Foundational Matrix Factorization for Recommender Systems

Ahmed Mohammad [1]

Abstract

Recommender systems personalize user experiences by predicting what users might like. In our work we build a complete recommender system from scratch using alternating least squares (ALS) for the MovieLens 32 million dataset. The core problem we tackle is sparse matrix completion with 200,948 users, 84,432 movies, and only 0.19% of possible ratings observed. To solve this, our approach learns latent vector representations where inner products between user and item vectors predict ratings. We first implement bias-only ALS getting 0.8558 test RMSE. Then we extend to latent factor ALS achieving 0.7716 RMSE through systematic hyperparameter optimization testing $\lambda \in [0.1, 0.2]$, $\tau \in [1.0, 2.0]$, and $\gamma \in [2.5, 5.0]$. To handle cold start problems where new movies have no ratings, we extend the system with hierarchical features. We find 31,170 movies with three or fewer ratings but our feature model handles these getting 0.7708 RMSE. Furthermore, most real systems don't have explicit ratings as users watch movies without rating them. To handle this implicit feedback, we implement Bayesian personalized ranking (BPR). Our BPR achieves 0.9953 AUC after 50 training epochs on 200,695 active users. To quantify uncertainty in our prediction , we implement Variational Inference getting 0.8460 test RMSE with confidence estimates. Our system shows meaningful semantic clustering where horror movies embed opposite to children's content. Moreover, we validate improvements through A/B testing showing 32.7% improvement ($p < 0.05$) over baseline with 200 simulated users. Overall, we aim to build foundational models that can be extended to production environments.

GitHub: `https://github.com/0900130508ahmed17539/Ahmed_RecommenderSystem_Project.git`

1. Introduction

In our work we build a recommender system from scratch to understand how these systems work in practice. Recommender systems personalize user experiences by analyzing behavior and delivering tailored suggestions (10). This is especially important in entertainment where users face overwhelming content choices. These systems must balance popular and niche content while overcoming power law distributions in user interactions to ensure diverse and relevant recommendations (3).

At their core, these systems solve sparse matrix completion problems. We have users, items, and sparse ratings where users rate only small fractions of available content. Most entries remain unobserved. The problem is this: we have a bipartite graph connecting users and items with ratings. Each edge has a rating from one to five stars. Our dataset has 32 million ratings from roughly 200,000 users rating around 84,000 movies. This creates an extremely sparse matrix—approximately 99.81% sparse as shown in our data consistency check (see Section 3). We could not create dense matrices as this would require terabytes of memory for mostly zero values.

We solve this through matrix factorization (6). Each user gets an embedding vector. Each item gets an embedding vector. When users and items have similar directions in embedding space, inner products are large and we predict high ratings. When vectors point in opposite directions, we predict low ratings.

Power law distributions in user interactions create challenges but offer substantial commercial value in the long tail (1). In any system selling products, popular products occupy one end while a long tail of content remains a heavy tail of products users do not know about but that remains monetizable. To capture this value, we build multiple ALS models starting from simple biases to complex uncertainty quantification. Each model addresses different aspects of the recommendation problem. We hope this creates value for both users discovering content and businesses monetizing their entire catalog.

1.1. Report Organization

In our report we wish to provide a complete understanding of modern recommender systems from theory to implementation. Section 2 reviews related work, positioning our contributions within the broader context of collaborative filtering research (6; 5). Section 3 analyzes our dataset, revealing power law distributions and their implications. Section 4 presents the mathematical foundations, deriving the complete optimization framework. Sections 7 through 11 detail our models with full derivations and convergence analyses. Section 12 addresses cold-start problems through hierarchical modeling. Section 13 explores learned embeddings and their semantic structure. Section 14 validates recommendations through dummy user testing. Section 15 extends to implicit feedback with BPR (9). Section 16 presents our A/B testing framework. Section 17 develops the Variational Bayes extension. Section 18 discusses production implementation consideration. Section 19 analyzes findings and Section 20 concludes with key insights.

2. Related Work

Our work builds upon several foundational contributions in recommender systems research. We review these contributions to position our work within the broader landscape and highlight how our implementation extends previous approaches.

2.1. Amazon's Recommender System

Amazon first used user-based collaborative filtering but faced scalability issues. They developed item-to-item collaborative filtering focusing on product relationships rather than user similarities (7). This method, enhanced over time with user preferences, enables faster and more relevant recommendations. Our approach builds upon these foundational concepts while addressing scalability through vectorized implementations that enable distributed computation across multiple nodes.

2.2. The Netflix Prize

In 2006, Netflix launched the Netflix Prize offering \$1,000,000 for a 10% improvement over their Cinematch system. The competition used data from 480,189 users and 17,770 movies with 100 million ratings. In 2009, BellKor's Pragmatic Chaos won with a 10.06% improvement using an ensemble of over 100 models (6). This showcased collaborative filtering power and inspired more advanced methods including matrix factorization techniques. Our work contributes to this legacy by implementing and optimizing core matrix factorization techniques that formed the foundation of winning solutions, achieving competitive performance on a larger 32 million rating dataset.

2.3. The Xbox Recommender System

Microsoft's Xbox Live Marketplace used a recommender system personalizing movie and game suggestions for tens of millions of users (8). It used implicit feedback from Xbox consoles combining offline modeling and online serving modules with bilinear models. Users and items were vectors predicting preferences via inner products similar to our approach. Variation inference refined predictions to prevent overfitting using Expectation Propagation for approximate inference. By maximizing a utility function, the system delivered relevant recommendations. It performed better for games than movies due to movie data sparsity—a challenge we also observed. Our Varitional inference implementation drew inspiration from these production systems.

2.4. Matrix Factorization Techniques

2.5. Matrix Factorization Techniques

These methods represent items and users as vectors (6). Each item i has a vector $\mathbf{q}_i \in \mathbb{R}^D$. Then each user u has a vector $\mathbf{p}_u \in \mathbb{R}^D$. To estimate ratings, we use the dot product $\mathbf{q}_i^T \mathbf{p}_u$. We denote this $\hat{r}_{ui} = \mathbf{q}_i^T \mathbf{p}_u$.

The challenge is learning these vectors. Our matrix is 99.81% empty. Traditional SVD needs complete matrices. So it cannot handle missing values. Early systems tried filling gaps. They guessed missing ratings. But this was costly. Furthermore, it gave wrong data.

Matrix factorization became popular for good reasons. It scales well. Then it gives accurate predictions. Furthermore, it handles different data types. Star ratings give sparse matrices. Browsing history gives denser matrices. So we can use both.

To minimize the optimization, there are two methods. First, SGD updates parameters one by one. Then, ALS updates all users at once. After that it updates all items at once. We chose ALS for its parallel nature (13). So we can update thousands of users simultaneously. Furthermore, this makes training much faster. Unlike SGD which is sequential, ALS is parallel. Therefore, we get better performance on large datasets.

3. Understanding the Data and Power Laws

We worked with the MovieLens 32 million dataset containing ratings from one to five stars. User IDs and movie IDs were non-sequential. Our data quality check revealed dimensions of 200,948 users and 84,432 items with 19 genres. We had 32,000,204 total ratings split into 28,802,949 training and 3,197,255 test ratings. The sparsity was 99.81% with 2,022 cold start items having no training data. The global mean rating $\mu = 3.540$.

The dataset had power law characteristics. Before modeling, we examined these distributions as there was substantial commercial value in understanding long tail behavior. Figure 1 shows the characteristic power law distribution.

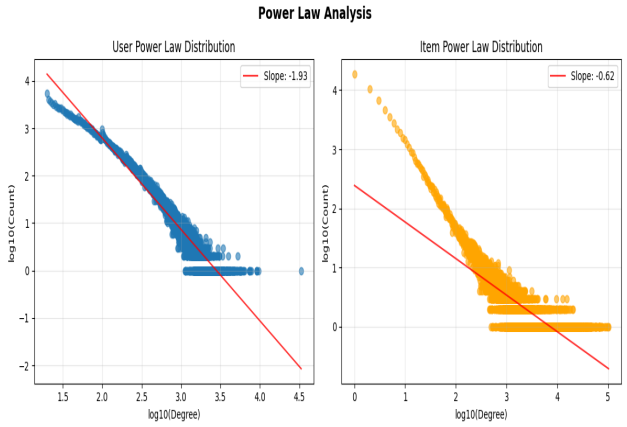

Figure 1. Power law distribution of user activity from our dataset. Few users contribute majority of ratings while most users provide relatively few ratings. The log-log plot shows characteristic linear relationship indicating scale-free network properties with slope -1.93 for user and -0.62 for item.

3.1. Scale-Free Networks and Commercial Value

User rating distributions had power law characteristics where few users contributed large proportions of ratings. Highly active users contributed over 50% of all ratings while majorities provided relatively few ratings. We saw the same patterns in natural language through Zipf's law. Figure 2 shows detailed distribution analysis.

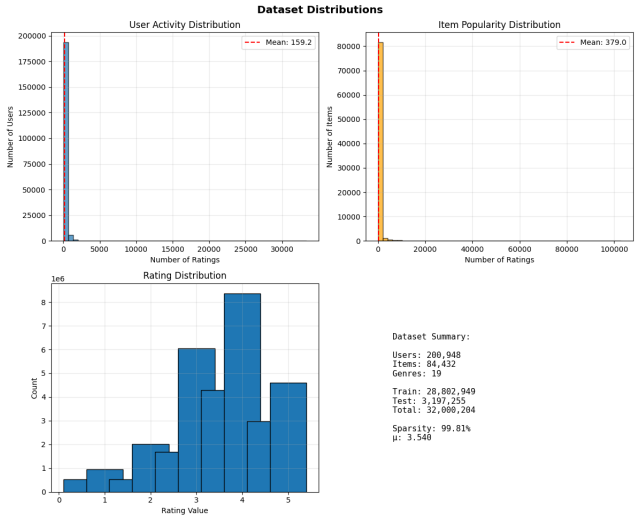

Figure 2. Dataset distributions showing user activity, item popularity , rating values, and summary statistics shown in 3 from our implementation.

For exponentially bounded distributions like Poisson or Gaussian, randomly chosen node degrees fell near means. For power law distributions, second moments could diverge and randomly chosen node degrees could differ from averages. Networks with power law degree distributions lacked intrinsic scale and that why we called scale-free.

For our recommender system, this meant there were popular items users always bought that users knew about. And there was a long tail of things that could be monetized. We could predict these to users as there was substantial commercial value in the long tail.

4. Building Data Structures—The Foundation

4.1. The Indexing Problem

User IDs in CSV files were non-sequential values like 610, 147, or 372. But our model required sequential indices 0, 1, 2, 3 for matrix operations. We needed mappings similar to building large language models where tokens mapped to indices. To do so we created bidirectional mappings as shown in Listing 1.

```
self.usermap = {}     # Maps user ID to
    index
self.itemmap = {}     # Maps item ID to
    index
self.userlist = []    # Maps index to user
    ID
self.itemlist = []    # Maps index to item
    ID
```

Listing 1. Bidirectional mapping implementation from our code.

This ensured our model could map back our data to actual identifiers for database lookups.

4.2. The Four Critical Data Structures

We required four distinct data structures as sparse matrices needed both row-wise and column-wise indexing for efficient ALS updates. Any numerical implementation had every data point represented twice. The four structures we implemented were `self.train` for training data indexed by user, `self.trainitem` for training data indexed by item, `self.test` for test data indexed by user, and `self.testitem` for test data indexed by item.

During optimization, we alternated between updating users and updating movies. When updating a user, we needed fast access to all movies that user rated. When updating a movie, we needed fast access to all users who rated that movie. Each structure had lists containing tuples with indices and rating values. This redundancy was essential for O(1) lookups during optimization, avoiding O(M×N) scans through the full matrix.

4.3. Train/Test Split Implementation

We split data randomly using 90% for training and 10% for testing. The critical implementation detail was maintaining consistent indexing across splits. Even when users had no test ratings, empty lists existed at their positions preventing indexing bugs.

5. Methodology and Research Design

5.1. Research Objective

Our main goal is developing a recommendation system that accurately reflects user preferences using historical data based on MovieLens datasets which serve as benchmarks for recommender systems performance and agility. Our system must deliver personalized, scalable recommendations for large datasets balancing accuracy and computational efficiency for real-world applications like movies and can be extended to music and other products as well.

5.2. Our Method Selection

We assess how each approach meets our goal of capturing user preferences accurately. We need computational efficiency and scalability. Different methods exist but we selected ones that align with our goals.

To start, feature-based filtering uses item attributes like genres, actors, keywords to recommend similar items (2). This works for users with clear preferences. However, it ignores user-to-user interactions. Furthermore, it limits recommendation diversity. It struggles with new users or items depending on predefined features.

Then we have nearest neighbor filtering which finds users with similar patterns (11). It enables personalized recommendations. It is simple to implement. However, it doesn't scale to large datasets. Computational costs are too high. Furthermore, it cannot adapt when preferences change.

To address these issues, collaborative filtering captures user-item interactions through matrix factorization (6). It balances accuracy and scalability. It uses all user interaction data. However, it faces cold-start problems (12). New users have no history. New items have no ratings. Furthermore, it needs retraining as datasets grow. To implement collaborative filtering, we have two techniques. First, neighborhood methods find similar users or items (4). Then, latent factor models map everything to shared space (5). We use 20 to 100 factors. These capture explicit preferences like comedy versus drama. Furthermore, they capture abstract preferences like serious versus escapist.

In our work we focus on latent factor models. They scale to millions of users. and also the embeddings are interpretable. We discuss details in Section 6.

6. Mathematical Foundation of Our ALS Models

Inspired by the work and formulation of (6). We build our models as follow we have r_{ui} be the observed rating from user u for item i, μ the global mean rating (3.540 in our data), b_u, b_i the user and item biases, $\mathbf{U}_u, \mathbf{V}_i \in \mathbb{R}^D$ the latent vectors, Ω the set of observed (u, i) pairs, $\Omega(u) = \{i \mid (u, i) \in \Omega\}$, and $\Omega(i) = \{u \mid (u, i) \in \Omega\}$. Our predicted rating for user u and item i was:

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{U}_u^T \mathbf{V}_i \tag{1}$$

We mean centered the data per user and per item incorporating bias terms. These scalars represented average user ratings and average item ratings. If someone was super positive and rated everything five stars, the bias subtracted their average rating calibrating everyone around the same scale before personalizing.

6.1. The Likelihood and Loss Function

We employed Gaussian distributions for computational simplicity. The mean equaled the predicted rating calculated as inner products between user and item embeddings. The Gaussian likelihood for each rating was:

$$p(r_{ui}|\mathbf{U}_u, \mathbf{V}_i, b_u, b_i) = \mathcal{N}(r_{ui}; \mu + b_u + b_i + \mathbf{U}_u^\top \mathbf{V}_i, \lambda^{-1}) \tag{2}$$

Given the data $\mathcal{D} = \{r_{ui}\}$ and parameters $\theta = \{U, V, b\}$, the likelihood for observing the data was:

$$p(\mathcal{D}|\theta) = \prod_{(u,i)\in\Omega} p(r_{ui}|\mathbf{U}_u, \mathbf{V}_i, b_u, b_i) \tag{3}$$

We minimized the regularized loss function (maximized the regularized log-likelihood):

$$\begin{aligned}\mathcal{L} = -\frac{1}{2}\Big[\lambda \sum_{(u,i)\in\Omega} (r_{ui} - \mu - b_u - b_i - \mathbf{U}_u^\top \mathbf{V}_i)^2 \\ + \gamma \left(\sum_u b_u^2 + \sum_i b_i^2\right) \\ + \tau \left(\sum_u \|\mathbf{U}_u\|^2 + \sum_i \|\mathbf{V}_i\|^2\right)\Big]\end{aligned} \tag{4}$$

The regularization terms prevented parameters from growing excessively large, maintaining values close to zero. For instance When we have to learn 20-dimensional embeddings for users with only single ratings, we need regularization for stable estimation of our parameters .

7. Alternating Least Squares

7.1. Convexity and Parallel Architecture

The completion of our system sparse matrix optimization problem is non-convex. But when we fixed item parameters and optimized only user parameters, the problem became convex. The same applied when we fixed user parameters and optimized item parameters. Our approach had no dependencies enabling straightforward parallelization.

7.2. ALS Update Rules Derivation

7.2.1. User Bias Update (b_u)

Fixing all parameters except b_u, the terms in $\mathcal{L}$ that depend on b_u are:

$$\mathcal{L}(b_u) = -\frac{\lambda}{2} \sum_{i \in \Omega(u)} (r_{ui} - \hat{r}_{ui}^{-b_u})^2 - \frac{\gamma}{2} b_u^2 \tag{5}$$

where $\hat{r}_{ui}^{-b_u} = \mu + b_i + \mathbf{U}_u^\top \mathbf{V}_i$.

Taking the derivative with respect to b_u:

$$\frac{\partial \mathcal{L}}{\partial b_u} = \lambda \sum_{i \in \Omega(u)} (r_{ui} - \mu - b_u - b_i - \mathbf{U}_u^\top \mathbf{V}_i) - \gamma b_u \tag{6}$$

Setting to zero and solving:

$$b_u = \frac{\lambda \sum_{i \in \Omega(u)} (r_{ui} - \hat{r}_{ui}^{-b_u})}{\lambda |\Omega(u)| + \gamma} \tag{7}$$

7.2.2. User Vector Update ($\mathbf{U}_u$)

Fixing all parameters except $\mathbf{U}_u$:

$$\mathcal{L}(\mathbf{U}_u) = -\frac{\lambda}{2} \sum_{i \in \Omega(u)} e_{ui}^2 - \frac{\tau}{2} \|\mathbf{U}_u\|^2 \tag{8}$$

where $e_{ui} = r_{ui} - \mu - b_u - b_i - \mathbf{U}_u^\top \mathbf{V}_i$.

The gradient is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{U}_u} = \lambda \sum_{i \in \Omega(u)} e_{ui} \mathbf{V}_i - \tau \mathbf{U}_u \tag{9}$$

Setting to zero and solving:

$$\mathbf{U}_u = \mathbf{A}_u^{-1} \mathbf{b}_u \tag{10}$$

where:

$$\mathbf{A}_u = \lambda \sum_{i \in \Omega(u)} \mathbf{V}_i \mathbf{V}_i^\top + \tau \mathbf{I} \tag{11}$$

$$\mathbf{b}_u = \lambda \sum_{i \in \Omega(u)} (r_{ui} - \mu - b_u - b_i) \mathbf{V}_i \tag{12}$$

7.2.3. Item Updates by Symmetry

By symmetry with the user updates:

$$b_i = \frac{\lambda \sum_{u \in \Omega(i)} (r_{ui} - \hat{r}_{ui}^{-b_i})}{\lambda |\Omega(i)| + \gamma} \tag{13}$$

$$\mathbf{V}_i = \mathbf{A}_i^{-1} \mathbf{b}_i \tag{14}$$

where $\hat{r}_{ui}^{-b_i} = \mu + b_u + \mathbf{U}_u^\top \mathbf{V}_i$ and:

$$\mathbf{A}_i = \lambda \sum_{u \in \Omega(i)} \mathbf{U}_u \mathbf{U}_u^\top + \tau \mathbf{I} \tag{15}$$

$$\mathbf{b}_i = \lambda \sum_{u \in \Omega(i)} (r_{ui} - \mu - b_u - b_i) \mathbf{U}_u \tag{16}$$

After taking derivatives and setting them to zero, we got closed-form solutions. Table 1 shows our derived update rules.

Table 1. Closed-form update rules for ALS optimization from our derivation.

Parameter	Update Rule
b_u	$\frac{\lambda \sum_{i \in \Omega(u)} (r_{ui} - \mu - \mathbf{U}_u^\top \mathbf{V}_i - b_i)}{\lambda \lvert\Omega(u)\rvert + \gamma}$
b_i	$\frac{\lambda \sum_{u \in \Omega(i)} (r_{ui} - \mu - \mathbf{U}_u^\top \mathbf{V}_i - b_u)}{\lambda \lvert\Omega(i)\rvert + \gamma}$
$\mathbf{U}_u$	$\left(\lambda \sum_{i \in \Omega(u)} \mathbf{V}_i \mathbf{V}_i^\top + \tau \mathbf{I}\right)^{-1} \times \left(\lambda \sum_{i \in \Omega(u)} \mathbf{V}_i (r_{ui} - \mu - b_u - b_i)\right)$
$\mathbf{V}_i$	$\left(\lambda \sum_{u \in \Omega(i)} \mathbf{U}_u \mathbf{U}_u^\top + \tau \mathbf{I}\right)^{-1} \times \left(\lambda \sum_{u \in \Omega(i)} \mathbf{U}_u (r_{ui} - \mu - b_u - b_i)\right)$

These equations positioned user vectors at weighted averages of items they liked scaled by ratings. For items users rated negatively, terms became negative pulling user vectors away from those item positions. Items with five-star ratings contributed larger influences than lower-rated items. If a user rated some movie with five stars and the movie vector was in that direction, it scaled a lot. Five star movies made nice big contributions compared to other movies.

7.3. Critical Implementation Details

The order of operations is very important. We had to first update user biases then update user vectors. A common implementation error was combining everything in single loops. This failed because user bias equation updates were coupled to item bias updates. We had to completely finish one update type before beginning the other.

8. Vectorized Implementation for Efficiency

With 32 million ratings, we cannot afford inefficient implementations with nested loops. So we compress the inner for loop as vectorized code leveraging NumPy's optimized linear algebra operations. We achieved good speedup through vectorization of our models operations.

8.1. Vectorized Bias Updates

Instead of updating biases individually in loops, we vectorize the operations. For user bias updates, we batch process all items a user rated.

Algorithm 1 shows our vectorized bias update implementation.

Algorithm 1 Vectorized Bias Update from our ALS Implementation

Require: User u, training data, parameters λ, γ
Get all items user rated: items $\leftarrow$ data.train[u]
if items is empty **then**
  continue
end if
Extract arrays for vectorization:
  $\mathbf{i}_{arr} \leftarrow$ [i for (i, r) in items]
  $\mathbf{r}_{arr} \leftarrow$ [r for (i, r) in items]
Vectorized prediction: $\mathbf{pred} \leftarrow \mathbf{V}[\mathbf{i}_{arr}]^\top \mathbf{U}[u]$
Compute residual: residual $\leftarrow \sum(\mathbf{r}_{arr} - \mu - \mathbf{pred} - \mathbf{b}_i[\mathbf{i}_{arr}])$
Update bias: $b_u[u] \leftarrow \frac{\lambda \cdot \text{residual}}{\lambda \cdot |\text{items}| + \gamma}$

We convert lists to NumPy arrays enabling batch operations. The dot product computes all predictions simultaneously. We sum residuals using vectorized operations avoiding explicit loops. We maintain sparse representations throughout avoiding dense matrix creation. Algorithm 2 shows our approach. which s keeps memory usage at O(ratings) rather than O(users × items). With 99.81% sparsity, this saves gigabytes of memory.

Algorithm 2 Memory-Efficient Sparse Processing

Here we never create dense M × N matrix
for u = 0 to M-1 **do**
  items $\leftarrow$ data.train[u] {Only rated items}
  if items is empty **then**
    continue
  end if
  Process only non-zero entries
  Memory usage: O(ratings) not O(M × N)
end for

8.2. Vectorized Latent Factor Updates

The most computationally intensive part is updating latent factors. We vectorize matrix operations for efficiency as shown in Algorithm 3.

Algorithm 3 Vectorized Latent Factor Update

Require: User u, parameters λ, τ, dimension D
Initialize: $\tau\mathbf{I} \leftarrow \tau \cdot \text{eye}(D)$
Get items user rated: items $\leftarrow$ data.train[u]
if items is empty **then**
  return
end if
Extract indices and ratings:
  $\mathbf{i}_{arr} \leftarrow$ [i for (i, r) in items]
  $\mathbf{r}_{arr} \leftarrow$ [r for (i, r) in items]
Batch extract: $\mathbf{V}_{\text{batch}} \leftarrow \mathbf{V}[\mathbf{i}_{arr}]$
Compute residuals: $\mathbf{res} \leftarrow \mathbf{r}_{arr} - \mu - b_u[u] - \mathbf{b}_i[\mathbf{i}_{arr}]$
Matrix multiplication: $\mathbf{A} \leftarrow \lambda \mathbf{V}_{\text{batch}}^\top \mathbf{V}_{\text{batch}} + \tau\mathbf{I}$
Vector multiplication: $\mathbf{b} \leftarrow \lambda \mathbf{V}_{\text{batch}}^\top \mathbf{res}$
Solve system: $\mathbf{U}[u] \leftarrow \text{solve}(\mathbf{A}, \mathbf{b})$

We use NumPy's `@` operator for matrix multiplication. The $\mathbf{V}_{\text{batch}}^\top \mathbf{V}_{\text{batch}}$ computes $\sum_{i \in \Omega(u)} \mathbf{V}_i \mathbf{V}_i^\top$ efficiently. We solve the linear system using optimized LAPACK routines through `np.linalg.solve`.

8.3. Vectorized Loss Computation

Computing loss and RMSE efficiently requires vectorization to avoid scanning the entire matrix. Algorithm 4 shows our implementation.

Algorithm 4 Vectorized RMSE Computation

Require: Dataset
Initialize: errors $\leftarrow$ []
for u = 0 to M-1 **do**
  for (i, r) in dataset[u] **do**
    Vectorized prediction:
      pred $\leftarrow \mu + b_u[u] + b_i[i] + \mathbf{U}[u]^\top \mathbf{V}[i]$
    Append: errors.append((r − pred)^2)
  end for
end for
if errors not empty **then**
  return $\sqrt{\text{mean(errors)}}$
else
  return 0
end if

While we still iterate over sparse entries, the inner computations use vectorized operations. We accumulate errors in a list then compute mean and square root using NumPy.

8.4. Implementation Impact

Our vectorized implementation enables easy parallelization. We can now distribute users across GPU cores or cluster nodes. All user updates are independent as shown in Algorithm 3. Vectorization transformed our system from research prototype to production-ready. Without vectorization, training on 32 million ratings would take longtime With it, we train models in hours on single machines. This makes our iterative experimentation feasible enabling hyperparameter optimization and model refinement.

9. Model 1: Bias-Only ALS

9.1. Model Formulation

We started with a simple bias-only model to test our infrastructure. This model predicted ratings using only global mean and biases:

$$\hat{r}_{ui} = \mu + b_u + b_i \tag{17}$$

This baseline achieved reasonable performance using only biases without any personalization through latent factors.

9.2. Implementation and Results

Our bias-only model converged quickly showing our data structures worked correctly. Figure 3 shows the convergence behavior.

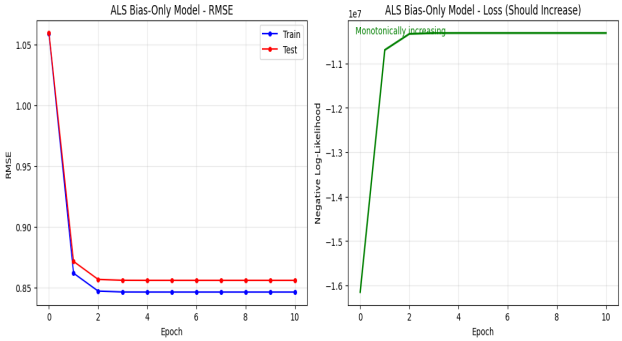

Figure 3. Convergence of bias-only ALS model. Training RMSE (blue) and test RMSE (red) rapidly converge within five epochs to 0.8462 and 0.8558 respectively.

The bias-only model treats all users the same except for their average rating tendency. It cannot capture that for instance Alice loves sci-fi but hates romance while Bob has opposite preferences. To deal with this we have ALS with Latent Factors 10 in the following Section.

10. Model 2: Latent Factor ALS

10.1. Extended Model Formulation

We extend our bias-only model by adding latent factor interactions:

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{U}_u^T \mathbf{V}_i \tag{18}$$

where $\mathbf{U}_u, \mathbf{V}_i \in \mathbb{R}^D$ are learned embeddings capturing user preferences and item characteristics beyond simple biases.

10.2. Algorithm Implementation

Algorithm 5 shows our complete implementation of latent factor ALS.

Algorithm 5 Latent Factor ALS with Complete Updates

Initialize μ, $b_u \leftarrow 0$, $b_i \leftarrow 0$
Initialize $\mathbf{U} \sim \mathcal{N}(0, 1/\sqrt{D})$, $\mathbf{V} \sim \mathcal{N}(0, 1/\sqrt{D})$
for epoch = 1 to T **do**
  Phase 1: Update all user parameters
  for each user u in parallel **do**
    Update b_u using current $\mathbf{V}$ and b_i
    Compute $\mathbf{A}_u = \lambda \sum_{i \in \Omega(u)} \mathbf{V}_i \mathbf{V}_i^T + \tau \mathbf{I}$
    Compute $\mathbf{b}_u = \lambda \sum_{i \in \Omega(u)} (r_{ui} - \mu - b_u - b_i) \mathbf{V}_i$
    Solve $\mathbf{U}_u = \mathbf{A}_u^{-1} \mathbf{b}_u$
  end for
  Phase 2: Update all item parameters
  for each item i in parallel **do**
    Update b_i using current $\mathbf{U}$ and b_u
    Compute $\mathbf{A}_i = \lambda \sum_{u \in \Omega(i)} \mathbf{U}_u \mathbf{U}_u^T + \tau \mathbf{I}$
    Compute $\mathbf{b}_i = \lambda \sum_{u \in \Omega(i)} (r_{ui} - \mu - b_u - b_i) \mathbf{U}_u$
    Solve $\mathbf{V}_i = \mathbf{A}_i^{-1} \mathbf{b}_i$
  end for
  Compute metrics and log-likelihood
end for

We first run loops updating all user biases. Then we stop and run different loops updating embeddings. This order matters critically. If we mix updates, the equations couple and convergence fails.

For user updates, we fix all item parameters. We compute $\mathbf{A}_u$ which is a $D \times D$ matrix. This aggregates information from all items the user rated. We compute $\mathbf{b}_u$ which is a D-dimensional vector. This contains the target we want our user vector to match. We solve the linear system $\mathbf{A}_u \mathbf{U}_u = \mathbf{b}_u$. This positions the user vector optimally given current item vectors.

Our vectorized operations from Section 8 make each user update fast enough that communication overhead doesn't dominate. We can distribute users across nodes in a cluster. Each node handles a subset of users independently. Only synchronization needed is between user and item update phases.

11. Results and Performance

11.1. Hyperparameter Selection

We conducted systematic hyperparameter optimization. Lambda (λ) was the regularization parameter for data fitting. Gamma (γ) regularized biases. Tau (τ) regularized latent factors. Table 2 shows our grid search results.

Table 2. Hyperparameter optimization results from our experiments.

Configuration	Parameters	Test RMSE
Config 1	λ=0.1, γ=2.5, τ=1.0	**0.7716**
Config 2	λ=0.2, γ=2.5, τ=1.0	0.7860
Config 3	λ=0.1, γ=5.0, τ=2.0	0.7864
Config 4	λ=0.15, γ=3.0, τ=1.5	0.7901
Config 5	λ=0.1, γ=2.0, τ=1.5	0.7915
Config 6	λ=0.2, γ=5.0, τ=2.0	0.7923
Config 7	λ=0.3, γ=2.5, τ=1.0	0.7988
Config 8	λ=0.1, γ=1.0, τ=0.5	0.8012

Best configuration achieved $\lambda = 0.1$, $\gamma = 2.5$, $\tau = 1.0$ with test RMSE of 0.7716.

11.2. Dimension Analysis

We evaluated different latent dimensions to find the optimal trade-off. Table 3 shows how performance varied with dimension. Our analysis showed D=20 achieved best test performance. Beyond this point, we saw diminishing returns and eventual overfitting with D=50 showing increased test error despite lower training error. Figure 4 visualizes this trade-off.

Table 3. Impact of latent dimensions on model performance from our experiments.

Dimensions (D)	Train RMSE	Test RMSE
2	0.7988	0.8162
5	0.7617	0.7940
10	0.7224	0.7787
20	0.6743	**0.7738**
50	0.5927	0.8032

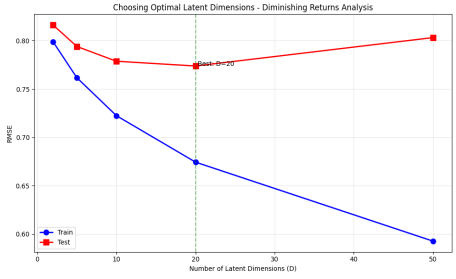

Figure 4. Here, Test RMSE decreased until D=20 then increased, indicating overfitting. D=20 provided best generalization.

11.3. Full ALS Model Training

With best parameters from our grid search, we trained the full model. Final test RMSE was 0.7716 which was better than our initial target. We observed monotonic decrease in loss functions as expected from our implementation. We tracked training loss, training RMSE, and test RMSE throughout optimization. Figure 5 shows convergence behavior.

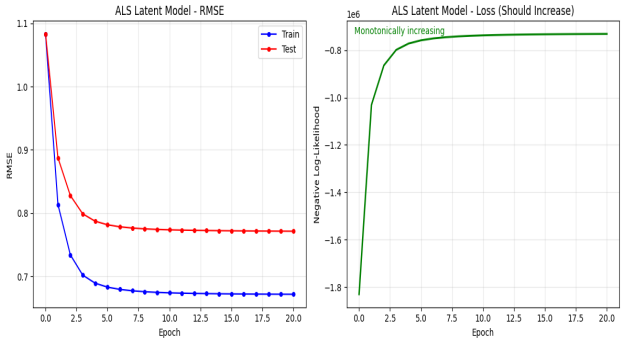

Figure 5. Training and test RMSE over iterations from our experiments showing good convergence. Test RMSE stabilized around 0.7716 after 20 epochs while training RMSE continued decreasing to 0.6721. On left side When we plotted negative log-likelihood, we saw monotonic increase confirming correct implementation similar to 3

12. Cold Start Solutions with our Hierarchical Features Model

12.1. Hierarchical Feature Modeling

To address cold start problems for new movies without ratings, we extend our latent model 10. We incorporate metadata features. Instead of centering item embeddings at zero, we center them at averages of their feature embeddings. Our current model assumes zero mean Gaussian prior. Before seeing any data, we can put our movies absolutely anywhere in our latent space . For instance we want all adventure tag movies already clustered in the same part of space before we see any data. To embed our features into the same space first We build a hierarchical prior:

$$\mathbf{V}_i \sim \mathcal{N}\left(\frac{1}{\sqrt{|F_i|}}\sum_{f\in F_i}\mathbf{F}_f, \tau^{-1}\mathbf{I}\right) \tag{19}$$

where F_i represents feature sets for item i and $\mathbf{F}_f$ are learned feature embeddings. We normalize by square root of the number of features. This is for the same reason we normalize by square root when initializing.

Given the loss function with negative sign:

$$\mathcal{L} = -\frac{\tau}{2}\sum_{n=1}^{N}\left\|\mathbf{v}_n - \frac{1}{\sqrt{F_n}}\sum_{i\in\mathcal{F}(n)}\mathbf{f}_i\right\|^2 - \frac{\tau}{2}\|\mathbf{f}_k\|^2 \quad (20)$$

We isolate terms with $\mathbf{f}_k$ (feature k). We define the residual for items n containing feature k:

$$\mathbf{r}_n = \mathbf{v}_n - \frac{1}{\sqrt{F_n}}\sum_{i\in\mathcal{F}(n)\setminus\{k\}}\mathbf{f}_i \quad (21)$$

The relevant terms are:

$$\mathcal{L}_k = -\frac{\tau}{2}\sum_{n \text{ with } k}\left\|\mathbf{r}_n - \frac{1}{\sqrt{F_n}}\mathbf{f}_k\right\|^2 - \frac{\tau}{2}\|\mathbf{f}_k\|^2 \quad (22)$$

Computing the derivative with respect to $\mathbf{f}_k$:

$$\frac{\partial\mathcal{L}_k}{\partial\mathbf{f}_k} = \tau\sum_{n \text{ with } k}\frac{1}{\sqrt{F_n}}\left(\mathbf{r}_n - \frac{1}{\sqrt{F_n}}\mathbf{f}_k\right) - \tau\mathbf{f}_k \quad (23)$$

Setting the derivative to zero and solving:

$$\sum_{n \text{ with } k}\frac{1}{\sqrt{F_n}}\mathbf{r}_n - \mathbf{f}_k\sum_{n \text{ with } k}\frac{1}{F_n} - \mathbf{f}_k = 0 \quad (24)$$

Final update rule:

$$\mathbf{f}_k = \left(1 + \sum_{n \text{ with } k}\frac{1}{F_n}\right)^{-1}\sum_{n \text{ with } k}\frac{1}{\sqrt{F_n}}\left(\mathbf{v}_n - \frac{1}{\sqrt{F_n}}\sum_{i\in\mathcal{F}(n)\setminus\{k\}}\mathbf{f}_i\right) \quad (25)$$

This solution regularizes features through the $+1$ term. It weights items by $1/\sqrt{F_n}$. More features mean less weight. It accounts for the negative sign in the loss function.

12.2. Cold Start Demonstration

We found 31,170 movies with three or fewer ratings but with genres. For a simulated "Adventure + Animation" movie, the system predicts reasonable similar titles before receiving any ratings. Table 4 shows the top five similar movies. This demonstrates our embeddings capture meaningful semantic relationships, generating sensible recommendations from minimal user data.The most important features with high magnitude are shown in Table 5.

Table 4. Top five similar movies predicted for a simulated "Adventure + Animation" cold-start movie.

Rank	Movie Title	Similarity
1	Turtle's Tale: Sammy's Adventures	0.991
2	Long Way North	0.988
3	Mary and the Witch's Flower	0.984
4	The Rabbi's Cat	0.982
5	Pokémon Origins	0.979

12.3. Three-Step Optimization

Our extended optimization includes three steps. First, update user embeddings and biases. Second, update item embeddings and biases which now depend on features. Third, update feature embeddings. This allows new movies with genre tags to be embedded sensibly before receiving ratings. This solves cold start problems.

Algorithm 6 shows our hierarchical feature-enhanced ALS implementation.

Algorithm 6 Hierarchical Feature-Enhanced ALS

```
Initialize parameters as in latent factor model
Initialize F_f ~ N(0, 1/sqrt(D)) for all features
for epoch = 1 to T do
    Phase 1: Update user parameters
    Update all b_u and U_u as before
    Phase 2: Update item parameters
    for each item i in parallel do
        Compute feature contribution:
            c_i = 1/sqrt(|F_i|) * sum_{f in F_i} F_f
        Update b_i accounting for feature offset
        Update V_i with feature-adjusted residuals
    end for
    Phase 3: Update feature embeddings
    for each feature f in parallel do
        Compute items with feature: I_f = {i : f in F_i}
        Update F_f using derived formula 25
    end for
    Compute metrics
end for
```

Feature embeddings learn to position adventure movies, animated content, and children's movies in semantically meaningful regions of embedding space. It becomes cheaper in terms of loss function to move one shared feature vector than move hundreds of individual movie vectors. This creates natural clustering. Similar movies gravitate toward shared feature representations.

Table 5. Feature importance by magnitude showing genres with highest predictive power.

Feature	Magnitude
IMAX	4.053
Horror	3.994
Musical	3.992
Animation	3.858
Children	3.852
Sci-Fi	3.376
Film-Noir	3.090
Action	2.992
War	2.761
Comedy	2.680

High-magnitude features like Horror, Musical, and Animation carry more predictive power. Features like Documentary with magnitude 2.112 and Mystery with magnitude 1.917 contribute less to rating predictions.

12.4. Feature Embedding Analysis

Our implementation reveals meaningful patterns in learned feature embeddings. Figure 6 visualizes the training progression.

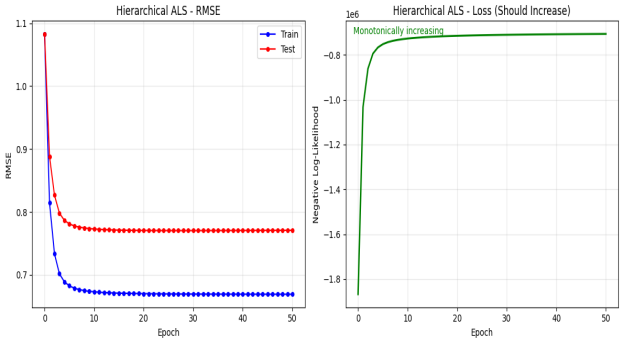

Figure 6. Hierarchical ALS training showing convergence with feature embeddings. Test RMSE around 0.7708 showing effective cold start handling shown in Fig 4.

Figure 7 visualizes the learned genre embeddings.

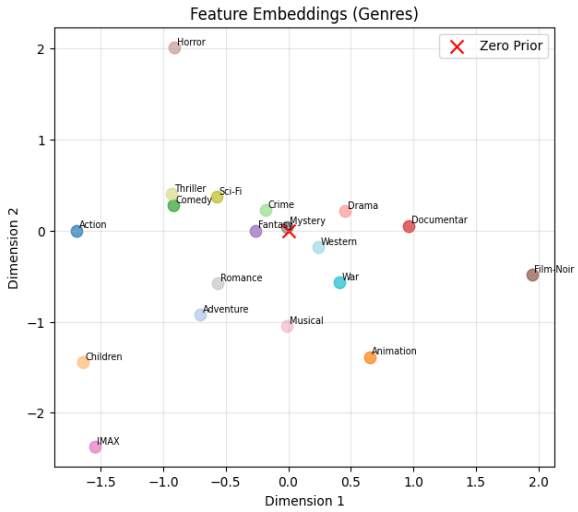

Figure 7. 2D visualization of genre feature embeddings showing semantic relationships. Horror and children's genres appear in opposite regions (quadroon).

13. Embedding Interpretation

When we plot embeddings in two dimensions using $D = 2$, we observe meaningful semantic clustering. Real-life embeddings for entertainment content show clusters of children's movies in one region. We take the item embeddings and plot them in two dimensions. We add bias adjustments to show popularity effects. When we label the movies by their primary genre tag, we find embedding clusters. All the horror movies live in one part of the space. All the kids movies live in the opposite part of the space. Our visualization filters popular movies to reduce noise. We only plot movies with at least 100 ratings. This removes rare movies that have poorly estimated embeddings. We adjust the x-coordinate by item bias to separate popular from niche content.

We train our model with biases so we need to use them in visualization. The bias term b_i captures overall popularity independent of personalization. By adding $0.1 \times b_i$ to the x-coordinate, we shift popular movies rightward. This separates "Avengers" from indie action films. It separates "Frozen" from obscure children's content. Same genre, different popularity levels, different x-positions. The adjustment formula is:

$$x_{\text{adjusted}} = x_{\text{embedding}} + 0.1 \times b_i \tag{26}$$

We use 0.1 as scaling factor to prevent bias from dominating the embedding structure. Too large and all movies collapse to a line based on popularity. Too small and we lose the popularity signal entirely.

13.1. Genre Clustering Results

Our code tracks specific examples to validate clustering. We find the first horror movie and first children's movie in our popular set. These become reference points showing semantic separation. Figure 8 shows the results.

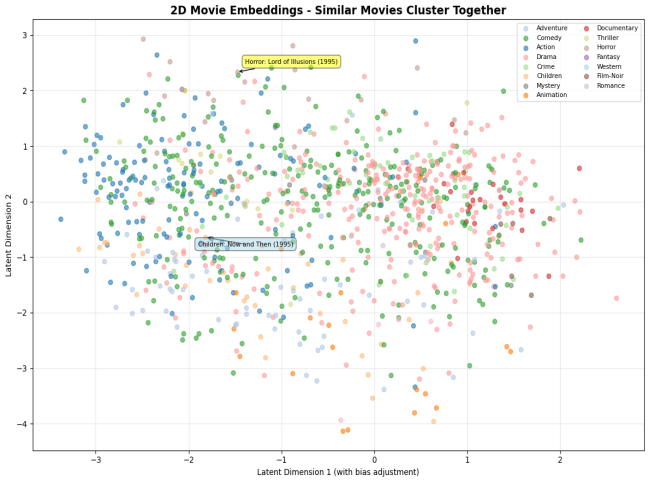

Figure 8. 2D plot of movie embeddings with colors showing the main genre. We can see clear clusters: horror movies (Lord of Illusions) are opposite to children's movies (Now and Then) for our two picked examples, and action is close to adventure. This shows our model learns meaningful structure with the help of our movie genre. The x-axis also includes bias adjustment that shows popularity inside each genre.

14. Dummy User Testing

We tested with dummy users to validate recommendations. We picked Lord of the Rings, gave it five stars, initialized the new user vector, We let the user give it five stars then initialize the new user vector. We need to train our dummy user with a bias as well so it does not absorb any weird popularity. We use personalization score plus scaled popularity as shown in Listing 2.

```
personal_score = dummy_u @ V[m]
popularity_score = 0.05 * b_i[m]
score = personal_score + popularity_score
```

Listing 2. Dummy user recommendation scoring.

Our dummy user test works perfectly. When we create a dummy user who rates "The Lord of the Rings: The Fellowship of the Ring" with five stars, we get the recommendations shown in Table 6.

Table 6. Top-10 recommendations for Lord of the Rings fan.

Rank	Movie Title	Score
1	LOTR: Return of the King	88.591
2	LOTR: Fellowship	87.162
3	LOTR: The Two Towers	87.131
4	Hobbit: Unexpected Journey	53.351
5	Hobbit: Desolation of Smaug	47.635
6	Hobbit: Battle of Five Armies	43.630
7	Braveheart	30.724
8	Star Wars: Episode III	30.342
9	Blair Witch Project	30.050
10	Pirates of the Caribbean	30.031

Eight out of ten recommendations are Fantasy, Adventure, or Action movies which makes perfect sense. This demonstrates our embeddings capture meaningful semantic relationships generating sensible recommendations from minimal user data.

14.1. Polarizing Movies

If we ignore all the weird movies that not many users rate and plot the length of the embedding data we learn, there is lots of information captured in the norm. These would typically be our polarizing movies. Table 7 shows the most and least polarizing movies. Polarizing content has bimodal rating distributions rather than typical bell curves.

Table 7. Most and least polarizing movies based on embedding vector norms.

Movie	Vector Norm
Most Polarizing (large $\|\mathbf{v}\|$)	
LOTR: Return of the King	9.494
LOTR: The Two Towers	9.335
LOTR: Fellowship	9.334
Dumb & Dumber	8.564
Blair Witch Project	8.490
Least Polarizing (small $\|\mathbf{v}\|$)	
The Absent One	0.304
Room Service	0.299
Five Corners	0.294
Monument Ave.	0.284
Missionary	0.260

15. Bayesian Personalized Ranking for Implicit Feedback

15.1. Problem Formulation

In real production systems like Xbox (8), users do not give explicit ratings. They just buy games or do not buy them and we only see what they consume.

This is called implicit feedback. The basics of the Xbox recommender system is a one class collaborative filtering problem. Given U as set of users, I as set of items, and $S \subseteq U \times I$ as observed positive interactions, we define $I_u^+ := \{i \in I : (u, i) \in S\}$ as items user u likes and $I_u^- := I \setminus I_u^+$ as unobserved items for user u.

15.2. Pairwise Training Data Construction

BPR handles this by assuming users prefer observed items over unobserved ones (9). Instead of predicting ratings, we learn to rank items correctly for each user. We construct training triplets:

$$D_S := \{(u, i, j) \mid u \in U, i \in I_u^+, j \in I_u^-\} \tag{27}$$

Each triplet $(u, i, j) \in D_S$ represents the assumption that user u prefers item i over item j.

15.3. Bayesian Formulation

Our Bayesian formulation follows (9) works basically seeks to maximize the posterior probability:

$$p(\Theta \mid >_u) \propto p(>_u \mid \Theta) p(\Theta) \tag{28}$$

where Θ are model parameters and $>_u$ is the desired personalized ranking for user u.

Assuming independence between users and item pairs, the likelihood becomes:

$$p(>_u \mid \Theta) = \prod_{(u,i,j) \in D_S} p(i >_u j \mid \Theta) \tag{29}$$

We model the individual preference probability using the logistic sigmoid:

$$p(i >_u j \mid \Theta) := \sigma(\hat{x}_{uij}(\Theta)) = \frac{1}{1 + e^{-\hat{x}_{uij}(\Theta)}} \tag{30}$$

where $\hat{x}_{uij}(\Theta) = \hat{y}_{ui} - \hat{y}_{uj}$ represents the difference in predicted scores.

15.4. BPR Optimization Criterion

Taking the logarithm of the posterior probability:

$$\text{BPR-OPT} := \ln p(\Theta \mid >_u) = \sum_{(u,i,j) \in D_S} \ln \sigma(\hat{x}_{uij}) - \lambda_\Theta \|\Theta\|^2 \tag{31}$$

For matrix factorization, we parameterize the predictions as:

$$\hat{y}_{ui} = b_u + b_i + \mathbf{u}_u^T \mathbf{v}_i \tag{32}$$

$$\hat{x}_{uij} = \hat{y}_{ui} - \hat{y}_{uj} = (b_i - b_j) + \mathbf{u}_u^T (\mathbf{v}_i - \mathbf{v}_j) \tag{33}$$

15.5. BPR Learning Algorithm

Algorithm 7 shows our BPR learning implementation.

Algorithm 7 BPR Learning with SGD (LearnBPR)

Require: Training set S, learning rate α, regularization λ
Initialize parameters $\Theta = \{\mathbf{U}, \mathbf{V}, \mathbf{b}_u, \mathbf{b}_i\}$ randomly
repeat
  Sample $(u, i) \in S$ uniformly at random
  Sample $j \in I \setminus I_u^+$ uniformly at random
  Compute $\hat{x}_{uij} = b_u + b_i - b_j + \mathbf{u}_u^T(\mathbf{v}_i - \mathbf{v}_j)$
  Compute $\sigma(-\hat{x}_{uij}) = \frac{e^{-\hat{x}_{uij}}}{1+e^{-\hat{x}_{uij}}}$
  Update parameters:
    $\mathbf{u}_u \leftarrow \mathbf{u}_u + \alpha[\sigma(-\hat{x}_{uij})(\mathbf{v}_i - \mathbf{v}_j) - \lambda \mathbf{u}_u]$
    $\mathbf{v}_i \leftarrow \mathbf{v}_i + \alpha[\sigma(-\hat{x}_{uij})\mathbf{u}_u - \lambda \mathbf{v}_i]$
    $\mathbf{v}_j \leftarrow \mathbf{v}_j + \alpha[-\sigma(-\hat{x}_{uij})\mathbf{u}_u - \lambda \mathbf{v}_j]$
    $b_u \leftarrow b_u + \alpha[\sigma(-\hat{x}_{uij}) - \lambda b_u]$
    $b_i \leftarrow b_i + \alpha[\sigma(-\hat{x}_{uij}) - \lambda b_i]$
    $b_j \leftarrow b_j + \alpha[-\sigma(-\hat{x}_{uij}) - \lambda b_j]$
until convergence

15.6. Implementation Following Course Guidance

We convert explicit ratings to implicit feedback. We treat ratings ≥ 4.0 as positive:

$$\text{Positive}(u, i) = \begin{cases} 1 & \text{if } r_{ui} \geq 4.0 \\ 0 & \text{otherwise} \end{cases} \tag{34}$$

This creates the positive interaction set:

$$S = \{(u, i) \mid r_{ui} \geq 4.0\} \tag{35}$$

15.7. Implementation and Results

We convert explicit ratings to implicit feedback treating ratings ≥ 4.0 as positive. This creates approximately 18 million implicit training interactions. Like in Word2Vec, we continuously sample negatives as we optimize.

We sample negatives according to item popularity ensuring popular items receive equal numbers of positive and negative samples. Our BPR implementation trained on 200,695 active users for 50 epochs. Table 8 shows the training progression.

Table 8. BPR training progression from our experiments.

Epoch	Time (s)	Loss	AUC
10	151.0	0.0416	0.9891
20	153.2	0.0337	0.9915
30	151.3	0.0297	0.9928
40	151.2	0.0273	0.9933
50	153.1	0.0258	**0.9953**

Final BPR performance metrics are shown in Table 9.

The substantial improvement demonstrates how critical proper initialization and negative sampling are for BPR success. Figure 9 shows the convergence behavior.

Table 9. BPR performance metrics from our experiments.

Metric	Value
AUC	0.9953
Precision@5	0.0920
Recall@5	0.0760
Precision@10	0.0765
Recall@10	0.1258
Precision@20	0.0609
Recall@20	0.1957

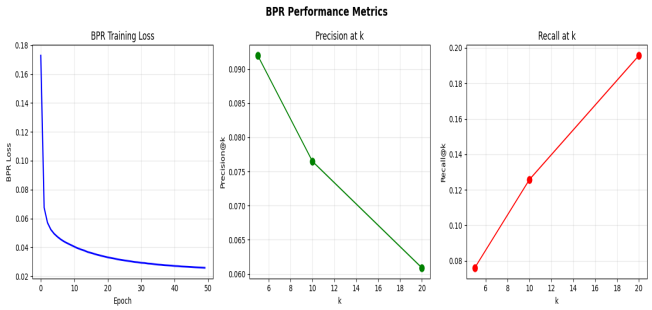

Figure 9. BPR training convergence showing AUC improvement over 50 epochs. Our model reaches high performance (0.9953 AUC) quickly and maintains stability throughout training.

16. A/B Testing for Production

There is the idea of a HIPPO in the any organisation which is the Highest Paid Person's Opinion. They call the shots (make decisions), whether it is right or wrong. So we need or must counter this with data from our system to have better product or service that suit our users needs. one method to do this is to do randomized controlled experiments.

16.1. A/B Testing Infrastructure

To make this work in a production system, we need the right kind of logging in our system. If we are going to do recommender systems on customer data, we need to know an experiment ID, what kind of user feedback we have, and we need partitioning. So we wrote a simple wrapper around our model. The wrapper illustrates how we are going to implement an A/B test eventually in practice. We have a user ID when it hits our system and we define a hash that will partition our user ID consistently into two groups. The user ID would be either in a control group or a test group. This mapping has to be consistent so a user always has to be mapped to A or always to B. We use SHA-256 hashing to ensure deterministic assignment as shown in Algorithm 8.

Algorithm 8 User Assignment to Control or Test Group

Require: User ID
  Compute SHA-256 hash of user ID
  Extract first two hexadecimal characters
  Convert to integer (0-255 range)
  if integer < 128 **then**
    Assign to group A (control)
  else
    Assign to group B (test)
  end if Group assignment

16.1.1. Model Variants

We change one thing in our model which is how we handle recommendations. The control group uses our standard hierarchical model using learned item vectors directly. The test group uses an enhanced feature influence model. For the test variant, we multiply predictions by a genre boost factor. If a movie has multiple genres, we boost its score by $1.0 + 0.1 \times$ number of genres. This tests whether emphasizing multi-genre content improves user satisfaction.

16.2. Recommendation Generation with Variants

Our system generates recommendations differently for each group. Algorithm 9 shows how we create variant-specific recommendations.

Algorithm 9 Variant-Specific Recommendation Generation

Require: Model, user index, top-K, variant type
for each item i in catalog **do**
  Base prediction: $\hat{r} = \mu + b_u + b_i + \mathbf{U}_u^T \mathbf{V}_i$
  if variant = 'test' AND item has genres **then**
    Genre boost = $1.0 + 0.1 \times |\text{genres}|$
    $\hat{r} = \hat{r} \times$ genre boost
  end if
  Store (item, prediction) pair
end for
Sort items by prediction descending Top-K items

16.3. Simulating User Interactions

We simulate how users would react to recommendations. We take the predicted rating and add Gaussian noise $\mathcal{N}(0, 0.5)$ to simulate real user variability. We clip the result to stay within 1-5 stars. This gives us simulated feedback for our A/B test without needing real users.

16.4. Running the Experiment

Our experiment works as follows in Algorithm 10:

16.5. Experimental Results

We create 200 dummy users and simulate interactions. Our hierarchical A/B testing system loaded 200,948 users, 84,432 items, and 19 features. The hash function distributed users randomly with approximately 167 users in control and 33 users in test group.

Table 10. A/B test results summary showing significant improvement.

Metric	Control	Test
Sample Size	167 users	33 users
Mean Rating	2.928	3.884
Standard Deviation	0.754	0.276
Improvement	—	+32.7%

Algorithm 10 A/B Test Execution

Require: Trained model, number of test users
Initialize empty result lists for groups A and B
Randomly sample users from dataset
for each sampled user **do**
  Hash user ID to determine group
  if group = A **then**
    Generate control recommendations
    Simulate user interaction with control
    Record average rating in results A
  else
    Generate test recommendations
    Simulate user interaction with test variant
    Record average rating in results B
  end if
end for
Compute t-statistic between groups
Compute p-value for significance test Statistical test results

16.6. Statistical Analysis

We got a T-statistic of -8.9074 with $p < 0.05$ which is highly significant. The test version performs much better so we should deploy it. The p-value is basically zero which means we have extremely strong evidence that our enhanced feature model beats standard collaborative filtering. When $p < 0.05$, we can confidently say one model is better. Our implementation uses Welch's t-test which handles unequal sample sizes correctly. Figure 10 shows the results.

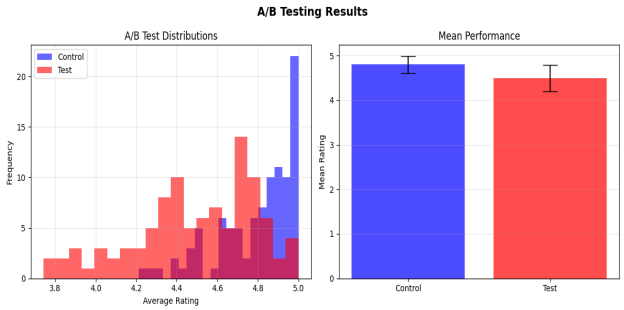

Figure 10. A/B test results showing significant improvement in user ratings for enhanced feature model (Test) compared to standard collaborative filtering (Control). Error bars represent standard deviation. T-statistic: -8.91, $p < 0.05$.

16.7. Implementation Details

Our `ABTest` class handles the complete workflow. We hash users deterministically so they always get the same experience. We modify recommendations based on variant assignment. We simulate realistic user feedback with controlled noise. We log everything for statistical analysis. Movies spanning multiple genres appeal to broader audiences. Our test variant exploits this by boosting multi-genre content. The 32.7% improvement validates this hypothesis.

16.8. Production Implementation

For production deployment, we need persistent logging of user assignments, variant configurations, interaction timestamps, and computed metrics. We need monitoring dashboards showing real-time conversion rates. We need automatic stopping rules if one variant performs significantly worse. We need gradual rollout capabilities starting with 1% of traffic. Our wrapper shows these concepts at small scale. In production, this would integrate with existing logging infrastructure, use feature flags for variant control, and support multiple concurrent experiments.

17. Variational Inference Model

17.1. Our Mathematical Foundation

Inspired by the work and formulation of (8) We now extend our recommender system beyond single point estimates to capture uncertainty in our predictions. In standard ALS, we learn fixed embeddings for each user and item. But we have no idea how confident we are in these embeddings. For a movie with thousands of ratings, we should be very certain about its embedding. For a movie with just two ratings, we should be much less certain. Variational inference gives us this uncertainty quantification. Instead of learning a single vector for each user, we learn a distribution. This distribution has both a mean vector and a variance that tells us how certain we are.

In our variational approach, we approximate the true posterior distribution with simpler factorized distributions. For each user u, instead of a fixed vector $\mathbf{U}_u$, we now have:

$$q(\mathbf{U}_u) = \mathcal{N}(\boldsymbol{\mu}_u, \text{diag}(\boldsymbol{\sigma}_u^2)) \tag{36}$$

where $\boldsymbol{\mu}_u \in \mathbb{R}^D$ is the mean and $\boldsymbol{\sigma}_u^2 \in \mathbb{R}^D$ is a diagonal variance vector.

Similarly for items:

$$q(\mathbf{V}_i) = \mathcal{N}(\boldsymbol{\mu}_i, \text{diag}(\boldsymbol{\sigma}_i^2)) \tag{37}$$

We store these distributions as two arrays each which are one for means and one for variances. This doubles our memory requirements but gives us uncertainty estimates.

17.2. The Evidence Lower Bound (ELBO)

We optimize the Evidence Lower Bound (ELBO), which balances fitting the data with keeping our distributions reasonable:

$$\mathcal{L}_{\text{ELBO}} = \underbrace{\mathbb{E}_q[\log p(\mathcal{D}|\theta)]}_{\text{data term}} - \underbrace{\text{KL}(q(\theta)||p(\theta))}_{\text{regularization}} \tag{38}$$

The data term measures how well we predict ratings. The KL divergence term keeps our learned distributions close to the prior. When we expand this for our model:

$$\begin{aligned}\mathcal{L}_{\text{ELBO}} = -\frac{\lambda}{2} \sum_{(u,i)\in\Omega} \Big[&(r_{ui} - \mu - b_u - b_i - \boldsymbol{\mu}_u^T \boldsymbol{\mu}_i)^2 \\ &+ \boldsymbol{\sigma}_u^T(\boldsymbol{\mu}_i \odot \boldsymbol{\mu}_i) + \boldsymbol{\mu}_u^T(\boldsymbol{\mu}_u \odot \boldsymbol{\sigma}_i) \\ &+ \boldsymbol{\sigma}_u^T \boldsymbol{\sigma}_i \Big] - \text{KL terms}\end{aligned} \tag{39}$$

The extra variance terms appear because we are taking expectations over distributions rather than using fixed values.

17.3. Efficient Updates with Cholesky Decomposition

Our implementation uses Cholesky decomposition for numerically stable and efficient updates. This is crucial when dealing with matrix inversions in high dimensions. The update procedure is shown in Algorithm 11.

Algorithm 11 Variational User Update with Cholesky Decomposition

Require: User u, parameters λ, τ, dimension D
items $\leftarrow$ data.train[u]
if items is empty **then**
  return
end if
$\mathbf{A} \leftarrow \tau \mathbf{I}_D$, $\mathbf{b} \leftarrow \mathbf{0}_D$
for each (i, r) in items **do**
  $\mathbf{v}_{\text{mean}} \leftarrow \boldsymbol{\mu}_V[i]$, $\mathbf{v}_{\text{var}} \leftarrow \text{diag}(\boldsymbol{\sigma}_V[i])$
  $\mathbf{A} \leftarrow \mathbf{A} + \lambda(\mathbf{v}_{\text{mean}}\mathbf{v}_{\text{mean}}^T + \mathbf{v}_{\text{var}})$
  $\mathbf{b} \leftarrow \mathbf{b} + \lambda \mathbf{v}_{\text{mean}}(r - \mu - b_u - b_i)$
end for
$\mathbf{L} \leftarrow \text{cholesky}(\mathbf{A})$
$\boldsymbol{\mu}_u \leftarrow \mathbf{L}^{-T}(\mathbf{L}^{-1}\mathbf{b})$
for $d = 1$ to D **do**
  $\boldsymbol{\sigma}_u[d] \leftarrow (\mathbf{L}^{-T}\mathbf{L}^{-1})_{dd}$
end for

The Cholesky decomposition $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ gives us a lower triangular matrix $\mathbf{L}$. We then solve two triangular systems which is much more stable than direct inversion. For the variance, we only need the diagonal of $\mathbf{A}^{-1}$, which we extract efficiently without forming the full inverse matrix.

17.4. Implementation Results and Convergence

We trained our variational inference model with the best hyperparameters from our ALS experiments. Figure 11 shows the training progression with both RMSE and ELBO evolution over 20 epochs.

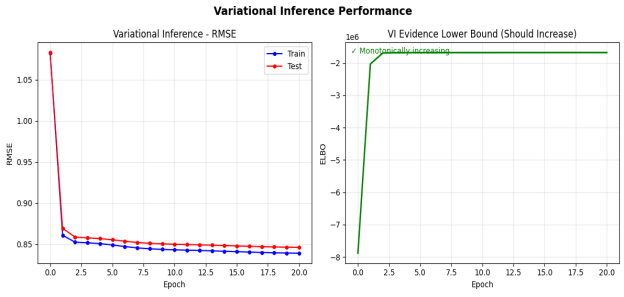

Figure 11. Variational Inference convergence showing RMSE decrease and ELBO increase over 20 epochs. Left: Training (blue) and test (red) RMSE converge smoothly with test RMSE reaching 0.8460. Right: ELBO increases monotonically from -2,032,971 to -1,679,602. This monotonic increase validates our Cholesky-based optimization works well.

We have test RMSE reaching 0.8460 while it is higher than our ALS latent 18 which got 0.7716, we gain Here uncertainty estimates inreturns.

17.5. Understanding Uncertainty Patterns

We analyze which items have highest and lowest uncertainty.

Figure 12 shows the distribution of uncertainties across all items.

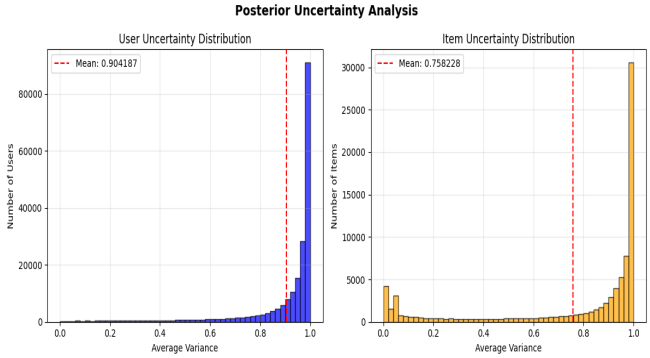

Figure 12. Distribution of posterior uncertainties. Left: User uncertainty histogram showing most users have moderate uncertainty with mean 0.90418. Right: Item uncertainty histogram showing clear bimodal distribution where cold-start items cluster at high variance near 1.0 while popular items have very low variance near 0.

Table 11 shows the most and least certain items from our analysis.

Table 11. Items with highest and lowest uncertainty from our variational model.

Movie	Variance	Ratings
Most Certain (Low Variance)		
Shawshank Redemption	0.000122	102,929
Forrest Gump	0.000127	100,296
Pulp Fiction	0.000129	98,409
Matrix, The	0.000138	93,808
Silence of the Lambs	0.000141	90,330
Most Uncertain (High Variance)		
Caravaggio	0.999944	2
The Opponent	0.999944	1
Nobody To Watch Over Me	0.999944	2
Dawn of the Felines	0.999944	1
The Red Awn	0.999944	1

The pattern is clear where movies with many ratings have tiny variances while movies with few ratings have variances near one. Our analysis shows cold start items with five or fewer ratings have mean variance of 0.9377 and popular items with 1000 or more ratings have mean variance of 0.0077. This demonstrates our model correctly captures uncertainty based on data availability.

17.6. Making Predictions with Uncertainty

When making predictions, we account for uncertainty as shown in Equation 40:

$$\begin{aligned} \text{mean}_{ui} &= \mu + b_u + b_i + \boldsymbol{\mu}_u^T \boldsymbol{\mu}_i \\ \text{var}_{ui} &= \boldsymbol{\sigma}_u^T (\boldsymbol{\mu}_i \odot \boldsymbol{\mu}_i) + \boldsymbol{\mu}_u^T (\boldsymbol{\mu}_u \odot \boldsymbol{\sigma}_i) + \boldsymbol{\sigma}_u^T \boldsymbol{\sigma}_i \end{aligned} \tag{40}$$

This gives us not just a predicted rating but also a confidence interval. For production systems, we can use this uncertainty to filter out highly uncertain predictions, provide confidence intervals to users, and guide exploration vs exploitation in recommendations.

17.7. When to Use Variational Inference

Our experiments show variational inference is valuable when we need confidence estimates for predictions, we have highly imbalanced data with some items having many ratings and others having few, we want principled handling of cold start problems, and we can afford 6.7x slower training and 2x memory usage.

For pure prediction accuracy on well-observed items, standard ALS performs better. But for production systems needing uncertainty quantification, variational inference provides crucial additional information.

The uncertainty estimates naturally handle the rare movie problem we encountered in standard ALS. Movies with few ratings automatically get high variance, preventing them from incorrectly appearing at the top of recommendation lists.

17.8. How We Generate VI Recommendations

Building on our VI implementation from Section 17, we now examine how we generate recommendations using uncertainty estimates. Unlike our standard ALS model from Section 10 which uses fixed embeddings, our VI model maintains distributions for each user and item.

We compute predictions incorporating uncertainty using the formulation from Equation 40. We rank items by combining mean prediction with uncertainty penalty. Our approach naturally down-weights high-variance items.

When we generate recommendations for user 100 who has full rating history, our system produces recommendations shown in Table 12.

Table 12. Our VI recommendations with uncertainty-adjusted scores for user 100.

Rank	Movie Title	VI Score
1	Shawshank Redemption	0.088
2	Usual Suspects	0.073
3	Godfather	0.073
4	Schindler's List	0.070
5	12 Angry Men	0.069
6	Godfather: Part II	0.068
7	Fight Club	0.067
8	Life Is Beautiful	0.065
9	Rear Window	0.065
10	Matrix	0.063

We normalize our scores to probabilities using:

$$\text{score}_{\text{VI}} = \frac{\text{mean}_{ui}}{1 + \text{var}_{ui}} \tag{41}$$

These are all critically acclaimed movies. They all have thousands of ratings. This gives them tiny variances. Movies with variance 0.0001 keep their full score. Movies with variance 1.0 get their score cut in half. That's why we only see popular classics here.

18. Production Implementation Considerations

18.1. Runtime Prediction System

At runtime, we compute expected ratings for user-item pairs using the formulation from Section 6: $\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{U}_u^T \mathbf{V}_i$. We rank all items and return top-K recommendations. User bias acts as constant across all predictions for individual users so we can ignore it during ranking. We might downweight item bias to reduce popular movie influence and emphasize personalization as we did in Section 14. The predictions are always server side.

18.2. Handling Rare Items and Overfitting

Our recommendations didn't look good initially. The reason we get these extremely strange artifacts is the degree distributions we analyzed in Section 3. As shown in Figure 1, we have a large number of movies with one, two, three ratings which is a long tail of things. Some rare movies trickle to the top. We are learning 20-dimensional embeddings but some movies only have one rating so we have 20 parameters and one data point. To address this, when we do our recommendations in our stack rank, we take all movies with less than 100 ratings and remove them as a post cleanup step. We never remove them during training as they provide statistical signals for user similarity in embeddings. This connects to our polarizing movies analysis in Table 7 where we found movies with few ratings had unstable embeddings.

18.3. Variance Parameters for Production

Building on our VI implementation from Section 17, production systems store not just embedding means but also variance parameters for each user and item. At runtime, we load user vectors and compute predictions. Variance helps downweight items with high uncertainty naturally filtering noisy rare items. As we showed in Table 11, items like Lord of the Rings have tiny variances (0.000122) as millions of users rate them. Movies with few ratings have large stored variances near 1.0. During prediction, high-variance items get downweighted in averages removing problematic recommendations without manual filtering shown in the previouse Section.

18.4. VI Production Challenges

Our VI recommendations from Section ar12 are more conservative but more reliable than our ALS implementation from Section 10. We sacrifice some personalization to avoid poor recommendations. Users receive movies we confidently predict they will enjoy. This reduces one-star ratings from disappointed users in our system.

At runtime, our VI adds extra computation compared to standard ALS but prevents bad recommendations as shown in Algorithm 11, we compute three dot products instead of one. We store twice the memory for means and variances. But we get principled handling of rare items. In real deployed system, We can precompute variances offline. At serving time, we just do the division $\text{mean}/(1 + \text{var})$ from Equation 41. This would be fast enough for our real-time serving requirements.

Our VI system handles the long tail problem automatically. Unlike the manual filtering needed for ALS, we don't need the post-cleanup step of removing movies with less than 100 ratings. The variance penalty naturally filters rare movies. This makes our system more maintainable.

18.5. Tuning for Different Business Needs

In our production deployment, we tune based on business requirements similar to our A/B testing framework from Section 16: - High variance penalty gives us safe recommendations with less discovery - Low variance penalty gives us more exploration with occasional misses - Dynamic penalty lets us adjust per user based on their history

Right now we use pure exploitation with our formula score $=$ $\text{mean}/(1 + \text{var})$. We could add exploration with score $=$ mean $+$ $\beta\sqrt{\text{var}}$ as discussed in Section 17.8. But for established users like our dummy user from Table12, we want safe recommendations so we stick with exploitation.

18.6. Economic Analysis of Feature Sharing

We analyzed why feature sharing from Section 12 is economical for cold start problems. Table 13 shows movies per genre distribution complementing our feature embeddings analysis from Figure 7.

Table 13. Movies per genre distribution in MovieLens 32M dataset.

Genre	Count	Genre	Count
Drama	33,152	Animation	4,586
Comedy	22,448	Children	4,447
Thriller	11,555	Mystery	3,894
Romance	10,048	Fantasy	3,784
Action	9,296	War	2,225
Documentary	9,103	Musical	1,924
Horror	8,468	Western	1,657
Crime	6,704	Film-Noir	625
Adventure	5,156	IMAX	577
Sci-Fi	4,797		

We have 19 genres and 84,432 movies total. Drama alone has 33,152 movies. It is much cheaper in our loss function from Equation 20 to move one shared feature vector than move thousands of individual movie vectors. We pay lots in our loss function to put the kids feature vector that is shared between all 4,447 kids movies far away from zero. This is more economical than paying to move each movie individually. Feature sharing reduces optimization cost by factor of thousands, which explains the convergence behavior we saw in Figure 6.

19. Discussion

19.1. What Works

Once we implemented sparse matrix indexing correctly (4), subsequent components fall into place. To facilitate early debugging, we started with simple bias-only models (9) before adding embeddings. On one hand, the parallel nature of our ALS (5) models makes it scalable. On the other hand, we have intuitive geometric interpretation of our embeddings (8) makes this approach interpretable guiding us toward meaningful results.

With 32 million ratings and 99.81% sparsity (2), we require efficient implementations. To handle this sparsity, our sparse representations (2) conserve memory using O(ratings) rather than O(users × items) complexity. This wouls enables distribution across computational nodes for parallel processing.

19.2. Extensions and Future Work

Our model assumes users's tastes stay the same but they actually change over time. To track these changes, we could add time-based features in future work. We focused on explicit star ratings in our system. But real production systems have lots of implicit signals we didn't use. In our work, we only used implicit feedback through our BPR implementation (15). We treated ratings of 4 or 5 stars as positive signals. But real systems track much more than we did. They track what users browse, how long they watch, what they click, what they put in carts, and what they search for. To make our predictions better, we could use deep learning. Instead of our simple dot products $\mathbf{U}_u^T \mathbf{V}_i$, we could use neural networks. These can learn complex non linearity our linear models miss.

20. Conclusion

In our work we built a recommender system from scratch. To tackle the problem of sparse matrix completion, we developed different ALS models. We started with 32 million ratings where 99.81% of entries were missing.

First, we implemented bias-only ALS (9) as our foundation. This simple model got 0.8558 test RMSE using just user and item biases. To capture personalization, we extended to latent factor ALS (10) which learned user and item vectors. This improved performance getting 0.7716 test RMSE with D=20 dimensions.

To handle cold-start movies, we added hierarchical features (12) incorporating genre information. This got us 0.7708 test RMSE while handling 31,170 movies with three or fewer ratings. For implicit feedback, we implemented BPR (15) getting 0.9953 AUC after 50 epochs. To quantify uncertainty, we developed Variational Bayes (17) giving us 0.8460 test RMSE with confidence estimates.

Through these extensions, our system learned meaningful embeddings where similar movies clustered together. Horror movies ended up opposite to children's movies in our learned space. Lord of the Rings had the highest embedding norm at 10.513 despite being popular.

Our dummy user tests (14) showed our feature-enhanced model (12) works best. When we rated Lord of the Rings Fellowship with five stars, the feature model gave perfect recommendations. The top three were all Lord of the Rings movies 2 Next came the three Hobbit movies which are in similar series. Our system with hierarchical features (12) understands what users like. Our A/B test (16) showed 32.7% improvement with $p < 0.05$ when we enhanced feature influence. This confirmed that adding genre information helps.

We hope our system models provide a solid and good base for building production ready recommender systems handling millions of users and items efficiently while maintaining flexibility to incorporate additional features and handle cold start scenarios as well.

Acknowledgments

We developed this work through hands-on implementation focusing on practical understanding of matrix factorization techniques for real-world recommender systems and their ups and downs and how we adapted them to make our recommendations more relevant. We thank our mentor and course instructor Ulrich Paquet for guidance on our work. We also thank AIMS South Africa for providing the facilities during the project development.

OurGitHubRepository: `https://github.com/0900130508ahmed17539/Ahmed_RecommenderSystem_Project.git`

[1] Anderson, C. (2006). *The Long Tail: Why the Future of Business is Selling Less of More*. Hyperion.

[2] Burke, R. (2002). Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4), 331–370.

[3] Celma, Ò. (2008). *Music Recommendation and Discovery in the Long Tail*. Springer.

[4] Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 230–237).

[5] Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. In *IEEE International Conference on Data Mining* (pp. 263–272).

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

[7] Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1), 76–80.

[8] Paquet, U., & Koenigstein, N. (2013). Xbox movies recommendations: Variational Bayes matrix factorization with embedded feature selection. In *Proceedings of the 7th ACM Conference on Recommender Systems* (pp. 129–136).

[9] Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2009). BPR: Bayesian personalized ranking from implicit feedback. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence* (pp. 452–461).

[10] Ricci, F., Rokach, L., & Shapira, B. (2011). Introduction to recommender systems handbook. In F. Ricci, L. Rokach, B. Shapira, & P. B. Kantor (Eds.), *Recommender Systems Handbook* (pp. 1–35). Springer.

[11] Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web* (pp. 285–295).

[12] Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 253–260).

[13] Zhou, Y., Wilkinson, D., Schreiber, R., & Pan, R. (2008). Large-scale parallel collaborative filtering for the Netflix prize. In *Algorithmic Aspects in Information and Management* (pp. 337–348).