CS 414 Final Report: Recommender Systems

Name: Roqia Alirezaei

Project Title: Implementation and Evaluation of Recommender System Models on MovieLens and Yelp Datasets

Abstract

This project investigates and compares fundamental algorithms used in recommender systems, focusing on **Item-Based Collaborative Filtering** and the **Baseline Bias Models**—both the statistical version and its optimized counterpart using **Gradient Descent**. All models were **implemented from scratch** using Python and evaluated on two real-world datasets: **MovieLens 100K** and a subset of the **Yelp reviews** dataset. These datasets represent different recommendation environments, with MovieLens offering a more balanced and denser user-item matrix, and Yelp presenting challenges due to its sparsity and less consistent rating patterns.

To assess model performance, we used **Mean Absolute Error (MAE)** as the primary evaluation metric. For Item-Based Collaborative Filtering, we tested various values of k (the number of similar items considered for prediction), while for the Baseline Gradient Descent model, we evaluated performance across different numbers of training epochs. The statistical baseline model, which simply estimates ratings by combining global, user, and item biases, served as a fast and surprisingly accurate benchmark for comparison.

In this project, a **User-Based Collaborative Filtering model** was provided as a reference. Although it was not part of the final implementation, it played a critical role in helping to understand collaborative filtering structures and influenced the development of the Item-Based model.

This report presents a comprehensive overview of the methods, including data preprocessing, similarity computation, parameter tuning, model evaluation, and results interpretation. Through systematic experimentation, we observed that while Item-Based CF can produce accurate results in dense settings, the statistical and gradient-based bias models offer more robust performance, particularly in sparse environments like Yelp.

Overall, this project provided valuable experience in **building recommender systems from scratch**, working with sparse data, analyzing algorithmic behavior, and understanding the trade-offs involved in real-world recommendation tasks. It also laid the foundation for deeper exploration into more advanced techniques such as latent factor models and hybrid recommender systems.

1. Introduction

Recommender systems have become a vital component of modern digital platforms, enabling users to navigate large volumes of content by providing personalized recommendations. They are widely used on platforms such as Amazon (for product suggestions), Netflix (for movies and shows), and Yelp (for local businesses and restaurants). The core purpose of recommender systems is to reduce information overload and improve user experience by automatically predicting which items a user might like, based on their preferences or behavior.

These systems have evolved into two primary types: **Content-Based Filtering** and **Collaborative Filtering**.

- **Content-Based Filtering** recommends items that are similar to those the user has liked before, based on item attributes such as genre, tags, or descriptions.

- **Collaborative Filtering (CF)**, on the other hand, relies on patterns of user behavior — such as ratings or interactions — without needing any information about the items themselves. CF models identify users or items that are similar and make predictions based on these similarities.

This project focuses on implementing and evaluating three collaborative filtering techniques from scratch:

1. **Item-Based Collaborative Filtering** – uses cosine similarity between items to predict ratings based on similar items rated by the user.

2. **Baseline Statistical Bias Model** – predicts ratings by accounting for the average rating, user bias, and item bias.

3. **Baseline Bias Model with Gradient Descent** – learns biases using gradient-based optimization to minimize prediction error.

The goal of the project is to understand how these classic models work, compare their accuracy, and evaluate their performance across two datasets with different characteristics: **MovieLens 100K**, a clean and dense benchmark dataset, and a **Yelp-based ratings dataset**, which is sparser and more realistic. Through this project, I aim to deepen my understanding of collaborative filtering principles, learn how to handle real-world data challenges (such as sparsity), and gain practical experience with evaluation techniques like **Mean Absolute Error (MAE)**.

2. Algorithm Descriptions

Item-Based Collaborative Filtering (Item-Based CF):

Item-Based Collaborative Filtering is a type of memory-based recommendation algorithm that focuses on finding similarities between items, rather than users. The main idea is that if a user has rated certain items, we can recommend new items that are similar to the ones they already liked.

Instead of relying on user-to-user comparisons, this method computes item-to-item similarities based on the ratings provided by all users. It assumes that users tend to rate similar items in similar ways. To predict a user's rating for a specific item, the model examines how the user has rated other similar items and combines those ratings.

How It Works:

Construct a user-item matrix
A matrix is created where each row represents a user, and each column represents an item. The values are the ratings that users have given to items. Most of the matrix is usually empty (sparse), as users typically rate only a small number of items.

Normalize item ratings
To handle user rating bias, each item's ratings are mean-centered. This means we subtract the average rating of each item from all of its ratings.

Compute item-item similarity
Similarity is measured using cosine similarity, which evaluates how similar two item rating vectors are. The formula is:

$$\text{sim}(i,j) = \frac{\sum_{u \in U_{i,j}} r_{u,i} \cdot r_{u,j}}{\sqrt{\sum_{u \in U_{i,j}} r_{u,i}^2} \cdot \sqrt{\sum_{u \in U_{i,j}} r_{u,j}^2}}$$

Where $U_{i,j}$ is the set of users who have rated both items i and j.

Predict the rating
To predict the rating of user u for item i, we look at the top-k most similar items to i that user u has rated. The prediction formula is:

$$\hat{r}_{u,i} = \frac{\sum_{j \in N_k(i)} \text{sim}(i,j) \cdot r_{u,j}}{\sum_{j \in N_k(i)} |\text{sim}(i,j)|}$$

Where:

- $N_k(i)$ is the set of top-k items most similar to item i,
- $r_{u,j}$ is the rating of user u for item j.

Why It's Useful

Item-Based CF is often more stable than User-Based CF because item similarity doesn't change as much over time as user preferences might. It also performs well when the number of users is much larger than the number of items (which is common in real-world platforms).

This method is easy to understand and works effectively when there are sufficient overlapping ratings between items. However, it may struggle with very sparse datasets, where users have rated only a few items.

Baseline Model (Statistical Bias Computation):

The Baseline Model is a simple yet effective recommendation algorithm that improves rating predictions by accounting for **biases** in the data. Specifically, it adjusts for the fact that some users tend to rate items higher or lower than others, and that some items receive consistently higher or lower ratings regardless of the user.

Rather than comparing users or items based on similarity, this model predicts ratings by **adding corrections** to the global average rating. These corrections come from the individual biases of users and items.

Prediction Formula

The predicted rating $\hat{r}_{ui}$ for user u and item i is calculated using:

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

μ is the global average rating across all users and items.

b_u is the user bias – how much higher or lower a user usually rates compared to the average.

b_i is the item bias – how much higher or lower an item is rated compared to the average.

This approach is simple but effective because it captures general tendencies without relying on similarity computations.

How We Compute Biases:

User bias is calculated as:

$$b_u = \frac{1}{|I_u|} \sum_{i \in I_u} (r_{u,i} - \mu)$$

Where I_u is the set of items rated by user u.

Item bias is calculated as:

$$b_i = \frac{1}{|U_i|} \sum_{u \in U_i} (r_{u,i} - \mu)$$

Where U_i is the set of users who rated item i.

These biases are computed using only the training data. If a user or item has no ratings, their bias is treated as zero, and the prediction defaults to the global average.

Why This Model Works

The Baseline model is often used as a starting point or reference model because it is:

- Fast: It doesn't require computing similarity or large matrix operations.
- Interpretable: It clearly shows how each component (user, item, global average) contributes to the prediction.
- Surprisingly accurate: It often outperforms more complex models on sparse datasets where deeper relationships are hard to learn.

Although it doesn't capture interactions between users and items, it forms a strong foundation and is often included as a component in hybrid and advanced models.

Baseline Model (Learned with Gradient Descent):

The Gradient Descent version of the Baseline model builds on the statistical bias approach by utilizing an optimization algorithm to learn the user and item biases, rather than calculating them directly. This enables the model to adapt more effectively to the data, particularly when the dataset is large or noisy.

Rather than just computing the average differences (biases), we treat bias learning as a minimization problem. The goal is to reduce the difference between actual and predicted ratings by adjusting the biases through multiple iterations using gradient descent.

Prediction Formula

Just like the statistical baseline model, the predicted rating is:

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

μ is the global average rating across all users and items.

b_u is the user bias – how much higher or lower a user usually rates compared to the average.

b_i is the item bias – how much higher or lower an item is rated compared to the average.

But here, instead of pre-calculating b_u and, b_iWe learn them by minimizing the loss function.

Objective (Loss Function)

We minimize the regularized squared error over all known ratings:

$$J = \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - b_u - b_i)^2 + \lambda \left(\sum_u b_u^2 + \sum_i b_i^2 \right)$$

b_u and b_i are variables that need to be learned from the training data.

Gradient Descent Updates

In each training iteration (called an epoch), we:

$$e_{ui} = r_{ui} - \mu - b_u - b_i$$

Update the user and item biases:

$$b_u \leftarrow b_u + \alpha(e_{u,i} - \lambda b_u)$$

$$b_i \leftarrow b_i + \alpha(e_{u,i} - \lambda b_i)$$

Where α is the **learning rate** (controls how fast we learn).

Why Use Gradient Descent?

- It learns more effectively with additional data and adjusts to subtle patterns in ratings.
- It gives more flexibility than computing averages.
- It helps minimize the overall prediction error step by step.

However, it must be tuned carefully. If you train for too many epochs or use a learning rate that is too high, the model can **overfit**, especially on sparse datasets like Yelp.

3. Experimental Study

This section describes the preparation and processing of the datasets, the structure of the experiments, the tools and technologies used, and the evaluation strategy employed for model comparison. It also explains the performance measurement process and the reasoning behind each choice.

Preprocessing

MovieLens 100K:

The MovieLens 100K dataset was straightforward to work with and required minimal preprocessing. It already included a clean structure and predefined splits for training and testing. Specifically, we used the files u1.base for training and u1.test for testing. Each row in the dataset contains four fields: User ID, Movie ID, Rating, and Timestamp. Since the data is well-maintained and commonly used in academic research, there was no need for any cleaning or filtering. We preserved the original format but converted the user and item IDs to 0-based indexing for compatibility with Python's array-based data structures. This dataset was ideal for testing the models in a controlled and balanced environment, as it features a dense rating matrix where most users have rated a variety of items. The reliable structure and moderate size of the dataset made it an excellent starting point for developing and evaluating collaborative filtering techniques and baseline models.

Yelp Dataset:

The Yelp dataset required more effort in preprocessing due to its unstructured and sparse nature. We began with a CSV file (ratings.csv) that included user reviews of businesses. From the original dataset, we selected only the columns relevant to our recommendation task: User ID, Business ID, Rating, and Date. Any extra columns, such as review text or business metadata, were dropped to focus strictly on numeric rating prediction. This step simplified the dataset and improved efficiency during matrix operations. Unlike MovieLens, the Yelp dataset did not come with predefined training and testing splits. Therefore, we implemented a time-aware splitting strategy. For each user, we sorted their ratings in chronological order using the Date column. We then allocated the first 80% of each user's ratings to the training set and the remaining 20% to the test set. This simulates a realistic recommendation scenario where models predict future user preferences based on past behavior. We did not apply missing value imputation; instead, our models were designed to make predictions only when sufficient supporting data was available, for instance, when a user had rated similar items or when item neighbors existed. This conservative approach avoids introducing artificial information into the system, ensuring that all predictions are based on actual user interaction patterns.

Programming Tools:

I used Python for the entire project because it's flexible and has great libraries for data and math. Python Libraries used:

- `Pandas` were used to load, clean, and manipulate the datasets. It provided an intuitive interface for sorting, filtering, and grouping ratings by user or item, especially when preparing training and test splits for the Yelp dataset.

- `NumPy` enables fast numerical computations using vectors and matrices. It helped perform mean calculations, apply update rules, and handle matrix math in a vectorized, efficient way.

- `SciPy.sparse` was critical for working with large user-item matrices. Since most users only rate a small fraction of items, representing the full matrix as a dense structure would be highly inefficient. Instead, I used sparse matrix formats (CSR and CSC) to store only non-zero entries, saving memory and accelerating computation.

- `Matplotlib` was used to visualize results, such as plotting MAE vs. the number of neighbors or epochs. These visualizations effectively illustrated trends in model performance, making it easier to interpret the effectiveness of different algorithms.

To ensure fast and accurate ID mapping, especially for the Yelp dataset where User IDs and Business IDs are alphanumeric strings, I used dictionaries to map them to continuous integer indices. This step was essential for matrix indexing and sparse matrix construction.

Additionally, I standardized the evaluation pipeline across all three algorithms and both datasets. Each model followed the same sequence: train on the training set, predict on the test set, and compute MAE for comparison. This consistent methodology ensured a fair and reliable comparison between models, allowing for meaningful observations about how performance varied with parameters such as neighborhood size or the number of epochs.

Item-Based Collaborative Filtering

Procedure:
For the Item-Based Collaborative Filtering model, the first step involved building a user-item rating matrix from the training data. This matrix represents users as rows and items (such as movies or businesses) as columns, where each entry indicates the rating a user gave to an item. Since most users rate only a small number of items, the matrix is mostly empty. To store this matrix efficiently, I used a sparse matrix format (CSR) from the scipy. sparse library, which helps save memory and speeds up matrix operations.

Next, I applied item normalization, which means adjusting each item's ratings by removing its average rating. This step is important because some items may naturally receive higher or lower ratings, which can distort similarity calculations. By subtracting the mean rating of each item, we ensure that similarities reflect rating patterns, not absolute values.

After normalizing the matrix, I computed cosine similarity between items. Cosine similarity measures the similarity between two items based on how users have rated them. For example, if two movies tend to be rated similarly by the same users, they will have a high cosine similarity. The result is an item-item similarity matrix, where each entry indicates the degree of relatedness between two items based on user behavior.

To make predictions for a specific user-item pair (i.e., to predict how much a user would rate an item they haven't rated yet), the model finds the top-k most similar items to the target item that the user has already rated. Then, it computes a weighted average of the user's ratings on those similar items, using the similarity score as weights. This approach assumes that users will rate an item similarly to how they rated other items that are closely related to it.

This entire process was repeated for multiple values of k, the number of neighbors (similar items) used for prediction. The model's accuracy was then evaluated using Mean Absolute Error (MAE) for each k**.**

Evaluation Strategy

To evaluate the performance of the Item-Based Collaborative Filtering model, I designed a controlled testing procedure that allowed me to observe how prediction accuracy varied with different parameter settings. The core metric I used to measure performance was Mean Absolute Error (MAE), which calculates the average absolute difference between predicted ratings and the actual ratings in the test set. MAE is a widely used metric in recommender system evaluation because it is easy to interpret—lower MAE values indicate more accurate predictions.

The key parameter in this model is k, the number of most similar items (neighbors) used to make a prediction. I evaluated the model using the following values of k:

$$k = \{1,5,10,50,100\}$$

This parameter study helped analyze how the neighborhood size affects prediction quality. Using a small k means only the most similar items influence the prediction, which may lead to more personalized but potentially less stable estimates. A larger k, on the other hand, includes more items and may yield smoother, more general predictions, but could also dilute the impact of highly similar neighbors.

For each test case (user-item pair) in the test set:

I identified the most similar k items to the target item that the user had already rated.

I used the cosine similarity scores as weights to compute a weighted average of the user's ratings on those similar items.

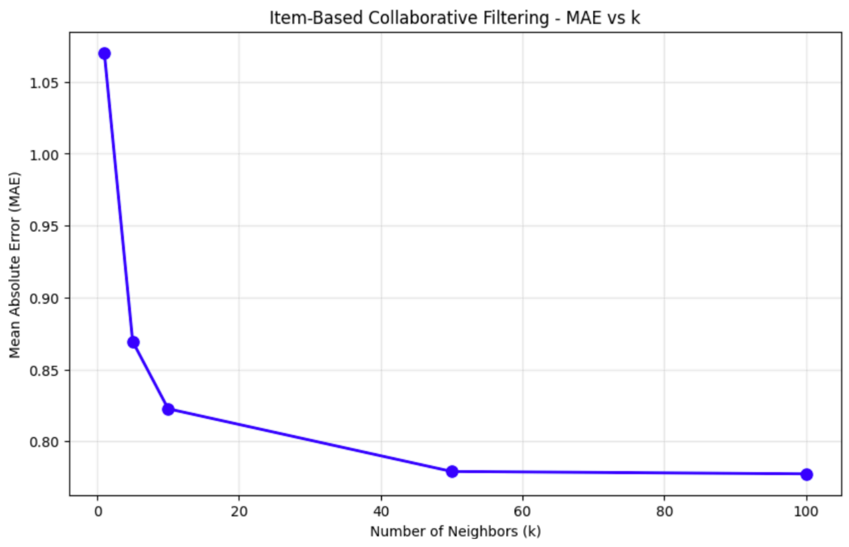
If the user had not rated any of the top-k similar items, no prediction was made for that pair.

After computing predictions for all valid test pairs, I calculated the MAE by comparing predicted ratings with the true ratings. This process was repeated separately for each value of k, allowing me to track how the model's accuracy changed with different neighborhood sizes.

Finally, the MAE values for all k values were plotted to visualize performance trends. This helped identify the optimal neighborhood size and provided insights into how model complexity impacts accuracy, particularly across different datasets (MovieLens vs. Yelp).

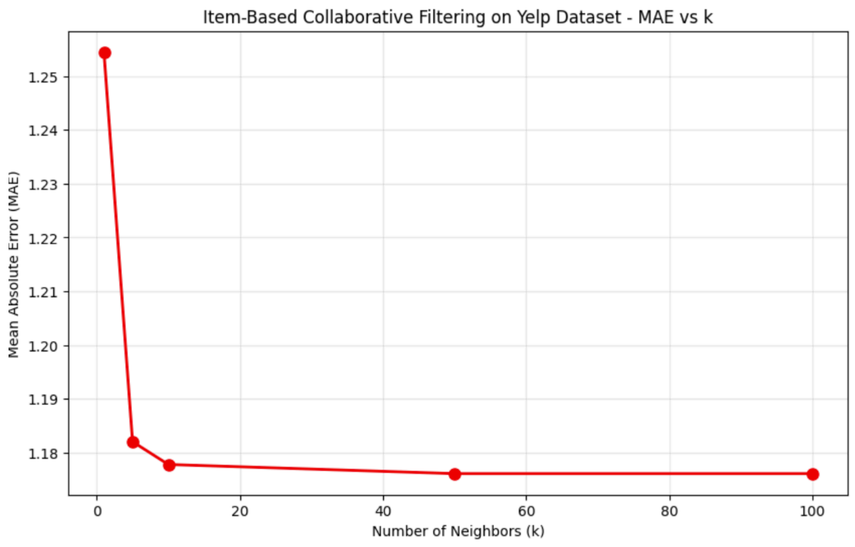
Results:

The Item-Based CF model worked well, especially when using more neighbors (k ≥ 10). On the MovieLens dataset, where user-item ratings are denser, it achieved better accuracy. As k increased, the MAE decreased or remained stable, indicating that having more similar items helped improve predictions. On the sparser Yelp dataset, the model didn't perform as well, but still followed the same general trend.



Summary of Results for MovieLens Dataset:

K	MAE
1	1.0702
5	0.8694
10	0.8227
50	0.7791
100	0.7775



Summary of Results for Yelp Dataset:

k	MAE
1	1.2544
5	1.1820
10	1.1778
50	1.1761
100	1.1761

Baseline Model (Statistical)

Procedure:

To apply the **Baseline Statistical Model** in this project, I began by building a user-item rating matrix from the training dataset using a sparse matrix format. This matrix was used to compute three key components that form the basis of the model's prediction formula:

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

μ is the global mean rating (across all ratings in the training set).

b_u is the user's average deviation from the global mean.

b_i is the item's average deviation from the global mean.

The process started by calculating μ as the mean of all ratings in the training matrix. Then, for each user u, I computed the **user bias** b_u by:

- Extracting all ratings that the user had given,
- Subtracting μ from each of those ratings,
- Taking the average of these differences.

Similarly, for each item i, I computed the **item bias** b_i by:

- Extracting all ratings the item had received,
- Subtracting μ,
- Taking the mean of those differences.

Once μ, b_u, and b_i were calculated, the prediction for any test user-item pair was made by simply adding these three values. This model was implemented identically for both the MovieLens and Yelp datasets. The only difference was the input matrix: MovieLens used u1.base and u1.test files, while the Yelp dataset required a custom 80/20 chronological split.

One of the strengths of this model is that it requires no training loop or optimization—it simply analyzes existing ratings and derives useful patterns in user and item behavior. This made the model computationally efficient and practical.

Evaluation

To evaluate the model's prediction accuracy, I used the Mean Absolute Error (MAE) metric. MAE is calculated by averaging the absolute difference between predicted ratings and actual ratings in the test set:

Where:

$$\text{MAE} = \frac{1}{n} \sum_{(u,i)} |r_{u,i} - \hat{r}_{u,i}|$$

- $r_{u,i}$ is the true rating,
- $\hat{r}_{u,i}$ is the predicted rating,
- n is the number of user-item pairs evaluated.

The evaluation process included the following steps:

- For each user-item pair in the test set, I used the formula $\mu + b_i + b_u$ to compute a predicted rating.
- If either the user or the item was not present in the training set (a cold-start scenario), the model skipped that prediction.
- For all valid predictions, the absolute error was calculated.
- MAE was computed by averaging these errors.

μ is the global average rating across all users and items.

b_u is the user bias – how much higher or lower a user usually rates compared to the average.

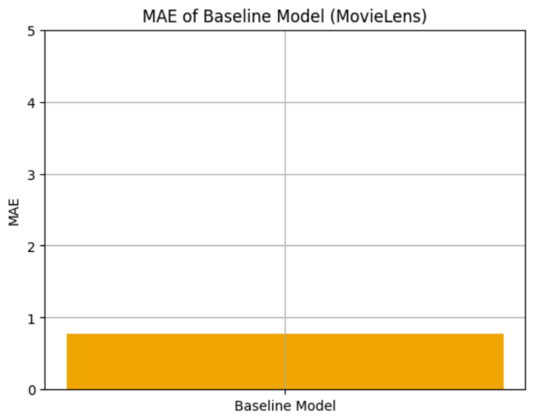
b_i is the item bias – how much higher or lower an item is rated compared to the average.
But here, instead of pre-calculating b_u and b_i, we learn them by minimizing loss function.

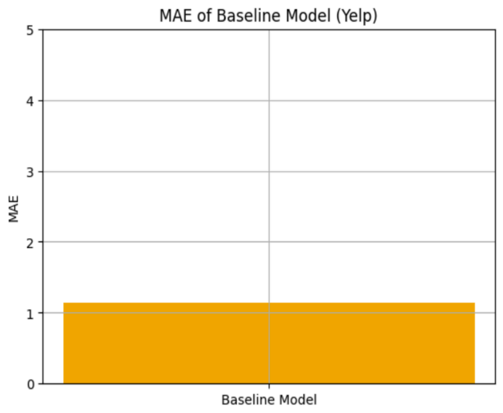
Results:

- On **MovieLens**, the model achieved an MAE of **0.7711**, indicating that the bias correction effectively captured important user and item patterns.

- On **Yelp**, the MAE was **1.144**, which, while higher, still showed decent performance given the increased sparsity of the data.

  Overall, the baseline model was a good starting point. It was easy to use, fast, and accurate enough to compare with more complex models, such as collaborative filtering or gradient descent.





MovieLens Dataset
MAE = 0.7710940008258919

Yelp Dataset
MAE = 1.1437658136030375

Baseline Model (with Gradient Descent)

The Baseline Gradient Descent Model builds on the idea of bias-based prediction, where a user's rating is estimated using:

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

However, unlike the standard baseline model, where b_u and b_i are computed directly as averages, this model learns the biases through iterative optimization using gradient descent.

For this model, I used an iterative training loop to update the user and item biases based on the prediction error for each rating in the training set. Initially, all biases were set to zero. The model then passed through the training data multiple times (in epochs), and during each pass, it adjusted the biases using feedback from the observed errors. This update process was controlled by two important parameters:

- A learning rate α, which determines the step size for updates,
- A regularization term λ, which prevents the biases from growing too large and overfitting to the training data.

  The updated rules applied during training were:

$$b_u \leftarrow b_u + \alpha(e_{u,i} - \lambda b_u)$$
$$b_i \leftarrow b_i + \alpha(e_{u,i} - \lambda b_i)$$

This approach allowed the model to gradually learn how much each user and item typically deviated from the global mean μ by minimizing the squared error of predictions.

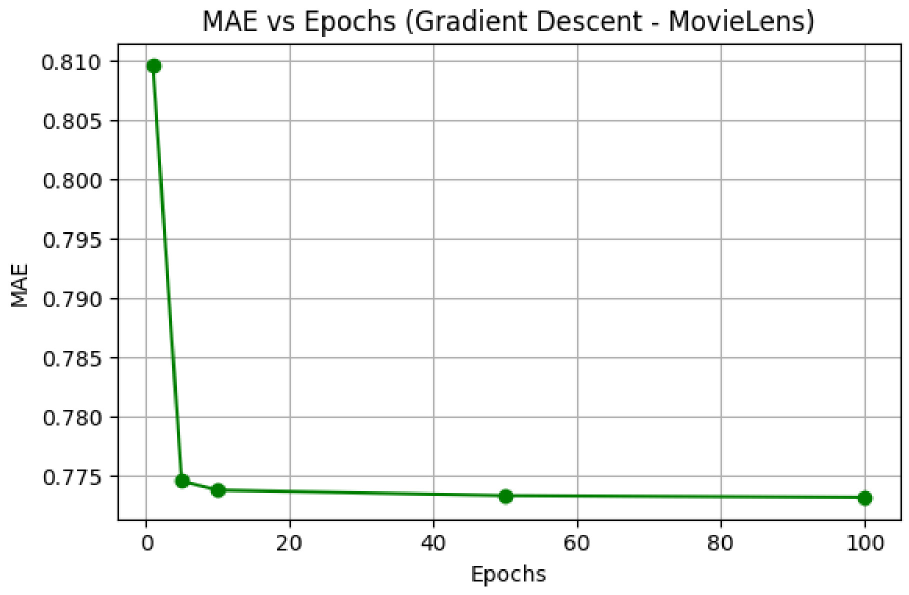
I tested the model's performance over multiple epochs, specifically 1, 5, 10, 50, and 100, and measured the MAE after each run. The goal was to observe how the model improved over time and when it began to overfit, particularly on the sparser Yelp dataset. I also visualized these results using a plot of MAE versus number of epochs to clearly observe the learning behavior over time.

Evaluation

To evaluate the Gradient Descent Baseline model, I measured prediction accuracy using Mean Absolute Error (MAE) after training for different numbers of epochs: 1, 5, 10, 50, and 100. For each epoch count, I used the final learned biases b_u and b_i to predict ratings on the test set and compared them to actual ratings. On the MovieLens dataset, MAE consistently improved with more training epochs, dropping from 0.8096 (1 epoch) to 0.7732 (100 epochs), indicating stable learning. However, on the Yelp dataset, MAE decreased initially (from 1.0806 to 1.0215 by epoch 10) but worsened at higher epochs (reaching 1.1025 at epoch 100). This suggests the model overfit the sparse Yelp training data, highlighting the need for early stopping or better regularization. I also visualized this trend using a plot of MAE versus number of epochs, which confirmed the early improvement and later decline on Yelp. This evaluation helped demonstrate both the potential and limitations of gradient-based learning in recommender systems.

Results:

The gradient descent model showed good improvement in the first few epochs. On both datasets, MAE decreased quickly early on. However, training for too many epochs, especially on Yelp, led to overfitting. This model was more flexible but needed careful tuning to avoid poor results.



MovieLens

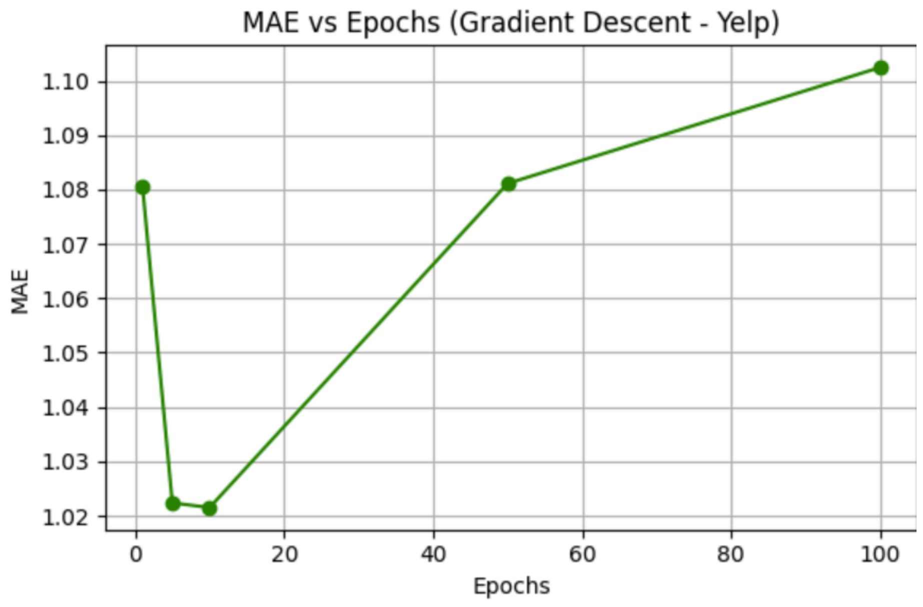
Epochs = 1, MAE = 0.8096

Epochs = 5, MAE = 0.7746

Epochs = 10, MAE = 0.7738

Epochs = 50, MAE = 0.7734

Epochs = 100, MAE = 0.7732



Yelp

Epochs = 1, MAE = 1.0806

Epochs = 5, MAE = 1.0224

Epochs = 10, MAE = 1.0215

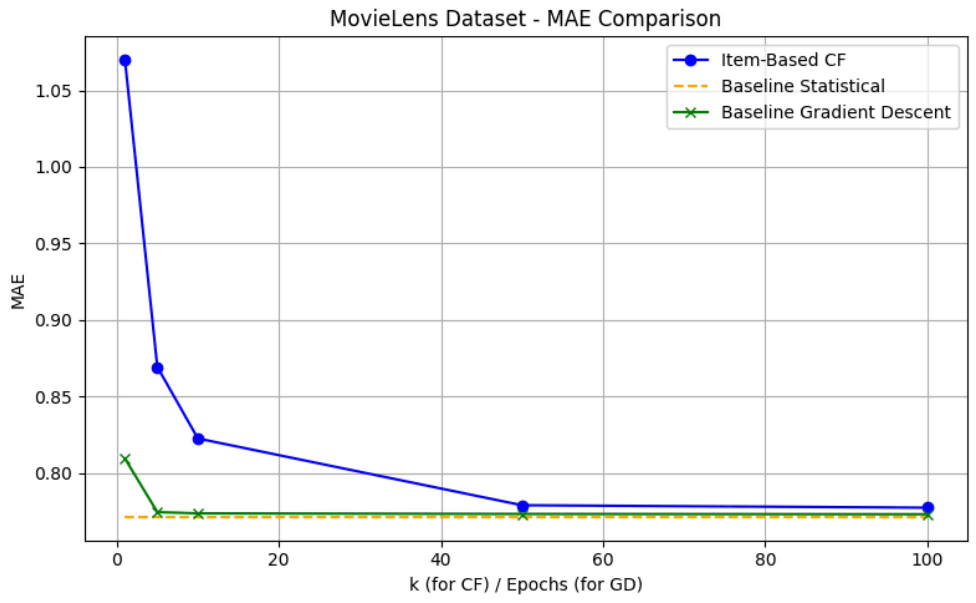
Epochs = 50, MAE = 1.0811

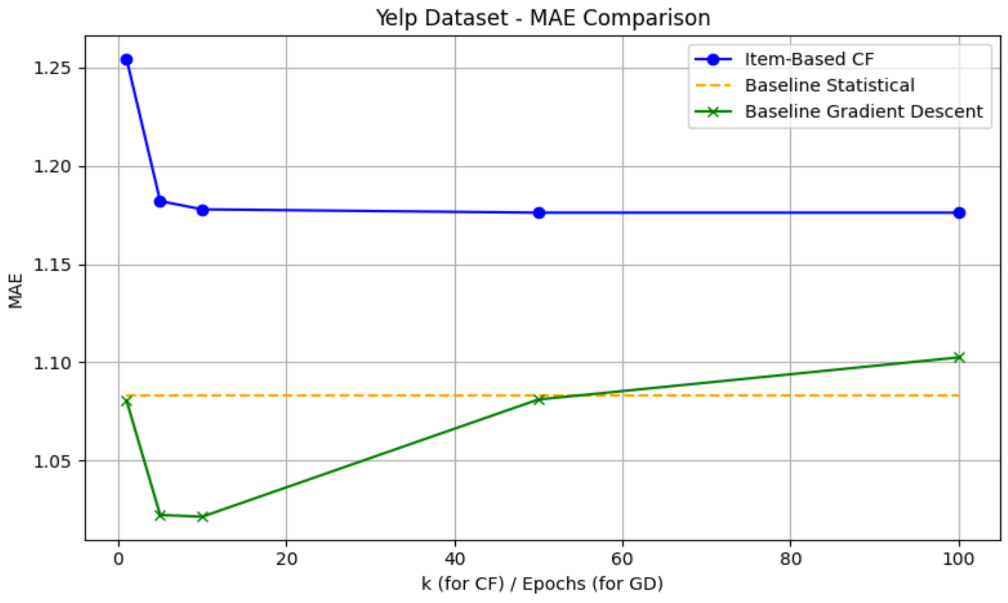
Epochs = 100, MAE = 1.1025

Model Comparison

When comparing the three models, we can see that the Baseline Statistical model gave the most consistent and accurate results on both datasets. In the MovieLens graph, it had the lowest MAE (around 0.77) and stayed stable. The Item-Based CF model improved as we increased the number of neighbors (k), starting with high error and getting better, but it still didn't beat the baseline. The Gradient Descent model also improved quickly in the first few training rounds (epochs), but then it leveled off near the baseline.

On the Yelp dataset, things were trickier because the data is sparser. The Item-Based CF didn't improve much, even with a higher k, and the Gradient Descent model performed well at first but actually deteriorated when trained for too long—this indicates that it started overfitting. In the end, the Baseline Statistical model was the most reliable and balanced across both datasets.





Observations and Results

After testing and comparing all three models across both datasets, several important patterns and insights emerged. The Item-Based Collaborative Filtering, Statistical Baseline Model, and Baseline Gradient Descent Model each had unique strengths and limitations that were evident through performance trends, parameter sensitivity, and behavior across dense vs. sparse datasets.

The Statistical Baseline Model consistently delivered strong results on both the MovieLens and Yelp datasets. On MovieLens, it achieved the lowest MAE (0.7711) among all models. Its simplicity, fast computation, and ability to capture user and item rating tendencies made it highly effective, especially in dense datasets where sufficient historical ratings exist for each user and item. Surprisingly, this model also performed better than Item-Based CF on the Yelp dataset, which suggests that modeling biases is often more reliable than relying on similarity measures when data is limited or sparse.

The Item-Based CF model performed reasonably well on the MovieLens dataset, especially when using a larger number of neighbors ($k \geq 10$). As the value of k increased, the MAE decreased, indicating that more context (i.e., neighboring items) improves predictions when data is available. However, this model underperformed on the Yelp dataset, with the best MAE still above 1.17. The likely cause is data sparsity—most Yelp users rated only a few businesses, and a small number of users rated the majority of items. As a result, it became difficult to find enough meaningful neighbors for a given item, making predictions less accurate.

The Gradient Descent Baseline Model showed the most dynamic behavior. On MovieLens, the model improved steadily with more epochs and ultimately approached the performance of the statistical baseline. This suggests that the gradient descent process was able to gradually refine the user and item biases with sufficient data. However, on Yelp, the model initially showed early improvement, followed by overfitting—after epoch 10, the MAE started to rise. This indicates that in sparse datasets, the model begins to memorize noise rather than generalize, which results in poorer predictions on unseen data.

From these patterns, a few key takeaways and areas for improvement emerged:

Yelp's sparsity is the primary challenge. Sparse data weakens collaborative filtering because it relies on finding common ratings between users or items. To address this, one strategy could be using matrix factorization (e.g., SVD or NMF) to capture latent factors, which can infer structure even from limited data. Alternatively, incorporating content-based features (e.g., business categories or review text) could help enrich the sparse matrix.

Regularization and early stopping are crucial tools for preventing overfitting in models like Gradient Descent, particularly on datasets such as Yelp. More advanced optimization techniques might also improve convergence and stability.

Although the Statistical Baseline Model performed well, its simplicity limits its flexibility. Future work could extend this approach to more expressive models, such as hybrid models that blend bias-based and similarity-based approaches.

Overall, the results confirm that the choice of model and parameters must align with the characteristics of the dataset. Dense, clean data enables more complex models to excel, while sparse data often benefits from simpler, yet more robust, approaches.

4. Conclusion

This project focused on designing and evaluating three recommendation models: **Item-Based Collaborative Filtering**, a **Statistical Baseline Model**, and a **Baseline Model trained with Gradient Descent**. These models were implemented from scratch in Python and tested on two datasets: **MovieLens 100K** and a custom-filtered subset of the **Yelp dataset**. The primary evaluation metric used was **Mean Absolute Error (MAE)**, and the performance of each model was studied under different parameter settings to understand their strengths and limitations.

To begin, I reviewed a sample implementation of **User-Based Collaborative Filtering**, which helped me understand the structure and flow of collaborative filtering algorithms. Although this sample was not part of my submitted work, it played a useful role in guiding the structure of the **Item-Based CF model**. This model relied on calculating cosine similarity between items and making predictions based on the top-k most similar items rated by a user. Through experimentation, I observed that the model's accuracy improved as the value of k increased, especially on the MovieLens dataset, where user behavior was denser. However, the performance was weaker on the Yelp dataset due to its sparsity.

The **Statistical Baseline Model** provided an efficient and surprisingly strong benchmark. It required no training loop and used a simple formula to incorporate user and item rating tendencies. Despite its simplicity, it achieved one of the lowest MAE scores on the MovieLens dataset. This outcome highlighted the effectiveness of bias-based models, particularly when the data is well-populated.

In contrast, the **Baseline Gradient Descent Model** introduced iterative learning to improve the bias estimates. I implemented a training loop to adjust the user and item biases using error feedback, learning rate, and regularization. This model enabled me to investigate how training duration (in epochs) impacts performance. On the MovieLens dataset, performance improved consistently over a greater number of epochs. On the Yelp dataset, however, MAE initially decreased but worsened after too many epochs, indicating **overfitting**. This taught me the importance of tuning hyperparameters and monitoring performance during training, especially with sparse data.

Throughout the project, I practiced full-cycle machine learning development—from **data preprocessing** to **model evaluation and visualization**. I wrote all models from scratch using core libraries such as `NumPy`, `Pandas`, and `SciPy`, and utilized `Matplotlib` for visualization. I also learned the importance of sparse matrix representations for improving memory efficiency and how to organize rating data for large-scale evaluation effectively.

Overall, this project deepened my understanding of recommender systems both theoretically and practically. It helped me learn how to analyze real-world datasets, build recommender models, and evaluate them fairly. I gained confidence in working with sparse data, debugging matrix-based implementations, and designing experiments to test different ideas. The hands-on experience has motivated me to continue learning about machine learning and recommendation technologies, and I now feel much more prepared to explore more advanced models such as latent factor methods or deep learning-based recommenders in future work.

5. Contribution Statement

Throughout this project, I took full responsibility for multiple key tasks, including reading and preparing the datasets to ensure clean and usable data, designing and implementing the recommendation algorithms, tuning model parameters for better accuracy, generating visualizations to clearly present the results, and writing this comprehensive report. Although the user-based collaborative filtering code was provided as a starting example, I independently developed the item-based collaborative filtering model as well as the two baseline models from scratch. To strengthen my understanding and ensure correctness, I carefully studied the course lecture slides and relevant formula explanations, particularly focusing on concepts such as cosine similarity, bias modeling, and gradient descent. This approach allowed me to align my work with the academic expectations and theoretical foundations of the course. Overall, this hands-on experience significantly deepened my understanding of recommender systems and enhanced both my programming abilities and problem-solving skills.