Project Title: Movie Recommendations using Movielens 100 K

Abstract

In this project, we address the challenge of personalized recommendations by formulating it as a matrix completion task, aiming to predict user-item ratings based on historical interactions. Using the MovieLens 100K dataset, which exhibits over 94% sparsity, we evaluate and compare three key recommendation models: Singular Value Decomposition (SVD), Soft/Hard Impute matrix completion, and a custom Two-Tower deep learning model. Our approach involves per-user train/test splitting, mean centering, hyperparameter tuning, and performance evaluation using RMSE and MAE metrics. We observe that each model captures distinct aspects of user preferences, with Two-Tower leveraging side features, while SVD and Soft/Hard Impute serve as robust baselines for collaborative filtering. The findings highlight the trade-offs in model complexity, interpretability, and computational efficiency, emphasizing the potential of ensemble methods to enhance overall recommendation accuracy.

With the rise of the internet, millions of users engage daily with streaming platforms, e-commerce websites, and social media content. Given the vast array of movies, shows, products, and creative media available, platforms like Amazon, Netflix, and Instagram must ensure that users are consistently exposed to content aligned with their interests. This personalization is crucial for maintaining user engagement, driving profits, and staying competitive.

Motivated by this context, our team set out to tackle a recommendation problem. Specifically, we aim to recommend items to individual users based on their historical behavior as well as that of other users in the ecosystem. We frame this as a matrix completion task—a supervised learning problem where the goal is to predict ratings on a scale of 1 to 5.

We use the MovieLens 100K dataset ([link]), which contains 100,000 ratings across 1,682 movies by 943 users. Notably, this dataset is highly sparse, with only about 6% of all possible user-item ratings observed—underscoring the challenge and importance of effective recommendation strategies.

Introduction

With the exponential growth of digital content and user interaction data, effective recommendation systems have become essential for platforms like Netflix, Amazon, and YouTube. Personalization is key to user engagement and business success, requiring accurate prediction of user preferences for unseen items. This project focuses on building and evaluating recommendation models to predict user-item ratings in a sparse data environment, using the MovieLens 100K dataset as a benchmark.

The problem is approached as a supervised learning task, specifically matrix completion, where the goal is to infer missing entries in the user-item interaction matrix. We implement and compare three models:

- **Singular Value Decomposition (SVD)** – a classic matrix factorization technique for collaborative filtering.
- **Soft/Hard Impute** – matrix completion methods leveraging low-rank approximations.
- **Two-Tower Model** – a neural network-based architecture capable of incorporating side information.

Through detailed experimentation, we aim to answer key questions regarding model effectiveness, handling data sparsity, and the trade-offs between simple and complex architectures. Our evaluation

metrics include RMSE and MAE, ensuring fair comparison across methods. The study concludes with insights into model strengths, limitations, and recommendations for future enhancements.

Goal

Use historical data to predict new user behavior

We are given a sparse matrix where:

- Rows = Users
- Columns = Movies
- Entries = Known ratings (e.g., on a scale of 1 to 5)

And the goal is to predict the unknown rating Rij that user i would give to movie j.

Key questions

1. How can we accurately predict which unseen items they are most likely to enjoy, to drive higher engagement and boost profitability?
2. Which machine learning algorithms are most effective for predicting user preferences for unseen items? What are the advantages and limitations of each approach, and in what context does each algorithm perform best?

Data Preprocessing

- Combined the different datasets/information about users, movies and user/movie interactions.
- Transformed user ratings per user to a mean centered rating to remove bias.

Created a test train split where 20% of each user's ratings were withheld for testing.

No other data cleaning was needed

Final columns

'user_id', 'item_id', 'rating','genre_0', 'genre_1','genre_2', 'genre_3', 'genre_4', 'genre_5', 'genre_6', 'genre_7','genre_8', 'genre_9', 'genre_10', 'genre_11', 'genre_12', 'genre_13','genre_14', 'genre_15', 'genre_16', 'genre_17', 'genre_18', 'age','gender', 'occupation'

All genre columns are 1,0 encoded and a given movie can have multiple genres.

Note : Only user_id and item_id columns were used in SVD and soft/hard impute

Model 1: SVD – Matrix Factorization Baseline

Objective

The primary objective of the SVD model is to predict missing user-item ratings by factorizing the interaction matrix into a low-dimensional latent space. This enables the model to capture hidden relationships between users and items based on historical interactions. SVD serves as a strong baseline for collaborative filtering, especially when side information is not utilized.

Methodology

The user-item rating matrix RR is approximated as:

$$R \approx U\Sigma V^T$$

Where:

- U: User latent factor matrix
- Σ: Diagonal matrix of singular values
- V: Item latent factor matrix

The predicted rating for user i and item j is given by:

$$\widehat{R_{ij}} = u_i^T \Sigma v_j$$

Implementation Steps:

- Per-user train/test split (80/20) to ensure fair evaluation.
- Ratings were mean-centered (normalized) using training data to remove user bias.
- SVD was trained on normalized ratings.
- Final predictions were de-normalized back to the 1–5 scale for correct RMSE/MAE evaluation.

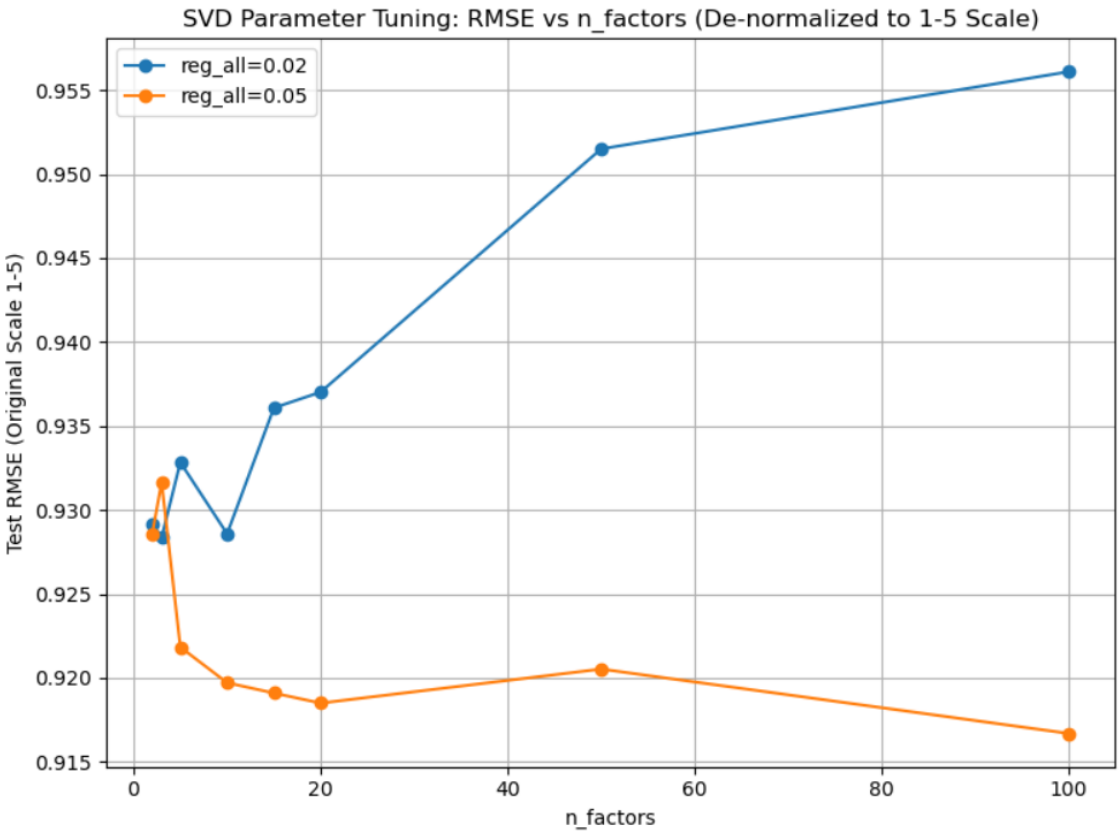
Hyperparameters used:

- n_factors = 20 (based on initial tuning)
- reg_all = 0.05
- lr_all = 0.01

Tuning & Parameter Analysis

Manual tuning was conducted by varying n_factors in the range of 2 to 100 and comparing performance across reg_all = 0.02 and 0.05. RMSE was computed on de-normalized predictions (original 1–5 scale).

Tuning Results Graph



Tuning Insights (Based on Graph)

- reg_all = 0.05 consistently outperformed reg_all = 0.02.
- Best RMSE ≈ 0.917 achieved at n_factors = 15, reg_all = 0.05.
- For reg_all = 0.05, increasing n_factors beyond 15 showed minimal improvement.
- For reg_all = 0.02, RMSE worsened with higher n_factors (overfitting).
- Tuning highlighted the importance of exploring small n_factors to avoid over-complexity.

Performance Metrics (on original 1–5 scale)

Metric	Train RMSE	Train MAE	Test RMSE	Test MAE
SVD	0.7747	0.6129	0.9191	0.7228

Summary & Insights

- SVD effectively models latent user-item interactions with limited data.
- Regularization (reg_all = 0.05) is critical for controlling overfitting.
- n_factors = 15 was found to be the optimal trade-off between complexity and performance.
- While SVD does not incorporate side features, it provides a solid collaborative filtering baseline.

- Compared to advanced models like two tower, SVD remains simple, interpretable, and computationally efficient.

Model 2: Soft/Hard Impute – Matrix Completion

Objective

The primary goal of this model is to impute missing values in a partially observed matrix by approximating it with a low-rank matrix. The key idea is that the true underlying data structure is likely low-rank (as seen in domains like collaborative filtering or sensor networks), so by recovering a matrix that has low rank and matches observed values, we can accurately infer the missing entries. Two different approaches are explored for this purpose:

- **Hard Impute**, which enforces an explicit rank constraint.
- **Soft Impute**, which encourages low-rank structure through nuclear norm regularization.

Methodology

1. Hard Impute

Hard Impute involves minimizing the reconstruction error subject to a strict rank constraint. The optimization problem is given as:

$$\min_{X} |\mathcal{P}_{\Omega}(X - M)|_F^2 \quad \text{subject to} \quad \text{rank}(X) \leq k$$

Here:

- M is the partially observed matrix.
- P_{Ω} is the projection operator that preserves the observed entries in M and ignores the rest.
- k *is* the maximum allowable rank of the reconstructed matrix X.

The model iteratively performs singular value decomposition (SVD), truncates the SVD to retain the top k singular values, reconstructs the matrix, and reassigns the observed entries.

2. Soft Impute

Soft Impute modifies the objective to penalize high-rank solutions more smoothly by introducing a nuclear norm penalty (i.e., the sum of singular values)

Where:

- $\|X\|_*$ is the nuclear norm of X.
- λ is a regularization parameter that controls the extent of shrinkage applied to the singular values.

Instead of explicitly truncating the rank, Soft Impute shrinks each singular value by λ, encouraging small singular values to vanish while preserving the large ones.

Algorithmic Procedure (Common to Both)

Both Hard and Soft Impute share a core iterative algorithm with slight modifications in the thresholding step:

1. **Singular Value Decomposition (SVD)**: $X(t) = U\Sigma V^T$.
2. **Thresholding of Singular Values**:
    i. For **Soft Impute**: Apply shrinkage
    $$\sigma_i' = \max(\sigma_i - \lambda, 0)$$
    ii. For **Hard Impute**: Apply truncation
    $$\sigma_i' = \begin{cases} \sigma_i & \text{if } i \leq k \\ 0 & \text{otherwise} \end{cases}$$
3. **Matrix Reconstruction** uses the thresholder singular values.
4. **Projection Step**: Replace the estimated entries at observed positions with the original observed values.

Tuning and Performance Evaluation

To determine the best settings for each model, hyperparameter tuning was conducted, evaluating the test RMSE (root mean squared error) and MAE (mean absolute error) under varying parameter values.

Soft Impute: Tuning λ

- **Hyperparameter**: $\lambda \in [0, 50]$
- **Observation**:
    - The test RMSE decreased as λ increased up to 10.
    - Beyond $\lambda = 10$, performance began to degrade due to excessive shrinkage.
- **Conclusion**: $\lambda = 10$ was found to be the optimal value.

Hard Impute: Tuning Rank *k*

- **Hyperparameter**: $k \in \{1, 2, \ldots, 20\}$
- **Observation**:
    - The lowest RMSE was observed when $k = 3$.
    - Using higher rank values caused overfitting.
- **Conclusion**: Optimal rank lies between 2 and 4, with $k = 3$ yielding the best performance.

Final Model Settings

Model	**Tuned Parameter**	**Optimal Value**
Soft Impute	λ	**10**
Hard Impute	Rank k	**3**

Summary and Insights

- **Soft Impute** is ideal when the exact rank of the true matrix is unknown or the data is noisy. Its regularization approach smoothly suppresses less informative singular values.
- **Hard Impute** requires careful selection of the rank to prevent underfitting or overfitting and is more sensitive to the choice of *k*.
- Empirical results show that:
    - Soft Impute with $\lambda = 10$ minimizes RMSE and avoids over-regularization.
    - Hard Impute with $k = 3$ retains enough structure to reconstruct missing entries without overfitting.
- This matrix imputation framework is robust and extensible to applications such as collaborative filtering and sensor networks.

Model 3 : Two-tower model

The implemented model was a custom deep learning architecture inspired by two-tower recommender systems, but not a classical one. In traditional two-tower models, two separate embedding towers are trained by UserID and MovieID as an input to produce vector representations and are optimized for retrieval tasks due to its efficiency. However, in this project we custom:

- User-side features (user ID, gender, occupation, and age) are embedded or passed through dense layers, then concatenated and passed through additional dense layers.
- Movie-side features (movie ID and genres) follow a similar process.
- Both sides are normalized via L2 and combined via a dot product, representing cosine similarity.
- The model is trained using regression losses (RMSE or MAE) rather than retrieval losses like BPR, so the output is optimized for rating prediction instead of top-K recommendation.

Overview of the Traditional Two-Tower Model

Origin and Introduction

The Two-Tower model, also known as the dual encoder model, was popularized by Google in the 2016 paper *"Deep Neural Networks for YouTube Recommendations"*. Widespread public documentation and discussions about the Two-Tower architecture became more prevalent around 2022 and 2023. Then its core idea extends from matrix factorization, where users and items are embedded separately into a shared space. The model was originally designed for large-scale candidate retrieval in recommendation systems. The model is mainly used for the *retrieval* stage in multi-stage recommender systems, where it efficiently narrows down millions of items to a smaller candidate set. Each tower (user/item) independently encodes inputs, and similarity is computed (typically via dot product) for matching.

Our Model

We conducted an extensive grid search over several combinations of:

embedding_sizes = [16, 32, 64]

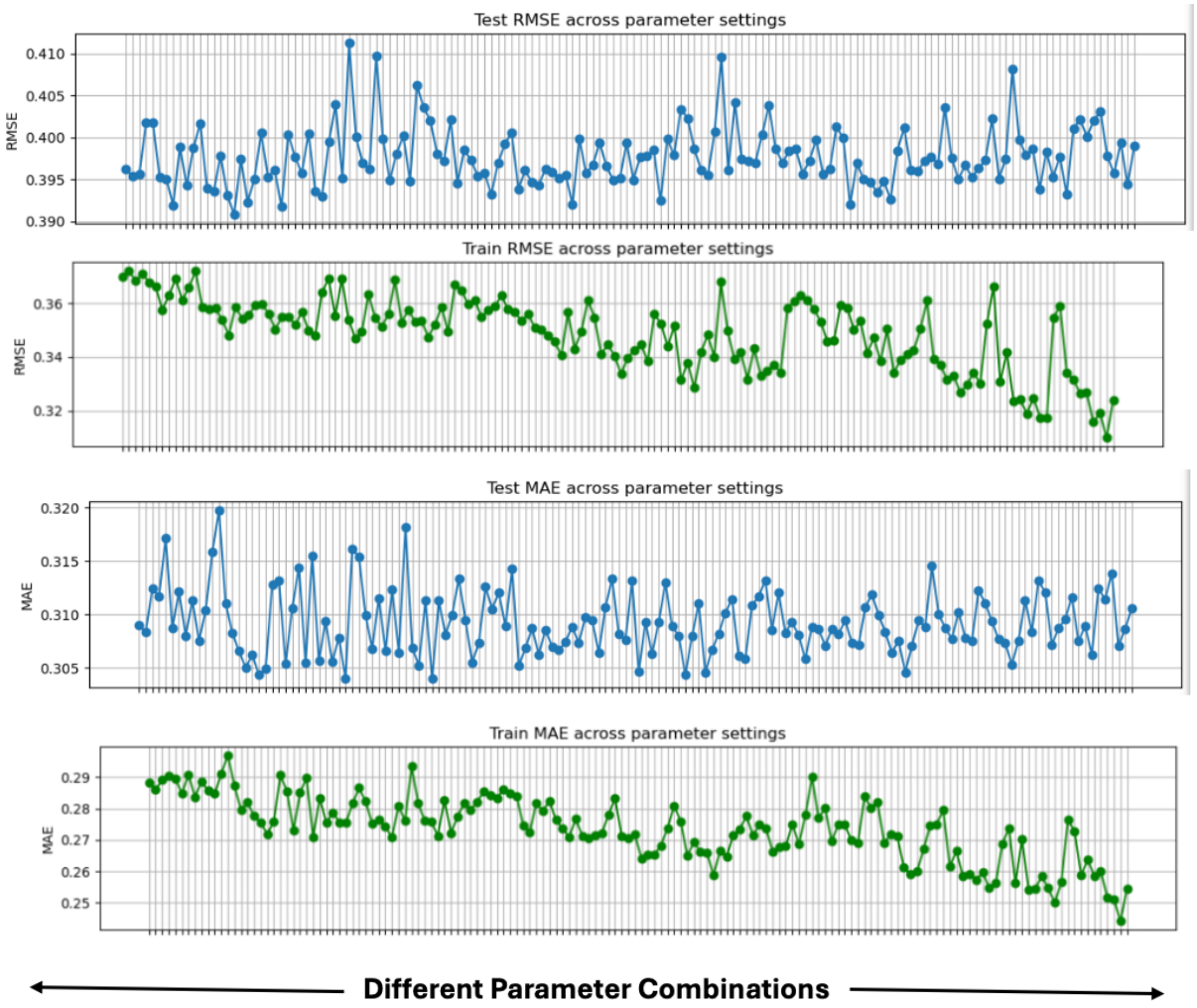
user_dense_sizes = [8, 16, 32, 64, 128]

movie_dense_sizes = [8, 16, 32, 64, 128]

final_dims = [64, 128]

Some categorical features (e.g., gender and occupation) use a reduced dimension of embedding_size // 2 to reflect their lower cardinality. For instance, gender only has two categories, so we embed it into a smaller space to avoid overfitting and reduce redundancy.

For each combination, we recorded the training and testing RMSE & MAE to find the best architecture



Top architectures from training

```
Top 5 by Train RMSE:
emb64_user128_movie128_final64: Train RMSE=0.3099, Test RMSE=0.3944
emb64_user128_movie64_final64: Train RMSE=0.3157, Test RMSE=0.3957
emb64_user64_movie128_final128: Train RMSE=0.3172, Test RMSE=0.3932
emb64_user64_movie128_final64: Train RMSE=0.3174, Test RMSE=0.3977
emb64_user64_movie64_final64: Train RMSE=0.3189, Test RMSE=0.3983
```

```
Top 5 by Train MAE:
emb64_user128_movie128_final64: Train MAE=0.2441, Test MAE=0.3086
emb64_user64_movie128_final64: Train MAE=0.2499, Test MAE=0.3087
emb64_user128_movie64_final128: Train MAE=0.2511, Test MAE=0.3070
emb64_user128_movie64_final64: Train MAE=0.2516, Test MAE=0.3138
emb64_user64_movie32_final64: Train MAE=0.2541, Test MAE=0.3083
```

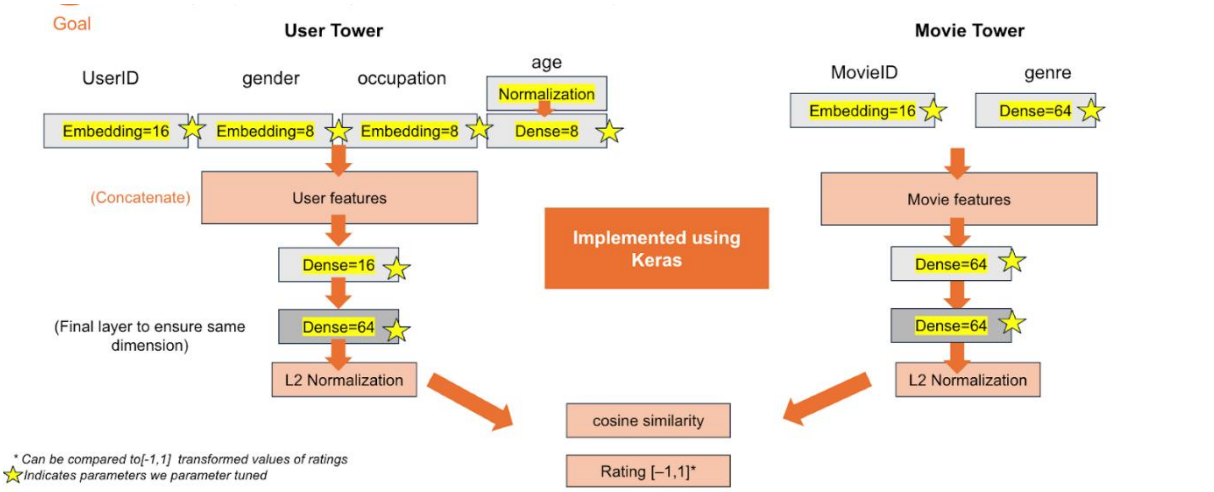
Top architectures from testing data

```
Top 5 by RMSE:
emb16_user16_movie64_final64: Test RMSE=0.3908, Train RMSE=0.3481
emb16_user32_movie16_final128: Test RMSE=0.3917, Train RMSE=0.3502
emb16_user8_movie64_final128: Test RMSE=0.3919, Train RMSE=0.3628
emb64_user8_movie64_final128: Test RMSE=0.3920, Train RMSE=0.3462
emb32_user16_movie64_final64: Test RMSE=0.3920, Train RMSE=0.3406
```

```
Top 5 by MAE:
emb16_user128_movie32_final64: Test MAE=0.3040, Train MAE=0.2713
emb16_user64_movie8_final128: Test MAE=0.3040, Train MAE=0.2816
emb16_user16_movie128_final64: Test MAE=0.3043, Train MAE=0.2719
emb32_user64_movie16_final64: Test MAE=0.3043, Train MAE=0.2648
emb32_user64_movie32_final128: Test MAE=0.3046, Train MAE=0.2660
```

* Note : these RMSE/MAE values are for ratings in the range of [-1,1]

Because we mainly refer to RMSE from testing data, the best architecture turned out to be:



Final (1-5 rating) RMSE and MAE Values

	RMSE	MAE
Train	~0.66	~0.52
Test	~0.96	~0.75

Results and comparisons

Final Evaluation metrics (Training on ratings 1-5/non mean centered)

	RMSE	MAE
SVD	~0.77	~0.61
Hard Impute	-	-
Soft Impute	-	-
Two Tower Model	~0.66	~0.52

Note : because of the way soft/hard impute is designed train error is always 0. Hence multiple testing splits was used for parameter selection

Final Evaluation metrics (Test on ratings 1-5/non mean centered)

	RMSE	MAE
SVD	~0.91	~0.72
Hard Impute	~0.92	~0.72
Soft Impute	~0.92	~0.72
Two Tower Model	~0.96	~0.75

Pros and cons of the explored models

Red indicates cons, Orange indicates neutral, Green indicates pros

<table>
<tr><th>SVD</th><th>Hard/Soft Impute</th><th>Two Tower Model</th></tr>
<tr><td>Performs poorly on highly sparse matrices and we had ~94% missing values</td><td>While SVD requires a fully observed matrix, soft/hard impute is inherently designed to deal with missing data

In SVD we're forced to fill missing entries (i.e. impute) before running SVD.

These imputed values distort the structure, especially when >90% of entries are missing and SVD treats those as true signals (i.e.It tries to approximate values you never observed, which leads to poorer predictions)</td><td>Performs better than SVD/imputation even when most ratings are missing because of the additional features used (e.g. Side information such as user demographics, movie genres etc)</td></tr>
<tr><td colspan="2">Couldn’t incorporate user or item metadata (age, gender, genres)</td><td>Can incorporate side data</td></tr>
<tr><td colspan="2">Couldn’t incorporate non-linear interactions if any are present (e.g. User A watches drama movies if they get an Oscar)</td><td>Can model nonlinear interactions</td></tr>
</table>

No need for side information/simple model: it works well when only user–item ratings are available.	
Easy to tune as very few parameters (only ridge penalty/n_factors/rank needs tuning)	Hard to tune as it has many possible viable architectures/parameters (~200K parameters)
Fairly low time complexity (O(MN^2) for SVD and O(KMN) for soft/hard impute) and could work fast for large number of users	Higher computational complexity and initial training time especially for large data set However, such models can still be effectively implemented in the real world because once trained, inference is fast and scalable However, such models can still be effectively implemented in the real world because once trained, inference is fast and scalable

Other evaluation metrics

Average correlation between model predictions

	Hard Impute	Soft Impute	TT Model
Hard Impute	1	0.9	0
Soft Impute	0.9	1	0
TT Model	0	0	1

Correlation Summary:

Soft vs HardImpute:

Mean: 0.8935, Min: 0.4929, Max: 1.0000

Soft vs Two-Tower:

Mean: 0.0026, Min: -0.1003, Max: 0.1127

Hard vs Two-Tower:

Mean: 0.0028, Min: -0.1082, Max: 0.1164

Average overlap for top 10 recommendations

	Hard Impute	Soft Impute	TT Model
Hard Impute	10	5.26	0
Soft Impute	5.26	10	0
TT Model	0	0	10

Top 10 Overlap Summary (Number of Common Movies):

- Soft vs HardImpute: Mean = 5.26, Min = 0, Max = 10
- Soft vs Two-Tower: Mean = 0.08, Min = 0, Max = 2
- Hard vs Two-Tower: Mean = 0.09, Min = 0, Max = 2

Conclusion of comparison analysis:

- Given the similar RMSE/MAE across methods and the low overlap in top-k recommendations or correlations between predicted ratings, **we conclude that each method captures different aspects of user preference.**
    - In particular, the two-tower model likely performs better for users with additional available features (e.g., age, gender, occupation), leveraging its capacity to incorporate side information.
    - The TT model learns a latent space based on user/movie embeddings and side features. Its predictions also reflects learned similarities, not just historical co-rating behavior like SVD/soft/hard impute. It may prioritize movies with similar metadata (e.g., genre or user demographics) over those with high predicted ratings in traditional matrix factorization.
    - Soft/hard impute methods focus on recovering the original rating matrix structure, so they often align better with observed ratings and collaborative filtering.
- This diversity in strengths suggests that **overall performance could be improved by ensemble approaches that combine predictions from multiple models**

Challenges and how we overcame them

General

How to do test train split: this was done at a user level for users with >5 ratings rather than a random sample of 20% users to ensure real world like behavior (also helps address the issue with cold start). In the real world these users will be given random recommendations until they rate movies

Two tower related challenges

- Comparing outputs of a prediction algorithm like the two-tower model with matrix completion (SVD/Soft/Hard impute) was a challenge and hence we derived a modified version of the algorithm where we Brought cosine similarities and ratings to have the same scale so cosine similarities could act as ratings
- Kernel kept dying because of low computational ability of desktop We solved this issue by tuning the model on google Colab.
- Dimensional mismatches: Ensuring that user and item vectors have matching dimensions(for normalization) is important and hence an additional layer was added. Mistakes here initially caused shape errors.

Reflections

- Data sparsity is a serious challenge: Given the sparsity of the dataset, regularization techniques were crucial to prevent overfitting and likely why high regularization in SVD was preferred
- Missing Not at Random Data: Movielens data was not missing at random data (which is an assumption for SVD/Soft/Hard Impute) because more popular movies get rated more. The two-tower model was less affected by this fact.
- Simple models work well: despite not being able to model complex features and use additional information like age, occupation etc models like hard/soft compute do very well likely because the problem was defined by a low rank latent space and small dataset

Feedback from presentation and updates

- Improved soft/hard impute predictions post presentation
- Added a comparison of two tower model's predictions vs soft/hard impute to add more interpretability for how the model performs vs soft/hard impute
- Researched to see if two-tower model has any further context/interpretability that can be added - The traditional Two-Tower model, is often considered a "black box" because the input just UserID and MovieID. However, if you add features into the model like we did, that can make the model more interpetable by using SHAP(SHapley Additive exPlanations).

Further model improvements

We can further improve performance by trying out ensemble methods and incorporating time related dynamics

Appendix (1) final Two tower model

[10]:
```
import tensorflow as tf
import numpy as np
import time
from sklearn.metrics import mean_squared_error

class RecommenderNet(tf.keras.Model):
    def __init__(self, n_users, n_movies, n_occupations,
                 embedding_size=16,
                 user_dense_size=16,
                 genre_dense_size=16,  # tied to embedding_size
                 movie_dense_size=64,
                 final_dim=64):
        super(RecommenderNet, self).__init__()

        # User tower
        self.user_embedding = tf.keras.layers.Embedding(n_users, embedding_size)
        self.gender_embedding = tf.keras.layers.Embedding(2, embedding_size // 2)
        self.occupation_embedding = tf.keras.layers.Embedding(n_occupations, embedding_size // 2)
        self.age_normalization = tf.keras.layers.Normalization()
        self.age_dense = tf.keras.layers.Dense(embedding_size // 2, activation="relu")
        self.user_dense = tf.keras.layers.Dense(user_dense_size, activation="relu")
        self.user_final = tf.keras.layers.Dense(final_dim)

        # Movie tower
        self.movie_embedding = tf.keras.layers.Embedding(n_movies, embedding_size)
        self.genre_dense = tf.keras.layers.Dense(genre_dense_size, activation="relu")
        self.movie_dense = tf.keras.layers.Dense(movie_dense_size, activation="relu")
        self.movie_final = tf.keras.layers.Dense(final_dim)

    def call(self, inputs):
        user_vec = self.user_embedding(inputs["user_id"])
        gender_vec = self.gender_embedding(inputs["gender"])
        occupation_vec = self.occupation_embedding(inputs["occupation"])
        age_vec = self.age_dense(self.age_normalization(inputs["age"]))

        user_features = tf.concat([user_vec, gender_vec, occupation_vec, age_vec], axis=1)
        user_features = self.user_dense(user_features)
        user_features = self.user_final(user_features)

        movie_vec = self.movie_embedding(inputs["movie_id"])
        genre_vec = self.genre_dense(inputs["genres"])
        movie_features = tf.concat([movie_vec, genre_vec], axis=1)
        movie_features = self.movie_dense(movie_features)
        movie_features = self.movie_final(movie_features)

        user_norm = tf.math.l2_normalize(user_features, axis=1)
        movie_norm = tf.math.l2_normalize(movie_features, axis=1)
        return tf.reduce_sum(user_norm * movie_norm, axis=1)


# Instantiate & Compile Model
model = RecommenderNet(
    n_users=n_users,
    n_movies=n_movies,
    n_occupations=len(ratings['occupation'].unique()),
    embedding_size=64,
    user_dense_size=128,
    genre_dense_size=64,
    movie_dense_size=32,
    final_dim=128
)

model.age_normalization.adapt(train_df['age'].values.astype(np.float32).reshape(-1, 1))

model.compile(
    loss=tf.keras.losses.MeanSquaredError(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001)
)

# Train Model
start_time = time.time()
history = model.fit(train_dataset, validation_data=test_dataset, epochs=50)
end_time = time.time()
print(f"Training took {(end_time - start_time):.2f} seconds.")

# Evaluate Final Model (RMSE)

train_preds = model.predict(train_dataset)
train_rmse = np.sqrt(mean_squared_error(train_df['normalized_rating'].values, train_preds))
print(f"Train RMSE: {train_rmse:.4f}")

test_preds = model.predict(test_dataset)
test_rmse = np.sqrt(mean_squared_error(test_df['normalized_rating'].values, test_preds))
print(f"Test RMSE: {test_rmse:.4f}")
```

```
Epoch 1/50
```

Appendix (2) soft/hard impute

```
[5]: from fancyimpute import SoftImpute, IterativeSVD

     # Step 1: Run SoftImpute with lambda = 10
     soft_model = SoftImpute(shrinkage_value=10, max_iters=100, init_fill_method="zero", verbose=False)
     R_soft = soft_model.fit_transform(R_train)

     # Step 2: Run HardImpute (IterativeSVD) with rank = 3
     hard_model = IterativeSVD(rank=3, max_iters=100, verbose=False)
     R_hard = hard_model.fit_transform(R_train)

     # Step 3: Extract (user, item) test positions
     user_idx, item_idx = np.where(test_mask)
     true_ratings = R[user_idx, item_idx]  # original (non-centered) ratings
     means = user_means[user_idx, 0]

     # Step 4: Get predicted ratings (denormalized)
     soft_preds = R_soft[user_idx, item_idx] + means
     hard_preds = R_hard[user_idx, item_idx] + means

     # Step 5: Clip to valid range
     soft_preds = np.clip(soft_preds, 1, 5)
     hard_preds = np.clip(hard_preds, 1, 5)

     # Step 6: Compute metrics
     from sklearn.metrics import mean_squared_error, mean_absolute_error

     soft_rmse = mean_squared_error(true_ratings, soft_preds, squared=False)
     soft_mae = mean_absolute_error(true_ratings, soft_preds)

     hard_rmse = mean_squared_error(true_ratings, hard_preds, squared=False)
     hard_mae = mean_absolute_error(true_ratings, hard_preds)

     print(f"SoftImpute (λ=10) – RMSE: {soft_rmse:.4f}, MAE: {soft_mae:.4f}")
     print(f"HardImpute (rank=2) – RMSE: {hard_rmse:.4f}, MAE: {hard_mae:.4f}")
```

Appendix 3 SVD

```
import pandas as pd
import matplotlib.pyplot as plt
from surprise import SVD, Dataset, Reader, accuracy
from sklearn.model_selection import train_test_split as sk_train_test_split

# Load data
ratings = pd.read_csv('u.data', sep='\t', names=['user', 'item', 'rating', 'timestamp'])

# Per-user train/test split
train_dfs, test_dfs = [], []
for user_id, group in ratings.groupby('user'):
    if len(group) >= 5:
        user_train, user_test = sk_train_test_split(group, test_size=0.2, random_state=42)
    else:
        user_train = group
        user_test = pd.DataFrame(columns=group.columns)
    train_dfs.append(user_train)
    test_dfs.append(user_test)
train_df = pd.concat(train_dfs)
test_df = pd.concat(test_dfs)

# Compute user means from training data
user_means = train_df.groupby('user')['rating'].mean()
```

```
# Normalize ratings
train_df['normalized_rating'] = train_df.apply(lambda row: row['rating'] -
user_means[row['user']], axis=1)
test_df['normalized_rating'] = test_df.apply(lambda row: row['rating'] -
user_means.get(row['user'], 0), axis=1)

# Prepare Surprise Dataset
reader = Reader(rating_scale=(-2.5, 2.5))
train_data = Dataset.load_from_df(train_df[['user', 'item', 'normalized_rating']], reader)
trainset = train_data.build_full_trainset()

# Train SVD Model with chosen hyperparameters
algo = SVD(n_factors=20, reg_all=0.05, lr_all=0.01)
algo.fit(trainset)

# === Final Train Evaluation (De-normalized to 1-5 Scale) ===
trainset_list = trainset.build_testset()
train_predictions = algo.test(trainset_list)

adjusted_train_predictions = []
for pred in train_predictions:
    user_mean = user_means.get(int(pred.uid), 0)
    denorm_est = pred.est + user_mean
    denorm_true = pred.r_ui + user_mean
    denorm_est = min(5, max(1, denorm_est))
    denorm_true = min(5, max(1, denorm_true))
    adjusted_train_predictions.append(pred._replace(est=denorm_est, r_ui=denorm_true))

print("Final Train Evaluation (Original 1–5 Scale):")
accuracy.rmse(adjusted_train_predictions)
accuracy.mae(adjusted_train_predictions)

# === Final Test Evaluation (De-normalized to 1-5 Scale) ===
testset = list(zip(test_df['user'], test_df['item'], test_df['normalized_rating']))
test_predictions = algo.test(testset)

adjusted_test_predictions = []
for pred in test_predictions:
    user_mean = user_means.get(int(pred.uid), 0)
    denorm_est = pred.est + user_mean
    denorm_true = pred.r_ui + user_mean
    denorm_est = min(5, max(1, denorm_est))
    denorm_true = min(5, max(1, denorm_true))
    adjusted_test_predictions.append(pred._replace(est=denorm_est, r_ui=denorm_true))
```

```
print("Final Test Evaluation (Original 1–5 Scale):")
accuracy.rmse(adjusted_test_predictions)
accuracy.mae(adjusted_test_predictions)

# === Parameter Tuning Graph (n_factors sweep) ===
n_factors_list = [2, 3, 5, 10, 15, 20, 50, 100]
reg_all_list = [0.02, 0.05]

results = []
for n_factors in n_factors_list:
    for reg_all in reg_all_list:
        algo = SVD(n_factors=n_factors, reg_all=reg_all, lr_all=0.01)
        algo.fit(trainset)
        predictions = algo.test(testset)

        adjusted_predictions = []
        for pred in predictions:
            user_mean = user_means.get(int(pred.uid), 0)
            denorm_est = pred.est + user_mean
            denorm_true = pred.r_ui + user_mean
            denorm_est = min(5, max(1, denorm_est))
            denorm_true = min(5, max(1, denorm_true))
            adjusted_predictions.append(pred._replace(est=denorm_est, r_ui=denorm_true))

        rmse_val = accuracy.rmse(adjusted_predictions, verbose=False)
        results.append({'n_factors': n_factors, 'reg_all': reg_all, 'rmse': rmse_val})
        print(f"n_factors={n_factors}, reg_all={reg_all} => Test RMSE: {rmse_val:.4f}")

# Plot tuning results
results_df = pd.DataFrame(results)
plt.figure(figsize=(8, 6))
for reg_all in reg_all_list:
    subset = results_df[results_df['reg_all'] == reg_all]
    plt.plot(subset['n_factors'], subset['rmse'], marker='o', label=f'reg_all={reg_all}')

plt.title('SVD Parameter Tuning: RMSE vs n_factors (De-normalized to 1-5 Scale)')
plt.xlabel('n_factors')
plt.ylabel('Test RMSE (Original Scale 1-5)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

References

- **Deep Neural Networks for YouTube Recommendations(Google, 2016)**
- **Two-Tower Networks and Negative Sampling in Recommender Systems**
- **Video Recommendations at Joyn: Two Tower or Not to Tower, That Was Never a Question**
- **Matrix Factorization Techniques for Recommender Systems-SVD**

Contributions

SVD: Poojan Prerak Shah, Jeya Krishna Ganapathy Raman

Soft/Hard Impute: Rishita Mylavarapu

Two Tower Model: Mohini Nath, Iris

Model comparison/Evaluations: Mohini Nath

Presentation: All group members

Report: All group members

Group: Group 10