 to $N-1$ **do**
    Update item bias $b_n^{(i)}$
    Update item vector $\mathbf{v}_n$
  end for
  for each feature i in 0 to num_features-1 **do**
    Update feature vector $\mathbf{f}_i$
  end for
until convergence

4. Results

This section discusses the outcomes of our experiments, as we will discuss the results of each of the three models as the primary objective was to enhance the quality of personalized movie recommendations while maintaining computational efficiency. Tto assess the performance of these models. The Root Mean Square Error (RMSE) was the primary metric for quantifying the accuracy of our models' predictions compared to the actual ratings. A lower RMSE value indicates higher accuracy in matching user preferences. The RMSE formula we used is as follows:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{u=1}^{U}\sum_{i=1}^{n_u}\left(r_{ui} - (\mathbf{v}_i \cdot \mathbf{u}_u + b_u + b_i)\right)^2}$$

where:
- $\mathbf{v}_i$ is the movie's latent factor vector.
- $\mathbf{u}_u$ is the user's latent factor vector.
- b_u is the user's bias term.
- b_i is the movie's bias term.

In case of bias only models, we just set U.V to zero
This equation reflects the process of computing the squared errors between the actual and predicted ratings, averaging them, and taking the square root to get the RMSE. This formula computes the squared differences between the actual ratings and the predicted ratings (using only the bias terms for users and movies), averages them over all ratings, and then takes the square root to obtain the RMSE. Let me know if you need further clarification or adjustments! The following sections detail the performance of each model, highlighting how the inclusion of trait vectors and additional features impacted both the recommendation quality. Visual representations, including convergence plots and RMSE comparisons, are provided to illustrate the models' performance differences clearly.

4.1. Bias-Only Model

The key insight in the bias only model is that they tend to converge (or diverge) very fast, it usually takes two or three epochs to reach a minima, wh see similar pattern in both results of the 100k dataset and the 32m dataset. figure 4 and 5 illustrate the performance of the model on both training and test set for different datasets.

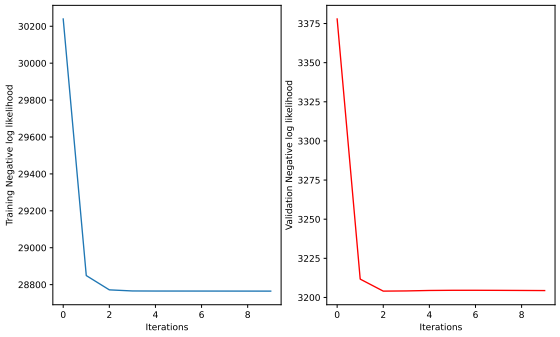

Figure 4: Figure shows the loss or the negative likelihood of the 100k bias only model

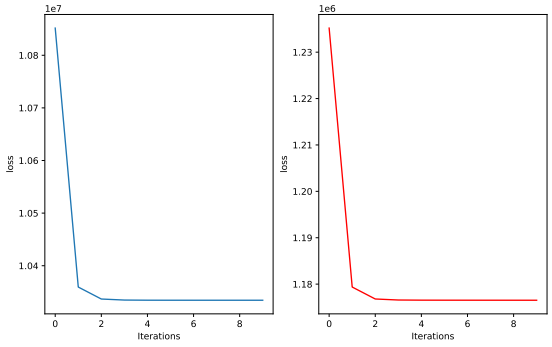

Figure 5: Figure shows the loss or the negative likelihood of the 32m bias only model

As seen in the figures, both models are showing a monotonic decrease in both training and validation losses, and converges after only two iterations.

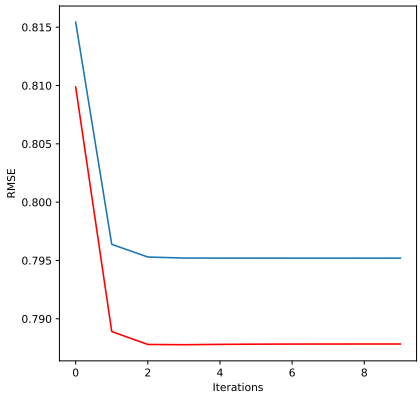

Figure 6: Figure shows the RMSE 100k bias only model

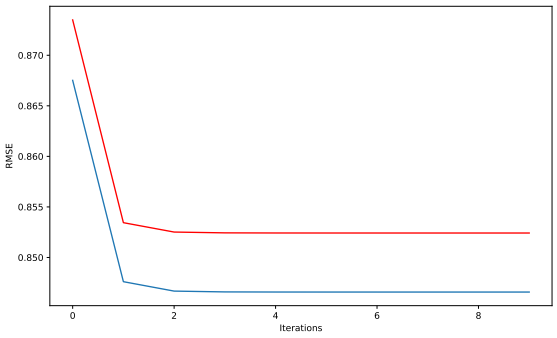

Figure 7: Figure shows the RMSE for the 32m bias only model

The RMSE evaluation, follows the same patterns.

4.2. Users and Movies with Bias Model

For this Model , the most relevant Experiments where done on the 32m dataset, In which we achieved an impressive 0.7776 RMSE on validation set, coupled with qualitatively great recommendations. The model was trained for 100 epochs, with a configuration of $\lambda = 1$ and $\gamma = 0.001$ and τ= 10, and latent factor vector of dimension of 30. figure **??** shows the negative log likelihood of the model decreasing monotonically as training continue.

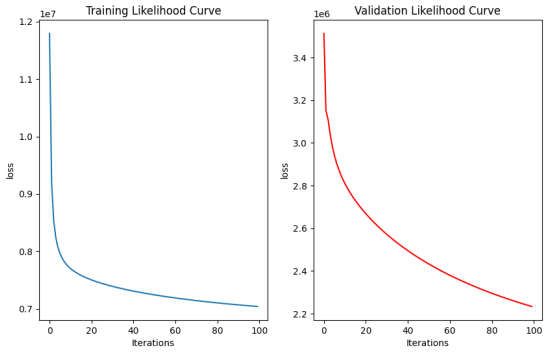

Figure 8: Figure shows the loss or the negative likelihood of the 32m model

and as for the RMSE for the training and test sets, figure 9 illustrates the decrease of the rmse over time till convergence.

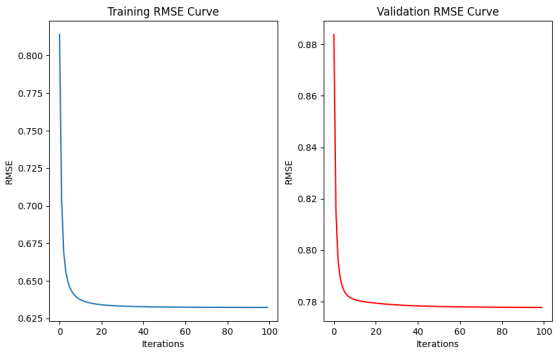

Figure 9: Figure shows the RMSE of the 32m model

Users and Movies with Bias Model use case with a fake user To qualitatively evaluate this model, I created a dummy user which only rated one movie, I infered the user vector U for this user along with his bias, and multiplied this latent vector with every movie in the latent movie factors V, and sorted out the results of multiplying each movie with this user in decsending order to see which movies ill be recom-

mended for this user. an Example of inference of the model is given in the following Tables.

Title	Genre
Toy Story 2 (1999)	Adventure—Animation—Children—Comedy—Fantasy
Monsters, Inc. (2001)	Adventure—Animation—Children—Comedy—Fantasy
Finding Nemo (2003)	Adventure—Animation—Children—Comedy
Toy Story 3 (2010)	Adventure—Animation—Children—Comedy—Fantasy—IMAX
Incredibles, The (2004)	Action—Adventure—Animation—Children—Comedy
Bug's Life, A (1998)	Adventure—Animation—Children—Comedy
Aladdin (1992)	Adventure—Animation—Children—Comedy—Musical
Lion King, The (1994)	Adventure—Animation—Children—Drama—Musical—IMAX
Shrek (2001)	Adventure—Animation—Children—Comedy—Fantasy
Up (2009)	Adventure—Animation—Children—Drama
Beauty and the Beast (1991)	Animation—Children—Fantasy—Musical—Romance—IMAX

Table 1: User Liked Toy Story 1, This is the Top 10 Recommendations according to the 32m Model

Title	Genre
Saw II (2005)	Horror—Thriller
Saw (2003)	Crime—Horror
Saw III (2006)	Crime—Horror—Thriller
Ring, The (2002)	Horror—Mystery—Thriller
Saw IV (2007)	Crime—Horror—Thriller
Scream (1996)	Comedy—Horror—Mystery—Thriller
Seven (a.k.a. Se7en) (1995)	Mystery—Thriller
Requiem for a Dream (2000)	Drama
Halloween (1978)	Horror
Nightmare on Elm Street, A (1984)	Horror—Thriller

Table 2: List of Recommendations for User who liked "Saw" Movie and Their Genres

Visualizing Movies Embedding in a 2D Space

References

Harper, F. M. and Konstan, J. A. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

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

Lü, L., Medo, M., Yeung, C. H., Zhang, Y.-C., Zhang, Z.-K., and Zhou, T. Recommender systems. *Physics reports*, 519(1):1–49, 2012.

A. You *can* have an appendix here.