Music Recommendation System using Collaborative Filtering

Abstract

This project implements a music recommendation system using collaborative filtering on the Last.fm dataset, comparing a popularity-based baseline with Alternating Least Squares (ALS) matrix factorization. The report documents the development process including data preprocessing, model implementation, hyperparameter tuning, and evaluation. The ALS model demonstrates significant improvements over the baseline, with analysis revealing insights about confidence weighting in implicit feedback systems and the relationship between training objectives and recommendation quality.

University of Technology Sydney
Foo Ming Kuang
26130714

Google Colab Link:
https://colab.research.google.com/drive/10G8qJ0M8izcIwVKYd6vFfQlLQjPMW0G4?usp=sharing

Problem Statement

Music streaming platforms contain millions of tracks, making it impossible for users to discover all relevant content. Simple popularity-based recommendations suffer from cold-start problems for new artists and popularity bias where already-popular items receive disproportionate exposure, creating a rich-get-richer dynamic.

Objective

Implement and compare three recommendation approaches to predict user preferences from sparse interaction data:

1. Baseline (Popularity): Recommend globally popular artists
2. ALS (Alternating Least Squares): Collaborative filtering via matrix factorization

The goal is to understand the theoretical foundations of collaborative filtering and empirically evaluate how matrix factorization improves over simple popularity-based recommendations.

Data Source

The music listening data is sourced from the Last.fm 360K dataset from Zenodo (https://zenodo.org/record/6090214). The dataset contains 17.5 million listening records from 358,868 users across 160,112 artists. Each record includes user ID (anonymized SHA-1 hash), artist ID (MusicBrainz UUID), artist name, and play count. The data represents implicit feedback - play counts indicate preference strength rather than explicit ratings.

```
!curl -L -o lastfm-dataset-360K.tar.gz https://zenodo.org/records/6090214/files/lastfm-dataset-360K.tar.gz

# Extract
!tar -xzf lastfm-dataset-360K.tar.gz

# Remove tar file
!rm lastfm-dataset-360K.tar.gz

data_path = 'lastfm-dataset-360K/usersha1-artmbid-artname-plays.tsv'

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  542M  100  542M    0     0  17.9M      0  0:00:30  0:00:30 --:--:-- 19.4M
```

```
# Load
df = pd.read_csv(data_path, sep='\t', header=None,
                 names=['user_id', 'artist_id', 'artist_name', 'play_count'])

print(f"Total interactions: {len(df):,}")
print(f"Unique users: {df['user_id'].nunique():,}")
print(f"Unique artists: {df['artist_id'].nunique():,}")
print(f"Play count range: {df['play_count'].min()} - {df['play_count'].max()}")

df.head()

Total interactions: 17,535,655
Unique users: 358,868
Unique artists: 160,112
Play count range: 0 - 419157
```

Figure 1: Data loading and exploration code

Data Preprocessing

Data Exploration

Three visualizations examined the data distribution:

- Play count distribution: Highly skewed with long tail
- User activity: Log-normal distribution of artists per user
- Artist popularity: Power law distribution

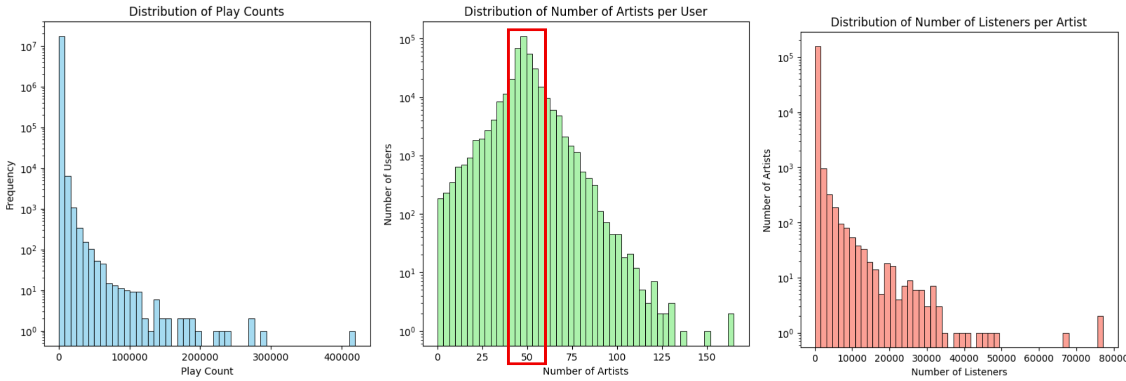


Figure 2: Distribution analysis - (a) Play count distribution,
(b) Number of artists per user, (c) Number of listeners per artist

Filtering Strategy

To address cold-start problems and computational constraints, I applied two filtering criteria:

- Users: Minimum 5 unique artists (ensures sufficient listening history)
- Artists: Minimum 5 listeners (avoids overfitting to niche content)

```
# Filter users with at least 5 unique artists
user_counts = df.groupby('user_id').size()
active_users = user_counts[user_counts >= 5].index
df_filtered = df[df['user_id'].isin(active_users)]

# Filter artists with at least 5 unique listeners
artist_counts = df_filtered.groupby('artist_id').size()
popular_artists = artist_counts[artist_counts >= 5].index
df_filtered = df_filtered[df_filtered['artist_id'].isin(popular_artists)]
```

Figure 3: User and artist filtering code

Data Scaling

For computational efficiency, I scaled down to the top 20,000 most active users. This maintains the most information-rich portion of the dataset while making model training feasible:

```
# Scale down to the top 20,000 users
top_users = df_filtered['user_id'].value_counts().head(20000).index
df_scaled = df_filtered[df_filtered['user_id'].isin(top_users)]


print(f"Final scaled data: {len(df_scaled):,} interactions")
print(f"Users: {df_scaled['user_id'].nunique():,}")
print(f"Artists: {df_scaled['artist_id'].nunique():,}")

Final scaled data: 1,357,087 interactions
Users: 20,000
Artists: 66,959
```

Figure 4: Data scaling to top 20,000 users

Sparse Matrix

The interaction matrix contains 99.90% zeros (1,357,032 non-zero entries out of 1,339,180,000 possible). This extreme sparsity is typical in recommendation systems since most users have only listened to a tiny fraction of available artists. Using SciPy's CSR (Compressed Sparse Row) format stores only non-zero values with their positions, reducing memory

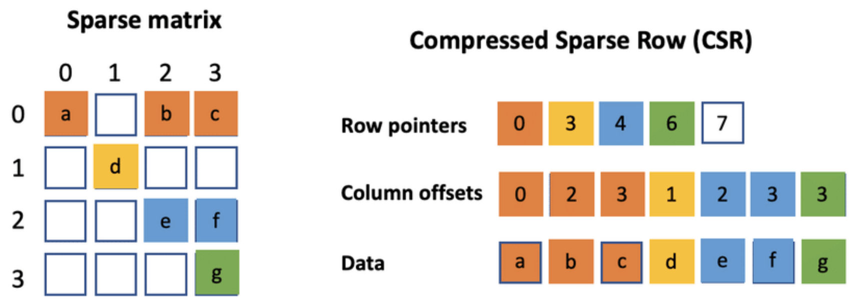


Figure 5: Sparse matrix representation showing CSR format

```
# Create sparse matrix
rows = df_scaled['user_idx'].values
cols = df_scaled['artist_idx'].values
data = df_scaled['play_count'].values
sparse_matrix = csr_matrix((data, (rows, cols)),
                           shape=(len(user_mapping), len(artist_mapping)))
```

Figure 6: Sparse matrix creation code

Train/Test Split

Recommendation systems require per-user splitting rather than random splitting. For each user with at least 5 interactions, 20% of their listening history was randomly held out for testing while the remaining 80% was used for training. This simulates predicting which artists a user will listen to next based on their history.

Model Implementation

Baseline Model: Popularity-Based Recommender System

I implemented a popularity-based baseline to establish a performance lower bound before evaluating more sophisticated collaborative filtering approaches.

```python
# Calculate artist popularity from training data
artist_popularity = np.array(train_matrix.sum(axis=0)).flatten()

# Get top artists for baseline recommendations
n_recommendations = 10
top_artists_baseline = np.argsort(artist_popularity)[::-1][:n_recommendations]
```

Figure 7: Baseline popularity calculation code

Definition:

The simplest possible recommendation strategy: rank all artist by total play counts, then recommend the top-K (in our case we use k=10, which is a standard practice in recommender system research (Cremonesi et al., 2010; He et al., 2017)) to every user regardless individual taste.

Mathematical Formulation:

$$\boldsymbol{popularity(j) = \sum_{i=1}^{n} r_{ij}}$$

where:

- n = users in the training set
- r_{ij}= play count of user i for artist j (implicit feedback)
- The sum aggregates all interactions across the user base

Hypothesis Space:

Contains one member: $h(u) = [a_1, a_2, \dots, a_{10}]$ for all users, where artists are sorted by popularity. No learnable parameters exist so the baseline simply memorizes global statistics.

$$\boldsymbol{H_{baseline} = \{h | h(u) = [a_1, a_2, \dots, a_{10}]\ for\ all\ users\ u\}}$$

Characteristics:

- |H| = 1 (single hypothesis)
- No learnable parameters
- Zero generalization challenge (memorization of training data)

Results:

Metric	Value
Precision	0.0282
Recall	0.0217
NDGC	0.0322
Coverage	0.0001

Zero personalization: The system recommends identical artists to everyone - a death metal fan and a classical music listener both get the same 10 popular artists. This explains the low precision (2.82%) and recall (2.17%).

Poor ranking quality: NDCG of 3.22% shows that even when the baseline does recommend something relevant, it's usually buried lower in the list instead of at the top where users would see it first.

Severe popularity bias: Coverage of 0.01% means only 10 artists out of 66,959 ever appear in recommendations. The same mainstream hits dominate for every user, while 66,949 artists (99.99% of the catalogue) remain completely invisible. This creates a "rich-get-richer" cycle where already-popular artists get even more exposure.

Missing user interests: Recall of 2.17% means we fail to surface 98% of what users would actually enjoy. The system only captures a tiny fraction of each user's real music taste.

Limitations:

Zero personalization, severe popularity bias, and inability to discover niche preferences. These weaknesses directly motivate collaborative filtering approaches like ALS, which can learn latent user-artist affinities and recommend from the long tail.

ALS Model: Collaborative Filtering via Matrix Factorization

After establishing the baseline performance, I implemented an ALS (Alternating Least Squares) model to address the fundamental limitations of popularity-based recommendations. While the baseline achieved only 3% precision and recommended the same 10 artists to every user, collaborative filtering through matrix factorization can learn personalized preferences by discovering latent patterns in user-artist interactions.

The extreme sparsity makes standard matrix operations computationally prohibitive. ALS is specifically designed to handle sparse matrices efficiently by only computing gradients for observed entries, making it feasible to train on this data.

The core idea behind ALS is to decompose the sparse user-item interaction matrix R (20,000 users × 66,959 artists in my scaled dataset) into two lower-dimensional matrices:

- User factors matrix U (20,000 users × k latent factors)
- Artist factors matrix A (66,959 artists × k latent factors)

The approximation is: $\boldsymbol{R} \approx \boldsymbol{U} \times \boldsymbol{A}^{\boldsymbol{T}}$

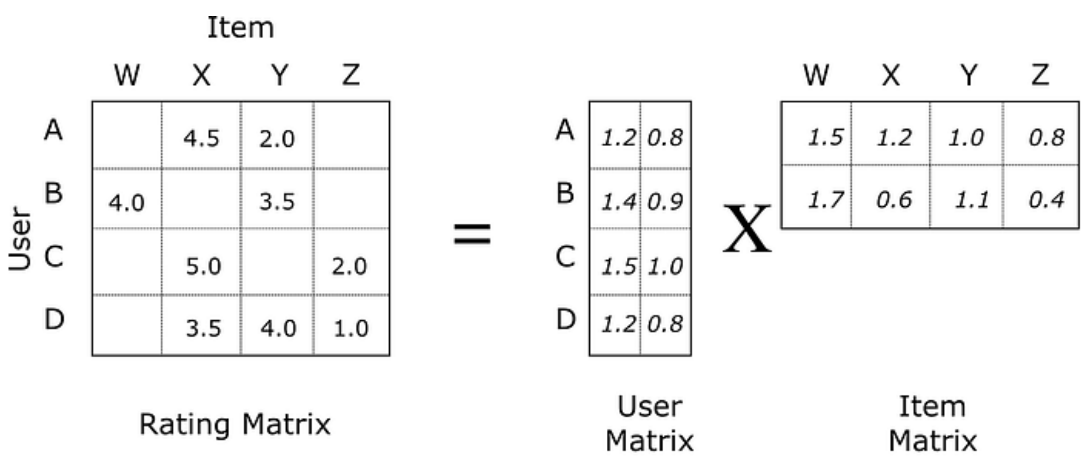


Figure 8: Matrix factorization showing Rating Matrix = User Matrix × Item Matrix

<u>Hypothesis Space</u>

The ALS hypothesis space consists of all possible low-rank matrix factorizations:

$$\boldsymbol{H_{ALS}} = \{\boldsymbol{h} | \boldsymbol{h}(\boldsymbol{i},\boldsymbol{j}) = \boldsymbol{u_i}^{\boldsymbol{T}} \boldsymbol{v_j}, \boldsymbol{where}\ \boldsymbol{U} \in \mathbb{R}^{\{n\times k\}}, \boldsymbol{V} \in \mathbb{R}^{\{m\times k\}}\}$$

where:

- i = user index
- j = artist index
- u_i = user i's k-dimensional factor vector
- v_j = artist j's k-dimensional factor vector

Visualizing Matrix Factorization (Collaborative Filtering Process)

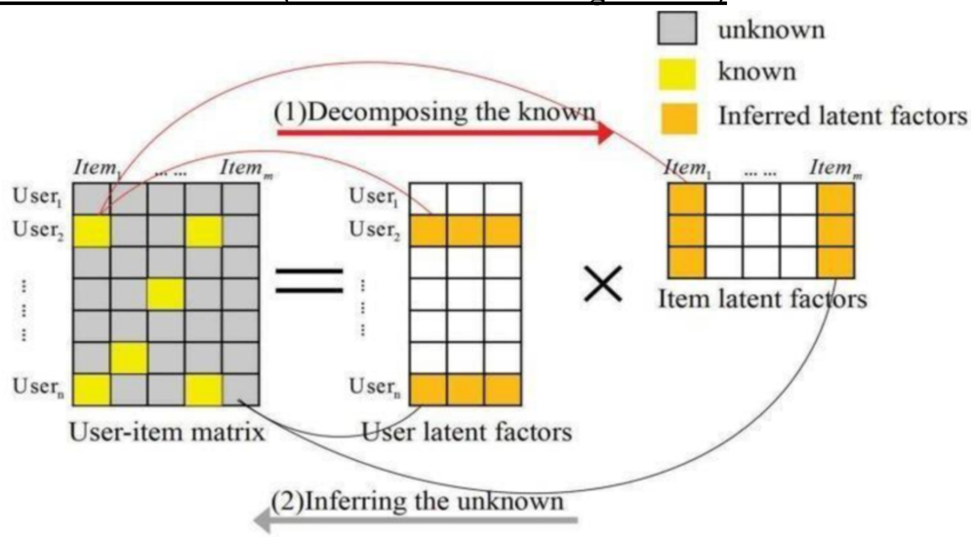


Figure 9 : Collaborative filtering through matrix factorization.
Phase 1 learns latent factors from known interactions (yellow cells). Phase 2 uses these factors to predict unknown interactions (grey cells).

Phase 1 - Decomposing the Known:
ALS decomposes the sparse interaction matrix into two dense matrices:

- User latent factors: Each user becomes a k-dimensional vector
- Item latent factors: Each artist becomes a k-dimensional vector

Where k is a hyperparameter (we explore k ∈ {30, 50, 70, 100} in our tuning).

Phase 2 - Inferring the Unknown:
To predict any unknown interaction $\hat{r}_{ij} = {u_i}^T \times v_j$
This dot product measures alignment in latent space. High alignment = recommended.

How the Alternating Least Square Algorithm Works

ALS solves matrix factorization through iterative optimization. The challenge: optimizing both user factors U and artist factors V simultaneously is non-convex (no guaranteed optimal solution). ALS circumvents this by alternating.

Algorithm Pseudocode:

```
Initialize U and V with random values
For iteration = 1 to 15:

  Step 1: Fix V, optimize U
    For each user i:
      Solve: minimize c_ij × (p_ij − u_i^T v_j)^2 + λ||u_i||^2
      Update u_i using closed-form least squares solution
```

```
Step 2: Fix U, optimize V
  For each artist j:
    Solve: minimize c_ij × (p_ij − u_i^T v_j)² + λ‖v_j‖²
    Update v_j using closed-form least squares solution

Compute total loss L
If loss improvement < threshold: stop
```

When we fix V (all artist vectors), optimizing for user i becomes a standard weighted least squares problem:

- We know artist vectors v_j for all artists user i listened to
- We solve for u_i that best predicts user i's play counts
- This has a closed-form solution (matrix inversion)

The same applies when fixing U and optimizing V.

<u>Loss Function and Optimization</u>

Since the Last.fm dataset contains implicit feedback (play counts rather than explicit ratings), I used a confidence-weighted loss function that the model minimizes:

$$\boldsymbol{L = \sum(i,j) \in all[c_{ij} \times (p_{ij} - u_i{}^T v_j)^2 + \lambda(\|u_i\|^2 + \|v_j\|^2)]}$$

where:

Preference (p_{ij}):

- $p_{ij} = 1$ if user i listened to artist j ($r_{ij} > 0$)
- $p_{ij} = 0$ if no interaction observed ($r_{ij} = 0$)
- Binary indicator separating "interacted" from "didn't interact"

Confidence (c_{ij}):

- $c_{ij} = 1 + \alpha \times r_{ij}$
- Weights each observation by confidence level
- Higher play counts = higher confidence in preference

Prediction ($u_i{}^T v_j$):

- Dot product of user and artist factor vectors
- $u_i{}^T v_j = u_{i1} \times v_{j1} + u_{i2} \times v_{j2} + \cdots + u_{ik} \times v_{jk}$
- Predicted preference score in latent space

Regularization (λ):

- $\|u_i\|^2$ = sum of squared values in user i's factor vector
- $\|v_j\|^2$ = sum of squared values in artist j's factor vector
- Prevents overfitting by penalizing large factor values
- Keeps the model generalizable to new data

Why This Loss Function?

1. Implicit Feedback Challenge:

Unlike explicit ratings (1-5 stars), play counts don't directly indicate preference strength. Zero plays are particularly ambiguous: they could mean dislike OR simply unawareness of that song. The binary preference (p_{ij}) with confidence weighting (c_{ij}) handles this elegantly:

- Positive interactions ($p_{ij} = 1$) are weighted by how many times the user played the artist
- Negative interactions ($p_{ij} = 0$) get minimal weight ($c_{ij} = 1$), reflecting uncertainty

2. Squared Error:

The squared term $(p_{ij} - {u_i}^T v_j)^2$ penalizes larger prediction errors more heavily, which is appropriate given the wide range of play counts in the dataset (1 to 419,157).

3. Regularization:

The $\lambda(\|u_i\|^2 + \|v_j\|^2$ terms prevent the model from using arbitrarily large values to fit training data perfectly, which would lead to poor generalization on new data.

Model Architecture and Hyperparameters

```
als_model = AlternatingLeastSquares(
    factors=int(best['factors']),
    regularization=best['regularization'],
    iterations=int(best['iterations']),
    alpha=best['alpha'],
    random_state=42
)
als_model.fit(train_matrix.tocsr(), show_progress=True)
print("Training complete.")
```

Figure 10: Model architecture and hyperparameter configuration code

- Factors (k): Dimensionality of the latent space. Higher values allow the model to capture more complex patterns but increase computational cost and overfitting risk.
- Regularization (λ): Strength of the L2 penalty term that prevents overfitting by discouraging large factor values. Higher λ produces simpler models.
- Iterations: Number of alternating updates between user and artist factors. More iterations improve convergence but with diminishing returns.
- Alpha (α): Confidence scaling factor in $c_{ij} = 1 + \alpha \times r_{ij}$. Controls how heavily high play counts are weighted relative to low play counts. Higher α emphasizes frequent listeners more.

Hyperparameter Tuning Process

I performed random hyperparameter search with 12 trials to find optimal parameter values. Each trial:

1. Sampled random parameter values from the search space
2. Trained the ALS model on the training set
3. Evaluated Precision@10 on 1,000 sampled users from the test set

Random search was chosen over grid search for computational efficiency as it explores the parameter space effectively while requiring fewer evaluations.

```
# Parameter search space
param_space = {
    'factors': [30, 50, 70, 100],
    'regularization': [0.001, 0.01, 0.05, 0.1],
    'iterations': [10, 15, 20],
    'alpha': [0.5, 1.0, 2.0, 5.0]
}

# Random search configuration
n_trials = 12
results = []

print(f"Random Hyperparameter Search: {n_trials} trials")
print("-" * 60)

for trial in tqdm(range(n_trials), desc="Searching hyperparameters", ncols=80):
    # Sample random parameters
    params = {
        'factors': np.random.choice(param_space['factors']),
        'regularization': np.random.choice(param_space['regularization']),
        'iterations': np.random.choice(param_space['iterations']),
        'alpha': np.random.choice(param_space['alpha']),
        'random_state': 42
    }
```

Figure 11: Random hyperparameter search configuration

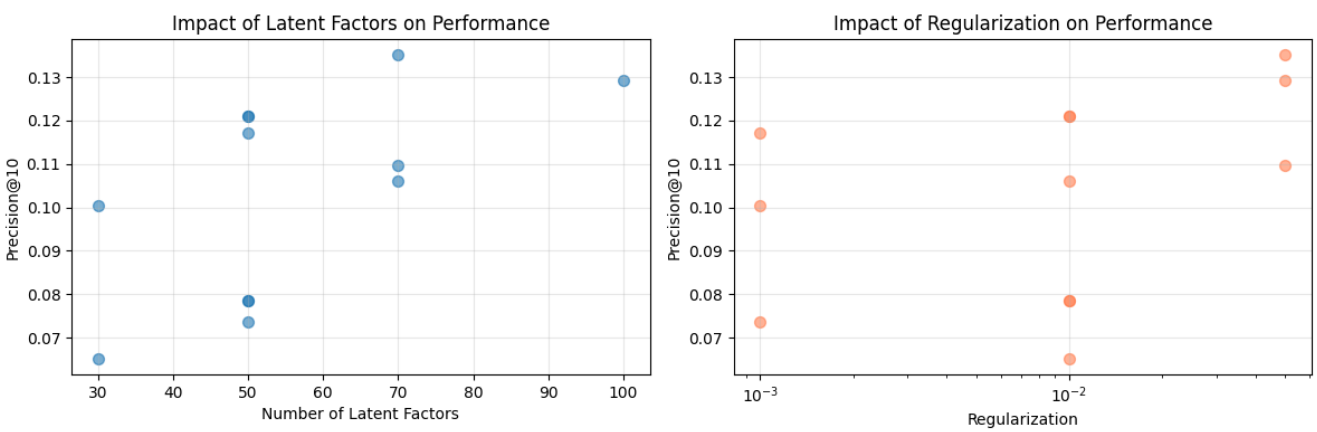


Figure 12: Hyperparameter sensitivity analysis - (a) Impact of latent factors, (b) Impact of regularization

Left plot - Latent Factors (k):
Higher dimensionality (k=70, 100) generally performs better, allowing the model to capture more complex taste patterns. However, k=30 shows competitive performance in some configurations, suggesting the interaction with other hyperparameters matters more than factors alone.

Right plot - Regularization (λ): Strong regularization (λ=0.05, 0.1) performs best, preventing overfitting on sparse data. Very weak regularization (λ=0.001) shows poor performance, confirming that the model memorizes noise without adequate penalty. The scatter pattern shows no single hyperparameter dominates—optimal performance requires balanced tuning across all parameters.

<u>Convergence Analysis</u>

To understand how many iterations ALS requires to reach optimal performance, we trained models with varying iteration counts while holding other parameters constant (factors=100, regularization=0.1, alpha=0.5).

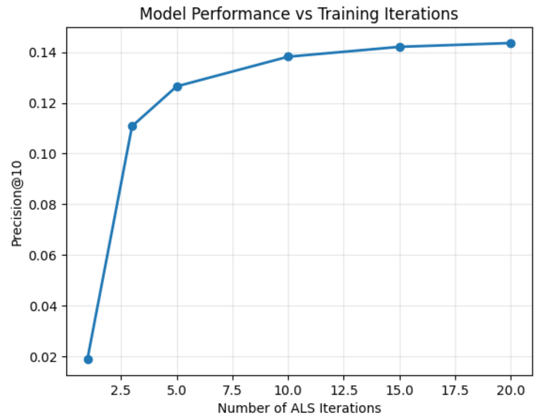


Figure 13: ALS convergence behaviour over training iterations

Observations:
- Iteration 1 - 5: Steep improvement
- Iteration 5 -10: Moderate gains
- Iteration 10 - 20: Minimal improvement

This demonstrates the alternating optimization finding progressively better solutions until convergence. The diminishing returns after iteration 10 validate our choice of 15 iterations for the final model which is enough to ensure convergence without unnecessary computation.

<u>Recommendation Generation</u>

After training, the model generates personalized recommendations by computing predicted scores for all artists a user hasn't listened to yet.

The prediction process:

1. Retrieve learned factors: Extract the user's k-dimensional factor vector u_i and all artist factor vectors v_j.
2. Compute scores: Calculate predicted preference for each artist

$$\boldsymbol{\hat{y}_{ij} = u_i{}^T \times v_j = \sum (f = 1\ to\ k)\, u_{if} \times v_{jf}}$$

where:
- i = user index
- j = artist index
- f = latent factor index

This dot product measures alignment between the user's preferences and the artist's characteristics in the latent space. Higher scores indicate stronger predicted preference.

3. Filter listened artists: Remove artists already in the user's listening history, since recommendations should prioritize discovery
4. Rank and select: Return the top-K artists with highest prediction scores

For example, if a user's factor vector aligns strongly with the factor vectors of indie rock artists, those artists will receive high prediction scores even if the user hasn't listened to them yet. This collaborative filtering effect emerges because the model learns that users with similar listening patterns tend to enjoy similar artists.

```python
def get_als_recommendations(user_idx, k=10):
    """
    Get personalized ALS recommendations for a user.

    Args:
        user_idx: user index (0 to n_users-1)
        k: number of recommendations

    Returns:
        artist_indices: array of recommended artist IDs
        scores: prediction scores
    """
    try:
        # Get user's listening history (for filtering)
        user_items = train_matrix[user_idx].tocsr()

        recs, scores = als_model.recommend(
            userid=user_idx,
            user_items=user_items,
            N=k,
            filter_already_liked_items=True
        )
        return recs, scores

    except Exception as e:
        print(f"Error getting recommendations for user {user_idx}: {e}")
        # Fallback to baseline
        return np.array([]), np.array([])
```

Figure 14: Recommendation generation implementation code

I implemented this using the `implicit` library's `recommend()` method with `filter_already_liked_items=True` to ensure novel recommendations that it doesn't recommend artist which the user already listened to as recommendations should help system discover new artists.

<u>Results and Comparison with Baseline Model</u>

The improvement comes from personalization. The baseline gives everyone the same 10 popular artists, but ALS learns individual preferences by finding users with similar taste. For example, if several users who listen to artist A also enjoy artist B, the model will recommend artist B to similar listeners—even if he/she isn't globally popular.

```python
# Evaluate baseline
baseline_results = evaluate_recommendations(
    "Baseline",
    get_baseline_recommendations,
    test_interactions,
    k=10
)

# Evaluate ALS
als_results = evaluate_recommendations(
    "ALS",
    get_als_recommendations,
    test_interactions,
    k=10
)

# Compare results
print(f"\n{'Metric':<15} {'Baseline':<12} {'ALS':<12} {'Improvement'}")
print("-"*60)
for metric in ['precision', 'recall', 'ndcg', 'coverage']:
    base = baseline_results[metric]
    als = als_results[metric]
    improvement = ((als - base) / base * 100) if base > 0 else 0
    print(f"{metric.capitalize():<15} {base:<12.4f} {als:<12.4f} {improvement:>+7.1f}%")
```

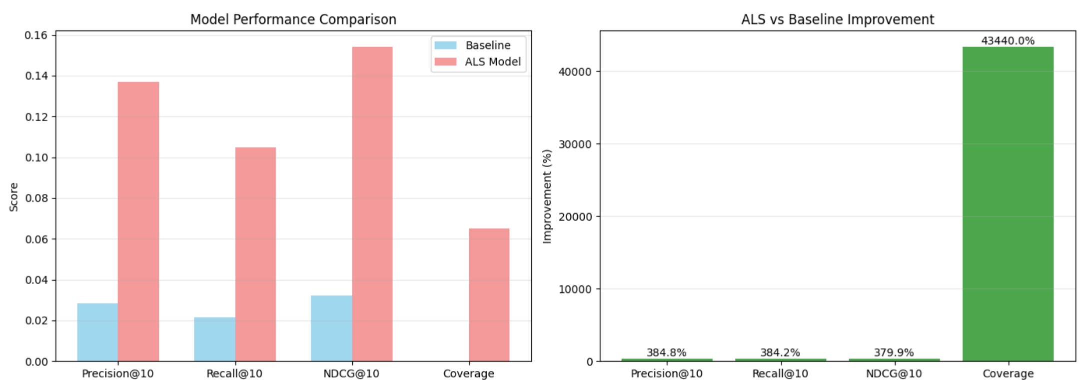
```
Evaluating Baseline...
  Evaluating on 20000 users
  Precision@10: 0.0282
  Recall@10: 0.0217
  NDCG@10: 0.0322
  Coverage: 0.0001

Evaluating ALS...
  Evaluating on 20000 users
  Precision@10: 0.1368
  Recall@10: 0.1049
  NDCG@10: 0.1543
  Coverage: 0.0650

Metric          Baseline     ALS          Improvement
------------------------------------------------------------
Precision       0.0282       0.1368        +384.8%
Recall          0.0217       0.1049        +384.2%
Ndcg            0.0322       0.1543        +379.9%
Coverage        0.0001       0.0650       +43440.0%
```

Figure 15: Model evaluation code

The coverage jump from 10 to 4,352 artists (0.0650(coverage) × 66,959 = 4,352 artists) shows ALS can surface niche content instead of just recycling the same mainstream hits. This breaks the "rich-get-richer" cycle where popular artists dominate recommendations simply because they're already popular.



Metric	Baseline	ALS	Improvement
Precision@10	0.0282	0.1368	+384.8%
Recall@10	0.0217	0.1049	+384.2%
NDGC@10	0.0322	0.1543	+379.9%
Coverage	0.0001	0.0650	+43,440%

Figure 16: Performance comparison - (a) Bar chart, (b) Improvement percentages, (c) Results table

ALS achieves 13.7% precision - nearly 4× better than baseline. While 86% of recommendations are still not in the test set, this represents significant progress. More importantly, coverage jumps from 0.01% to 6.5%, meaning 4,352 distinct artists now appear in recommendations instead of just 10. This breaks the "rich-get-richer" cycle.

The improvement comes from personalization. The baseline gives identical recommendations to every user. ALS learns individual taste profiles by finding users with similar listening patterns. If several users who play Radiohead also enjoy Arcade Fire, the model recommends Arcade Fire to similar listeners even if it's not globally popular.

However, precision of 13.7% highlights remaining challenges. The extreme data sparsity (99.9% missing values) limits the model's ability to learn accurate preferences, especially for users with few listening records or niche artists with small listener bases.

<u>Sample Recommendations Analysis</u>

To illustrate the difference between baseline and collaborative filtering, I examined actual recommendations for a sample user.

```python
sample_user = 0

# User's profile
user_artists = train_matrix[sample_user].nonzero()[1]
total_plays = train_matrix[sample_user].sum()

print(f"Sample User {sample_user} Profile:")
print(f"  Artists listened to: {len(user_artists)}")
print(f"  Total plays: {total_plays}")

# Show what they actually listen to
print(f"\n  Top 5 artists in listening history:")
user_plays = [(idx, train_matrix[sample_user, idx]) for idx in user_artists]
user_plays.sort(key=lambda x: x[1], reverse=True)
for i, (artist_idx, plays) in enumerate(user_plays[:5]):
    print(f"    {i+1}. {get_artist_name(artist_idx)} ({int(plays):,} plays)")

# Get recommendations
baseline_recs = get_baseline_recommendations(5)
als_recs, als_scores = get_als_recommendations(sample_user, 5)

print(f"\nBaseline Recommendations (Same for Everyone):")
for i, artist_idx in enumerate(baseline_recs):
    plays = int(artist_popularity[artist_idx])
    print(f"  {i+1}. Artist {artist_idx} ({plays:,} total plays)")

print(f"\nALS Recommendations (Personalized for User {sample_user}):")
for i, (artist_idx, score) in enumerate(zip(als_recs, als_scores)):
    print(f"  {i+1}. {get_artist_name(artist_idx)} (score: {score:.3f})")

# Check against test set
user_test_items = [item['item_idx'] for item in test_interactions if item['user_idx'] == sample_user]
if user_test_items:
    baseline_hits = len(set(baseline_recs) & set(user_test_items))
    als_hits = len(set(als_recs) & set(user_test_items))
    print(f"\nActual Relevance (Test Set Hits):")
    print(f"  Baseline: {baseline_hits}/5")
    print(f"  ALS: {als_hits}/5")
```

Figure 17: Sample user recommendation analysis code

```
Sample User 0 Profile:
  Artists listened to: 52
  Total plays: 12727

  Top 5 artists in listening history:
    1. master's hammer (748 plays)
    2. misfits (726 plays)
    3. rubella ballet (687 plays)
    4. cauda pavonis (609 plays)
    5. creepersin (583 plays)

Baseline Recommendations (Same for Everyone):
  1. Artist 156 (1,015,612 total plays)
  2. Artist 305 (712,181 total plays)
  3. Artist 816 (505,658 total plays)
  4. Artist 344 (503,991 total plays)
  5. Artist 421 (436,294 total plays)

ALS Recommendations (Personalized for User 0):
  1. the doors (score: 1.404)
  2. tom waits (score: 1.244)
  3. inkubus sukkubus (score: 1.227)
  4. christian death (score: 1.195)
  5. horrorpops (score: 1.159)

Actual Relevance (Test Set Hits):
  Baseline: 0/5
  ALS: 2/5
```

Figure 18: Sample User 0 recommendation comparison output

ALS successfully predicted 2 out of 5 artists that User 0 actually listened to in the test set, while baseline's globally popular recommendations matched none of this user's real interests. This demonstrates how collaborative filtering discovers individual taste patterns - the model learned that User 0's listening history (master's hammer, misfits, rubella ballet) shares characteristics with other users who also enjoy gothic/alternative rock artists like the doors and christian death. The baseline, recommending identical popular artists to everyone, completely missed this user's actual music preferences.

<u>Recommendation Quality at Different List Length</u>

To understand how recommendation quality changes with list size, I evaluated both models at K ∈ {1, 5, 10, 15, 20}.

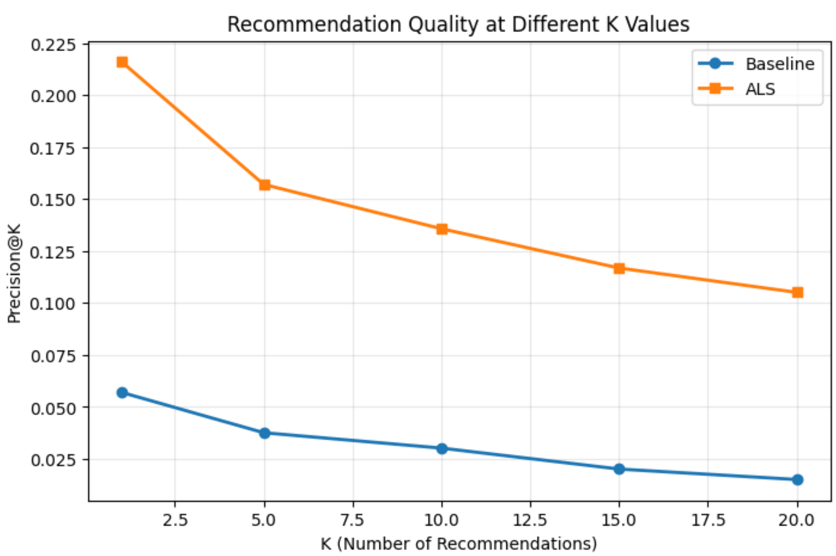


Figure 19: Precision@K for varying recommendation list lengths.

As K increases, precision decreases because lower-ranked recommendations are less confident predictions. For user interfaces, K=5-10 balances precision (relevance per item) with recall (total relevant items found). Beyond K=15, diminishing returns suggest users ignore lower-quality recommendations.

<u>Understanding the Gap Between Training Loss and Real-World Goals</u>

The model minimizes reconstruction error during training - it learns to predict play counts accurately. But that's not really what matters. The problem is play count doesn't equal preference. Someone playing an artist 100 times as background music isn't the same as loving that artist. One perfect discovery is worth more than 100 mediocre recommendations, but the model treats them the same.

The model also can't tell if plays happened in 2009 or last week - it treats all history equally even though tastes change over time. And by optimizing to reconstruct the existing matrix, we end up recommending similar artists to what users already know, which limits discovery.

Our Precision@10 metric only tells us "did the user listen?" not "did they love it?" Production systems need additional metrics like diversity, novelty, and actual engagement (clicks, saves). We optimize what we can measure (play counts) and hope it correlates with what we want (user satisfaction).

<u>Limitations</u>

Despite strong performance, ALS has key limitations:

- Cold Start Problem: New users or artists without interaction history cannot receive or generate recommendations. The model requires listening data to compute factor vectors, so a brand new user with zero plays gets no personalized recommendations.
- Implicit Feedback Ambiguity: Zero plays are ambiguous - they could mean dislike OR simply unawareness. We treat all zeros as weak negatives (confidence = 1), but the model can't distinguish between "user hates this artist" and "user doesn't know this artist exists."
- Popularity Bias: Despite improving coverage from 10 to 4,352 artists, we still only recommend 6.5% of the catalogue. Popular artists create a feedback loop: more plays lead to higher confidence, which leads to more recommendations, which generates even more plays. The remaining 93.5% of artists never get recommended.
- No Temporal Dynamics: User tastes evolve over time, but ALS treats a 2009 play count the same as yesterday's play. The model assumes static preferences with no concept of recency or changing tastes.
- Interaction-Only Learning: ALS learns purely from the play count matrix. Even though the Last.fm dataset includes user demographics (age, gender, country) the model cannot incorporate any of this side information. This limits prediction quality on sparse data where interaction signals are weak.

Conclusion

This project implemented and evaluated collaborative filtering for music recommendations using the Last.fm 360K dataset. The baseline popularity-based model achieved only 2.8% precision with severe coverage limitations (0.01%), recommending identical artists to all users. ALS matrix factorization improved precision to 13.7% and coverage to 6.5% by learning personalized taste profiles from the sparse interaction matrix. While ALS demonstrates the value of collaborative filtering, significant challenges remain including cold-start problems, implicit feedback ambiguity, and the inability to incorporate available demographic and metadata features.

Future Improvement

Factorization Machines could address ALS limitations by incorporating the user demographic data (age, gender, country) available in our dataset. This would improve cold-start predictions for new users and enable the model to learn demographic-based preference patterns. Neural Collaborative Filtering could capture non-linear interaction patterns beyond matrix factorization.

Reference

M. Vigliensoni and I. Fujinaga, "The music listening histories dataset (MLHD)," Zenodo, 2020. [Online]. Available: https://zenodo.org/records/6090214

Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," Computer, vol. 42, no. 8, pp. 30-37, Aug. 2009, doi: 10.1109/MC.2009.263.

P. Cremonesi, Y. Koren, and R. Turrin, "Performance of recommender algorithms on top-n recommendation tasks," in Proc. 4th ACM Conf. Recommender Syst., Barcelona, Spain, 2010, pp. 39-46, doi: 10.1145/1864708.1864721.

X. He et al., "Neural collaborative filtering," in Proc. 26th Int. Conf. World Wide Web, Perth, Australia, 2017, pp. 173-182, doi: 10.1145/3038912.3052569.

C. C. Aggarwal, Recommender Systems: The Textbook. Cham, Switzerland: Springer, 2016.

SciPy developers, "scipy.sparse.csr_matrix," SciPy Documentation. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

S. Watson, "A gentle introduction to recommender systems with implicit feedback," Apr. 2018. [Online]. Available: https://sophwats.github.io/2018-04-05-gentle-als.html

UK Intellectual Property Office, "The impact of algorithmically-driven recommendation systems on music consumption and production," Gov.uk, 2022. [Online]. Available: https://www.gov.uk/government/publications/research-into-the-impact-of-streaming-services-algorithms-on-music-consumption

Appendix

ChatGPT Links:

1. https://chatgpt.com/share/68e1d76b-5da0-8010-ac2f-6a56432ac8aa
2. https://chatgpt.com/share/68e1d77f-95ac-8010-b105-0b752c5cfa2b

Query 1: Alpha Parameter Performance

My Query:
"My hyperparameter search showed alpha=0.5 performed better than alpha=40.0. Why does mid-range alpha outperform very high alpha for my Last.fm data? What does this tell me about my play count distribution?"

ChatGPT Response (key part):
"Very large alpha makes the model treat high play counts as almost certain signals (huge confidence). The ALS objective will fit those heavy counts extremely well — often at the expense of learning from many low-count interactions. This can overfit the few heavy-listen events and hurt ranking/generalization.

The fact that alpha=5 beats alpha=40 suggests that single-plays and small counts are informative, not just the heavy listeners. Your model benefits from learning patterns from those sparse signals."

My Understanding:
The confidence formula $c_{ij} = 1 + \alpha \times r_{ij}$ means that with alpha =4 0, a user who played an artist 100 times gets a confidence of 4001, while 1 play only gets confidence of 41. That's a massive gap. The model will basically only care about fitting the high play counts and ignore everything else.

My dataset after filtering (min 5 artists per user, top 20K users) still has lots of 1-10 play interactions that actually mean something. These aren't just noise they're real preference signals. When alpha is too high, the model treats these as basically irrelevant and only learns from the extreme cases.

That's why alpha = 0.5 worked better because it still gives more weight to repeated plays but doesn't completely ignore the sparse interactions.

How I Used This:
This helped me understand why my empirical results differed from the literature. Papers using alpha=40 are probably working with unfiltered data where low counts really are noise. My preprocessing changed the data characteristics, so optimal hyperparameters changed too. I referenced this in the Limitations section when discussing how confidence weighting creates bias toward popular items.

Query 2: Loss Function vs Evaluation Metric

My Query:

"I need to explain the difference between the loss function and practical objectives. ALS minimizes weighted reconstruction error but I evaluate with Precision@10. Why do we optimize one thing but measure another? Can a model have low training loss but poor Precision@10?"

ChatGPT Response (key part):
"Precision@10 measures ranking quality but is non-differentiable: you can't compute gradients through 'is the item in the top-10 or not?'. Also combinatorial: evaluating requires sorting all items for each user, computationally explosive for large datasets.

Example: imagine a heavy user who has listened to thousands of mainstream pop songs. ALS will learn to reconstruct their history very well (low loss), because the model gives high confidence weight to frequent interactions. But when you rank items, ALS might recommend 'yet another mainstream pop track' that the user has already listened to instead of something new and relevant. Precision@10 drops because recommendations don't contain novel relevant artists."

My Understanding:
During training, ALS just tries to rebuild the user-item matrix accurately. If it predicts play counts close to the actual values, loss goes down. But that doesn't mean the top-10 ranked items will be relevant to the user.

The example makes sense: if a user listened to 500 pop songs, the model will get really good at predicting "yes, this user likes pop" and reconstruction error will be low. But for recommendations, suggesting the 501st pop song might be less useful than suggesting something from a different genre the user occasionally listens to. The model optimized for reconstruction, not for "give me 10 artists I haven't heard but would like."

This is why my training loss converged smoothly to around 0.02 by iteration 15, but Precision@10 varied wildly (0.07 to 0.14) depending on hyperparameters. They're measuring completely different things.