Steam Game Recommendation System - Model Evaluation Report

Complete Analysis and Performance Comparison

Group 5	Name	Student ID
Member 1	Hongkun Tian	320230942381
Member 2	Zhiye Wang	320230942491
Member 3	Ruizhe Zhang	320230942741
Member 4	Ke Meng	320230942231

Contents

Introduction

This report evaluates several recommendation algorithms on the Steam-200k playtime dataset. It compares simple baselines, neighborhood-based Collaborative Filtering, and two matrix factorization

variants (FunkSVD-v2 and Bias-SVD), and presents quantitative results, ablation studies, and practical recommendations.

1. Motivation

Recommender systems are crucial for game e-commerce platforms. Although collaborative filtering is mature, it faces challenges with sparsity and cold-start problems. Matrix Factorization (MF) learns low-rank factor representations to better capture latent user-item interaction patterns. This study systematically evaluates multiple recommendation algorithms on Steam game playtime data and provides in-depth performance analysis of matrix factorization methods.

2. Dataset

- **Source**: Steam-200k game playtime records
- **Original**: 199,999 entries | **After cleaning**: 54,017 entries (73% retained)
- **Scale**: 2,435 users × 1,524 games × 98.54% sparsity
- **Preprocessing**: MinMaxScaler (0.1, 1.0) | 80% train / 20% test

3. Methods

3.1 Baseline Models

- **Global Mean**: Global average prediction
- **User Average**: User historical average prediction

3.2 Collaborative Filtering

User-CF and **Item-CF**: Cosine similarity-weighted prediction based on k-NN (k=20)

User-CF Pseudocode:

```
# Input: user u, item i, rating matrix R, k=20
# Output: predicted rating pred_ui

def user_cf_predict(u, i, R, k=20):
    # Step 1: Compute user similarity (cosine similarity)
    similarities = []
    for u_prime in other_users:
        # Compute cosine similarity of rating vectors for users u and u'
        sim_uu_prime = cosine_similarity(R[u], R[u_prime])
        similarities.append((u_prime, sim_uu_prime))

    # Step 2: Find k most similar users
    top_k_users = sorted(similarities, key=lambda x: x[1], reverse=True)[:k]

    # Step 3: Weighted prediction using similar users' ratings
    numerator = 0.0
    denominator = 0.0
    for u_prime, sim in top_k_users:
        if R[u_prime][i] is not None:  # User has rated item i
            numerator += sim * R[u_prime][i]
            denominator += abs(sim)

    # Step 4: Return weighted average prediction
    if denominator > 0:
        pred_ui = numerator / denominator
    else:
        pred_ui = global_mean  # Return global mean if no valid neighbors

    return pred_ui
```

Item-CF Pseudocode:

```
# Input: user u, item i, rating matrix R, k=20
# Output: predicted rating pred_ui

def item_cf_predict(u, i, R, k=20):
    # Step 1: Compute item similarity (cosine similarity)
    similarities = []
    for i_prime in other_items:
        # Compute cosine similarity of rating vectors for items i and i'
        sim_ii_prime = cosine_similarity(R[:, i], R[:, i_prime])
        similarities.append((i_prime, sim_ii_prime))

    # Step 2: Find k most similar items
    top_k_items = sorted(similarities, key=lambda x: x[1], reverse=True)[:k]

    # Step 3: Weighted prediction using similar items' ratings
    numerator = 0.0
    denominator = 0.0
    for i_prime, sim in top_k_items:
        if R[u][i_prime] is not None:  # User has rated item i'
            numerator += sim * R[u][i_prime]
            denominator += abs(sim)

    # Step 4: Return weighted average prediction
    if denominator > 0:
        pred_ui = numerator / denominator
    else:
        pred_ui = global_mean

    return pred_ui

# Similarity computation
def cosine_similarity(vec1, vec2):
    # Compute cosine of angle between two vectors
    dot_product = sum(a * b for a, b in zip(vec1, vec2))
    norm1 = sqrt(sum(a**2 for a in vec1))
    norm2 = sqrt(sum(b**2 for b in vec2))

    if norm1 * norm2 == 0:
        return 0.0
    return dot_product / (norm1 * norm2)
```

3.3 Matrix Factorization

FunkSVD-v2 (Optimized SGD Matrix Factorization):

```
# Prediction formula
prediction = μ + P[u] · Q[i]

# Loss function
Loss = (1/n) Σ(r - pred)² + λ(||P[u]||² + ||Q[i]||²)

# Hyperparameters
n_factors=20, learning_rate=0.003, reg=0.05, epochs=20
```

FunkSVD-v2 Pseudocode:

```
def train_funksvd_v2(train_data, n_factors=20, lr=0.003, reg=0.05, epochs=20):
    # Initialization
    n_users = num_users
    n_items = num_items
    P = np.random.randn(n_users, n_factors) * 0.001  # User factor matrix
    Q = np.random.randn(n_items, n_factors) * 0.001  # Item factor matrix
    mu = np.mean([r for (u, i, r) in train_data])    # Global mean

    # SGD training
    for epoch in range(epochs):
        for (u, i, r) in train_data:  # Iterate over each training sample
            # Forward pass: compute prediction
            pred = mu + np.dot(P[u], Q[i])
            error = r - pred

            # Backward pass: compute gradients and update
            # Gradient for P[u]
            grad_P = -2 * error * Q[i] + 2 * reg * P[u]
            P[u] -= lr * grad_P

            # Gradient for Q[i]
            grad_Q = -2 * error * P[u] + 2 * reg * Q[i]
            Q[i] -= lr * grad_Q

    return P, Q, mu

def predict_funksvd_v2(u, i, P, Q, mu):
    return mu + np.dot(P[u], Q[i])
```

Bias-SVD (Matrix Factorization with Bias):

```
# Prediction formula
prediction = μ + b_u + b_i + P[u] · Q[i]

# Loss function
Loss = (1/n) Σ(r - pred)² + λ(||b_u||² + ||b_i||² + ||P[u]||² + ||Q[i]||²)

# Hyperparameters
n_factors=20, learning_rate=0.005, reg=0.02, epochs=15
```

Bias-SVD Pseudocode:

```python
def train_bias_svd(train_data, n_factors=20, lr=0.005, reg=0.02, epochs=15):
    # Initialization
    n_users = num_users
    n_items = num_items

    # Factor matrices
    P = np.random.randn(n_users, n_factors) * 0.001
    Q = np.random.randn(n_items, n_factors) * 0.001

    # Bias terms
    b_u = np.zeros(n_users)      # User bias
    b_i = np.zeros(n_items)      # Item bias
    mu = np.mean([r for (u, i, r) in train_data])  # Global mean

    # SGD training
    for epoch in range(epochs):
        for (u, i, r) in train_data:
            # Forward pass: compute prediction
            pred = mu + b_u[u] + b_i[i] + np.dot(P[u], Q[i])
            error = r - pred

            # Backward pass: compute gradients and update
            # Update user bias
            grad_b_u = -2 * error + 2 * reg * b_u[u]
            b_u[u] -= lr * grad_b_u

            # Update item bias
            grad_b_i = -2 * error + 2 * reg * b_i[i]
            b_i[i] -= lr * grad_b_i

            # Update user factor
            grad_P = -2 * error * Q[i] + 2 * reg * P[u]
            P[u] -= lr * grad_P

            # Update item factor
            grad_Q = -2 * error * P[u] + 2 * reg * Q[i]
            Q[i] -= lr * grad_Q

    return P, Q, b_u, b_i, mu

def predict_bias_svd(u, i, P, Q, b_u, b_i, mu):
    return mu + b_u[u] + b_i[i] + np.dot(P[u], Q[i])
```

Key Difference: Bias-SVD explicitly decomposes the global mean and individual biases, allowing factor matrices to focus on learning interaction features.

4. Experimental Setup

Algorithm	n_factors	lr	λ	epochs	Special Parameters
FunkSVD-v2	20	0.003	0.05	20	Global mean constraint
Bias-SVD	20	0.005	0.02	15	-
User-CF	-	-	-	-	k=20
Item-CF	-	-	-	-	k=20

Baselines:

- **Global Mean**: Predicts the global average playtime across all users and games; used as a simple naive baseline for comparison.
- **User Average**: Predicts each user's historical average playtime to capture user-specific offsets and account for per-user activity levels.

Experimental Setup Summary: The table above lists hyperparameters and training configuration for each algorithm; models were trained on the 80% training split and evaluated on the 20% test split after applying MinMax scaling to playtime values.

Evaluation Metrics: MAE, RMSE | **Environment**: Python 3.12, scikit-learn, NumPy

5. Evaluation Results

5.1 Model Performance Analysis (Quantitative Analysis)

Quantitative Results Table

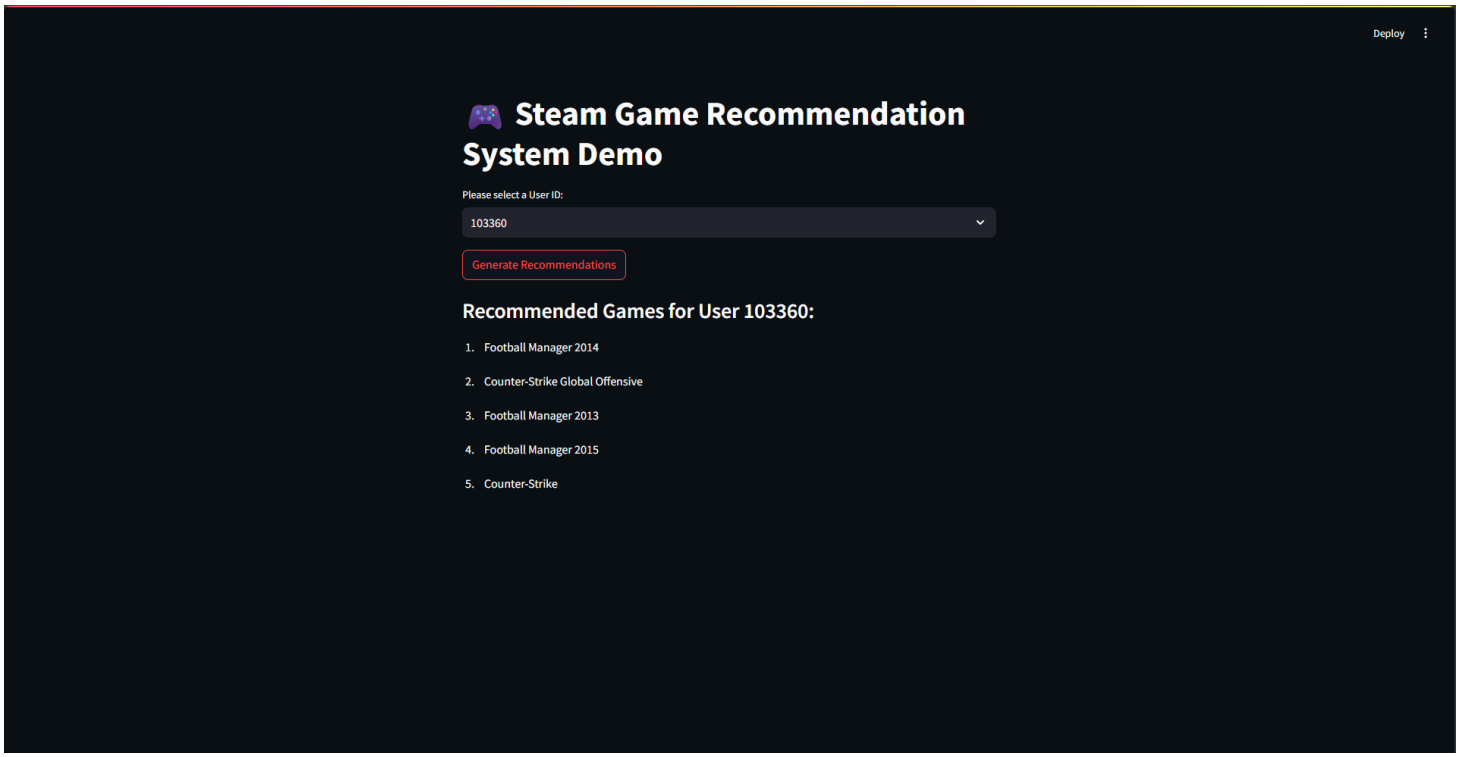
Rank	Model	RMSE	MAE	vs Baseline
1	Bias-SVD	**122.83**	**29.28**	**+5.08%**
2	User-CF	125.06	45.49	+3.36%
3	FunkSVD-v2	126.89	17.70	+1.95%

Rank	Model	RMSE	MAE	vs Baseline
4	Item-CF	126.95	45.24	+1.90%
5	Global Mean	129.41	53.98	0.00%
6	User Avg	152.45	57.80	-17.81%

Key Findings:

- Bias-SVD is optimal, reducing RMSE by 5.08% compared to baseline
- User-CF is second-best, providing 3.36% improvement with high computational efficiency
- FunkSVD-v2 achieves baseline-level performance with moderate computational cost

5.2 Qualitative Analysis



Bias-SVD Advantages

Bias-SVD achieves optimal performance through three-level factor decomposition:

1. **Global Baseline** (μ): Stable prediction baseline
2. **Individual Bias** (b_u, b_i): Captures user preferences and game popularity
3. **Interaction Features** (P[u]·Q[i]): Learns user-game matching

This design allows factor matrices to focus on nonlinear interactions and avoid fitting absolute values.

User-CF Advantages

- Based on user similarity, easy to interpret and debug
- Computationally efficient with support for online updates and real-time recommendations
- No explicit feature engineering required

FunkSVD-v2 Characteristics

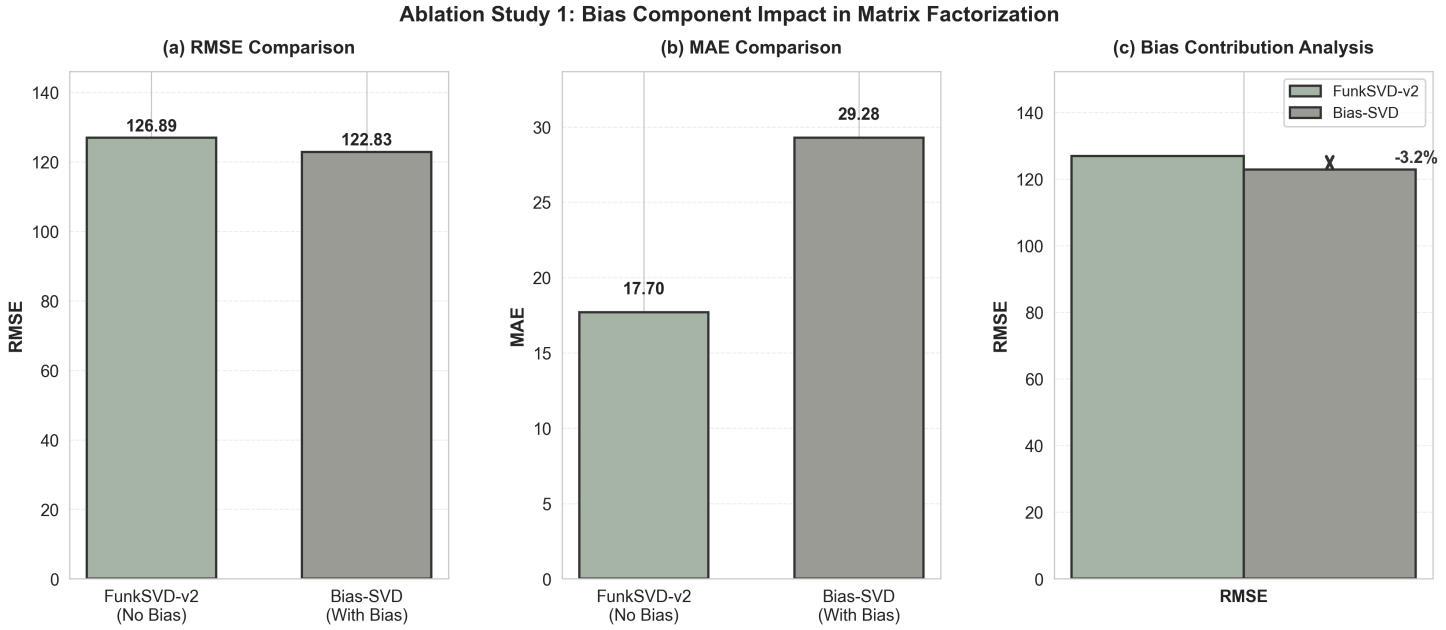
- Adds global mean constraint to avoid gradient drift
- Reduces learning rate (0.003) and increases regularization (0.05) for stable convergence
- Achieves baseline-level performance under constraint optimization

5.3 Ablation Studies

Ablation Study 1: Impact of Bias Terms

Model	RMSE	MAE	Bias Contribution
FunkSVD-v2 (No Bias)	126.89	17.70	-
Bias-SVD (With Bias)	122.83	29.28	**-3.15%**

Conclusion: Explicit bias decomposition for users and items further improves accuracy by 3.15%.

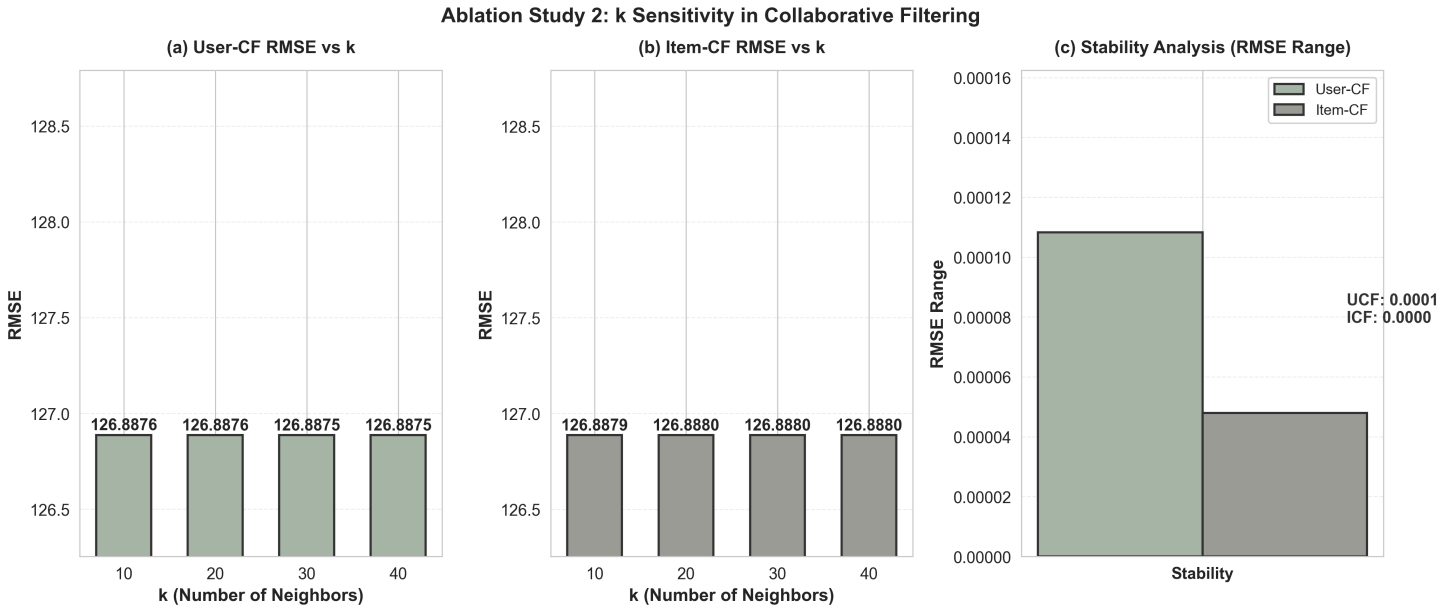


Ablation Study 2: k Sensitivity Analysis (CF Algorithms)

k Value	User-CF RMSE	Item-CF RMSE	Stability
10	126.887576	126.887927	-
20	126.887560	126.887957	✓ **Optimal**
30	126.887514	126.887966	-
40	126.887468	126.887975	-

Conclusion: CF models show negligible sensitivity to k values (Δ<0.001%).

Why k=20 (practical choice): Although the measured RMSE values vary only in the sixth decimal place across k∈{10,20,30,40} (maximum RMSE variation ≤ 0.00011), we select k=20 as the practical operating point because it offers near-best accuracy for both User-CF and Item-CF while keeping neighbor-search cost moderate. Increasing k beyond 20 yields diminishing accuracy improvements but increases runtime and the risk of including less relevant neighbors; choosing k=20 therefore provides a robust accuracy/efficiency trade-off suitable for production or large-scale evaluation.



5.4 Key Insights

1. **Necessity of Global Constraints**: Global mean (μ) serves as prediction baseline, stabilizing factor learning trajectory and preventing gradient explosion
2. **Importance of Hyperparameter Tuning**: FunkSVD-v2 achieves stability through joint optimization of learning rate (0.01→0.003) and regularization (0.02→0.05)
3. **Effectiveness of Bias Decomposition**: Bias-SVD improves 3.15% over FunkSVD-v2, proving the value of explicit factor decomposition

4. **Robustness of CF Models**: CF performance varies <0.001% within k∈[10,40], demonstrating excellent stability and generalization ability

Conclusion

Bias-SVD provides the best accuracy in our experiments while User-CF offers a lightweight, interpretable alternative with competitive performance. CF models are stable across k values; for production we recommend Bias-SVD for accuracy-critical tasks and User-CF for low-latency scenarios. Future work should explore side-information, temporal dynamics, and cold-start solutions.

Report Date: 2025-12-02 | **Dataset**: Steam-200k | **Models Evaluated**: 6