Probabilistic Matrix Factorization (PMF)

Introduction to Recommender Systems

Automated recommender systems have become an integral part of many platforms, offering users product or content suggestions tailored to their preferences. These systems fall into two major categories:

1. **Content-Based Filtering**:
    - Uses user-specific data (explicit preferences) to recommend items.
    - Example: Recommending books based on genres a user prefers.
2. **Collaborative Filtering**:
    - Relies on data from other users with similar preferences to fill in missing information.
    - Example: Suggesting movies based on ratings provided by users with similar tastes.

Why Collaborative Filtering?

Collaborative filtering shines when user-specific data is incomplete.

Probabilistic Matrix Factorization (PMF) is a popular collaborative filtering technique designed to handle this scenario efficiently. This tutorial focuses on understanding PMF and its workings.

What is Probabilistic Matrix Factorization?

Probabilistic Matrix Factorization (PMF) is a technique used primarily for **collaborative filtering** in recommendation systems. It aims to predict missing entries in a **user-item interaction matrix** (e.g., ratings or preferences), which is sparse because users typically interact with only a small subset of items.

PMF operates on a user-item ratings matrix R, where:

- Rows represent **users**.
- Columns represent **items**.
- Each cell contains a **rating**, which can be empty (sparse matrix).

PMF aims to:

- Predict missing ratings in R.
- Recommend items to users based on estimated ratings.

To achieve this, PMF factorizes *R* into two low-rank matrices:

1. *U* : User features matrix (*N* * *D*).
2. *V* : Item features matrix (*D***M*).

The approximation is expressed as:

$$R = U^T V.$$

Here, *D* is the latent feature dimension.

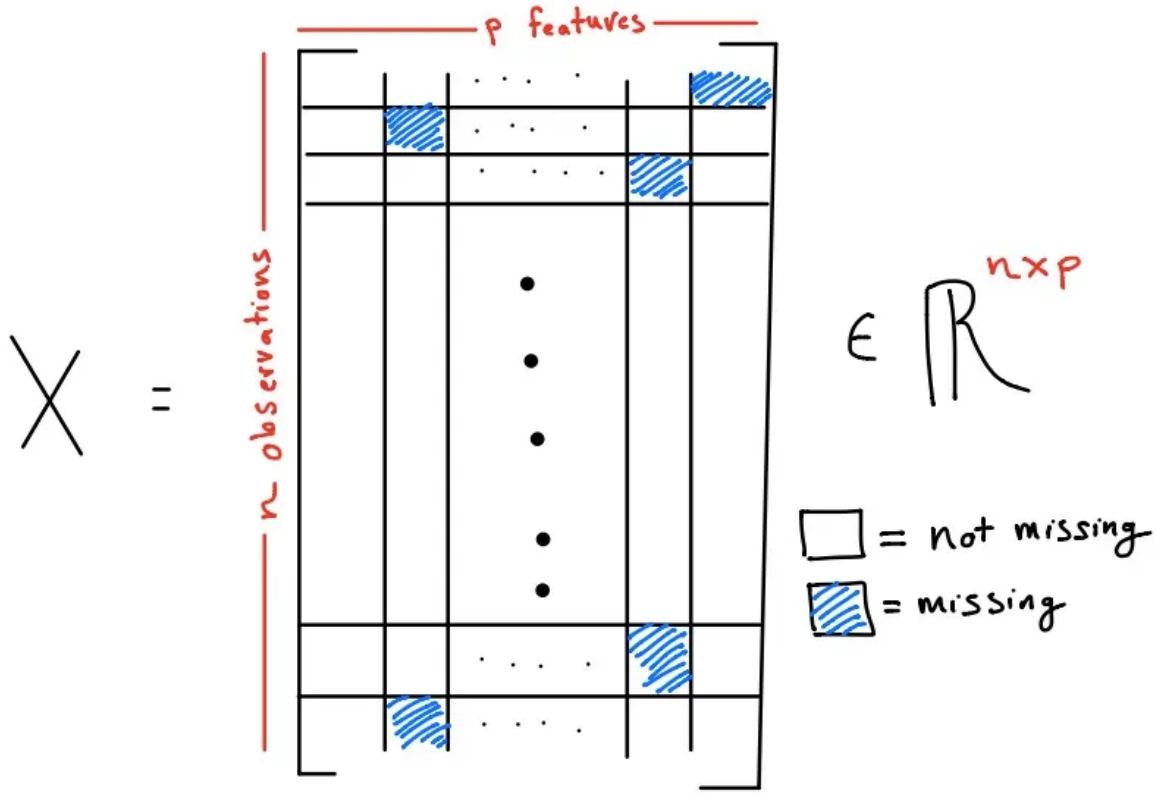


Figure 1The diagram depicts a matrix with missing values.

```
In [8]: def get_ratings_matrix(df, train_size=0.75):
            user_to_row = {}
            movie_to_column = {}
            df_values = df.values
            n_dims = 10
            parameters = {}

            uniq_users = np.unique(df_values[:, 3])
            uniq_movies = np.unique(df_values[:, 0])

            for i, user_id in enumerate(uniq_users):
                user_to_row[user_id] = i

            for j, movie_id in enumerate(uniq_movies):
                movie_to_column[movie_id] = j

            n_users = len(uniq_users)
            n_movies = len(uniq_movies)

            R = np.zeros((n_users, n_movies))

            df_copy = df.copy()
            train_set = df_copy.sample(frac=train_size, random_state=0)
            test_set = df_copy.drop(train_set.index)

            for index, row in train_set.iterrows():
                i = user_to_row[row.userId]
                j = movie_to_column[row.movieId]
                R[i, j] = row.rating

            return R, train_set, test_set, n_dims, n_users, n_movies, user_to_row, movie_to_column
```

Figure 2 A screenshot of a code to get ratings

How PMF Works

1. **Matrix Factorization**:
    - PMF uses Bayesian principles to estimate *U* and *V*.
    - The objective is to minimize the difference between observed ratings in *R* and their approximations in $U^T V$.
2. **Bayesian Inference**:
    - PMF applies **Bayes' rule** to update its parameters iteratively:

$$p(\theta|\mathbf{X},\alpha) = \frac{p(\mathbf{X}|\theta,\alpha)p(\theta|\alpha)}{p(\mathbf{X}|\alpha)} \propto p(\mathbf{X}|\theta,\alpha)p(\theta|\alpha)$$

Equation 1 Equation for Bayes' rule

Where:

- *X*: Data (ratings in *R*).
- θ : Model parameters (*U,V*).
- α : Hyperparameters.

    - Posterior distributions of parameters *U,V* are computed using likelihood and prior information.

Why use Bayesian inference?

- Bayesian inference integrates both the observed data (R) and prior knowledge (U, V) into a unified probabilistic framework.
- This ensures that the model balances the observed data's contribution and the regularization imposed by the priors.

3. **Training Process**:
   - R : Observed data.
   - U and V : Parameters to estimate.
   - Optimization : The goal is to maximize the posterior distribution by minimizing the negative log-posterior
   - Gradient descent updates iteratively to achieve this.

4. **Workflow**

Input Data: Start with a sparse user-item ratings matrix , where most entries are missing.

Initialization: Randomly initialize and matrices. Ensure follows a Gaussian distribution with mean 0 and variance .

Compute Posterior: Use Bayes' Rule to combine the likelihood function (Gaussian for observed ratings) with priors (regularization terms).

Gradient Descent: Update and iteratively using gradients of the log-posterior. Repeat for a fixed number of epochs or until convergence.

Prediction: Use to predict missing ratings. Scale predictions to match the observed rating range.

Evaluation: Calculate Root Mean Square Error (RMSE) on the test set to measure performance.

Output: Generate personalized recommendations based on predicted ratings.

Learning through example (Analogy)

The Problem Topic : Recommending Items

Imagine a library trying to recommend books to its readers. The librarian has a big grid (user-item matrix) where rows represent readers and columns represent books. The numbers in the grid are ratings from readers about the books they've read. But the problem is, most of the grid is blank because people have only rated a few books.
The librarian’s goal is to guess the missing ratings to recommend books people might enjoy.

Breaking the Problem into Pieces (Matrix Factorization)

Instead of directly filling in the blanks, the librarian has a clever idea:
"What if I can describe each reader by their preferences (e.g., how much they like fantasy, thrillers, or history) and describe each book by its traits (e.g., how thrilling or historical it is)? Then, I can match readers and books by looking at how well their preferences align."

This idea is matrix factorization. The librarian breaks the big user-item matrix into two smaller sets of information:

1. **Reader features** (e.g., how much a reader likes certain traits).
2. **Book features** (e.g., how much a book has those traits).

Taking a Probabilistic Approach

The librarian isn't sure about every reader's preferences or every book's traits, so they decide to be cautious and use probabilities to handle the uncertainty. For example:

- If a reader seems to like thrillers a lot, but the data is sparse, they assume there's some chance they might like fantasy too.
- Similarly, if a book has some thrilling elements, it might appeal to readers who like thrillers—but there's a chance it could interest others too.

This probabilistic approach allows for flexibility and handles gaps in the data better.

Gaussian Likelihood: Trusting Data with Some Uncertainty
Next, the librarian makes another assumption:
"Ratings usually follow a normal pattern—most ratings are close to what the book deserves, but there's room for error."

For instance, if a thriller is generally well-liked, most ratings might cluster around 4/5, but occasionally someone might give it a 3 or 5. This assumption is modelled using *Gaussian likelihood* (a bell curve for predicting ratings).

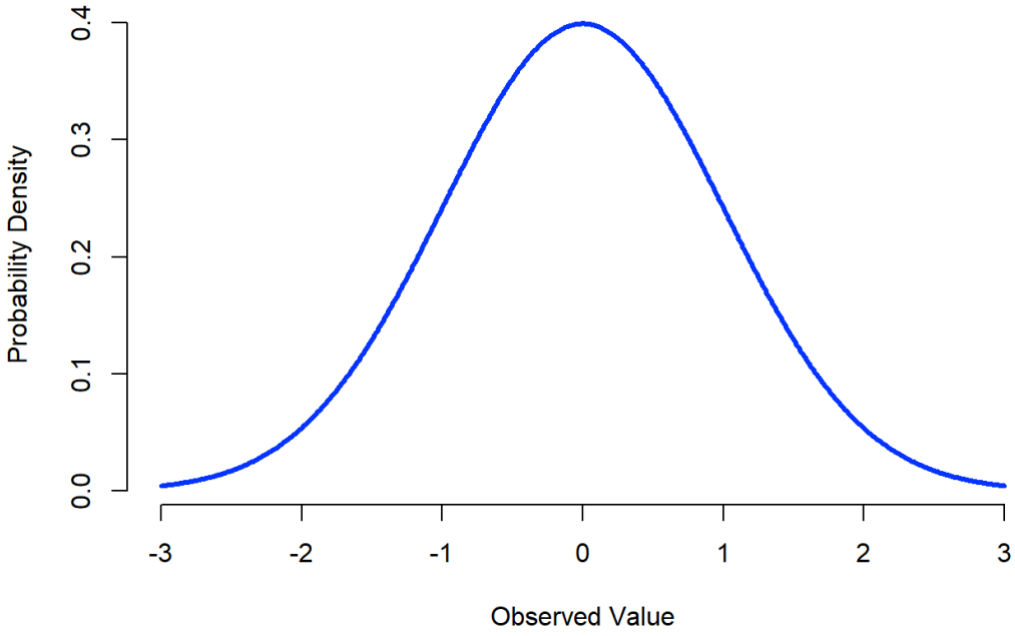


Figure 3 An image example of bell curve

Regularization: Avoiding Overconfidence

While assigning traits to books and preferences to readers, the librarian realizes a danger: "If I overfit my predictions to the existing ratings, I might make bad guesses for new readers or books."
For example, if one reader gives very extreme ratings, the librarian might wrongly conclude they only like one type of book. To prevent this, they add regularization, which acts like a rule to keep guesses more balanced and less extreme.

Prior and Posterior: Updating Beliefs

The librarian starts with some basic assumptions:

- Readers generally like an average number of traits (prior belief).
- Books generally fit into a few broad categories (prior belief).

But as new ratings are observed, the librarian updates their knowledge:

- "This reader seems to prefer thrillers over history books."
- "This book appeals more to readers who like fantasy."

This process of combining prior beliefs with observed data to refine predictions is called *Bayesian inference.*

Optimization: Finding the Best Guess

Now the librarian has a plan. They need to fine-tune the reader and book features to minimize the difference between predicted ratings and actual ratings.

This process is like adjusting ingredients in a recipe to get the perfect cake. They use a method called *gradient descent*, which takes small steps to gradually improve predictions.

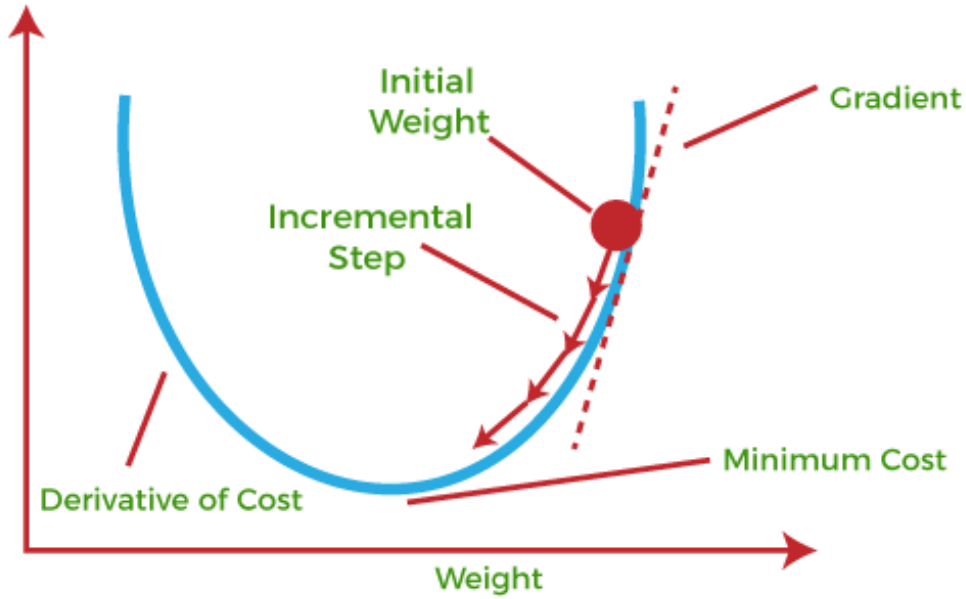


Figure 4 A visual diagram of gradient descent works

Latent Features: The Hidden Magic

As the librarian works, they discover something amazing:
The features they're uncovering aren't obvious traits like "thriller" or "fantasy." Instead, they're **latent features**—hidden patterns in the data that no one directly labelled but still explain why someone likes or dislikes a book. For instance:

- A latent feature might capture how "complex" or "simple" a book's plot is.
- Another might represent how "emotional" or "analytical" a book feels.

These hidden patterns help make better predictions.

Result: Personalized Recommendations

Finally, the librarian combines all this work into a system. For any reader, they can predict how much they'll like a book they haven't rated yet by comparing the reader's preferences to the book's traits.

Key Concepts

Likelihood Function:

- Captures how well the predicted ratings match observed ratings:
- Assumes Gaussian-distributed noise around predicted ratings.

$$p(R|U,V,\sigma^2) = \prod_{i=1}^{N}\prod_{j=1}^{M}\left[\mathcal{N}(R_{ij}|U_i^T V_j,\sigma^2)\right]^{I_{ij}}$$

Equation 2 Likelihood function.

```
    lambda_U = parameters['lambda_U']
    lambda_V = parameters['lambda_V']
    U = parameters['U']
    V = parameters['V']

    # Likelihood component
    UV = np.dot(U.T, V)
    R_UV = (R[R > 0] - UV[R > 0])  # Difference for observed ratings
```

Code 1 Snippet of likelihood function

Why use a Gaussian likelihood?

- A Gaussian distribution is a natural choice for modelling real-valued data, such as ratings. It assumes that the observed ratings are normally distributed around the predicted ratings with some noise.
- This approach provides a probabilistic framework to quantify uncertainty in predictions.

Limitations:

- **Assumption of Gaussian Distribution:** The model assumes that the ratings are normally distributed around the predicted values. This may not always hold true, as ratings are often ordinal (e.g., 1 to 5 stars) or discrete.
- **Uniform Variance:** A single variance is used across all ratings, which ignores the possibility that different users or items may have different levels of variability.
- **Inability to Model Non-Linear Relationships**

Prior Distributions

The prior distributions act as a regularization mechanism to prevent overfitting when estimating the user (*U*) and item (*V*) feature matrices.

$$p(\mathbf{U}|\sigma_u^2) = \prod_{i=1}^{n} N(\mathbf{u}_i|0, \sigma_u^2\mathbf{I}) \qquad p(\mathbf{V}|\sigma_v^2) = \prod_{j=1}^{p} N(\mathbf{v}_j|0, \sigma_v^2\mathbf{I})$$

Equation 3 Prior distribution

```
    lambda_U = parameters['lambda_U']
    lambda_V = parameters['lambda_V']
    U = parameters['U']
    V = parameters['V']

    # Likelihood component
    UV = np.dot(U.T, V)
    R_UV = (R[R > 0] - UV[R > 0])  # Difference for observed ratings

    # Priors (regularization terms)
    U_prior = lambda_U * np.sum(U**2)
    V_prior = lambda_V * np.sum(V**2)
```

Code 2 Snippet of Prior distribution following likelihood function.

Why use priors?

- The priors ensure that *U* and *V* remain small and do not grow excessively large during training, which helps the model generalize better.
- They make the model robust, especially in cases where the data is sparse (many missing entries in *R*).

Limitations:

- **Rigid Regularization:** The priors assume a zero-mean Gaussian distribution, which may not always align with the true distribution of user and item features.
- **Independence Assumption:** User and item priors are assumed to be independent, which may not reflect real-world scenarios where features can be interdependent.

Posterior Distribution

The posterior distribution combines the likelihood function and the priors using Bayes' Rule to estimate the model parameters *U* and *V*.

Use the log-posterior (sum of log-likelihood and log-priors) to find optimal *U* and *V*.

$$L = -\frac{1}{2}\left(\sum_{i=1}^{N}\sum_{j=1}^{M}(R_{ij} - U_i^T V_j)^2_{(i,j)\in\Omega_{R_{ij}}} + \lambda_U \sum_{i=1}^{N} \|U_i\|^2_{Fro} + \lambda_V \sum_{j=1}^{M} \|V_j\|^2_{Fro}\right)$$

Equation 4 Log-posterior

```
In [28]: def log_a_posteriori():
             """
             Computes the log-posterior for the PMF model

             Components:
             - Likelihood: Gaussian likelihood for observed ratings.
             - Priors: Regularization terms for U and V matrices.
             """
             lambda_U = parameters['lambda_U']
             lambda_V = parameters['lambda_V']
             U = parameters['U']
             V = parameters['V']

             # Likelihood component
             UV = np.dot(U.T, V)
             R_UV = (R[R > 0] - UV[R > 0])  # Difference for observed ratings

             # Priors (regularization terms)
             U_prior = lambda_U * np.sum(U**2)
             V_prior = lambda_V * np.sum(V**2)

             return -0.5 * (np.sum(R_UV**2) + U_prior + V_prior)
```

Code 3 Snippet of Log-posterior function

Why optimize the log-posterior?

- The logarithm converts the product terms in the posterior into a sum, making differentiation and optimization tractable.
- It simplifies computation while preserving the relationships between the terms.

Limitations:

- **Complexity of Computation:** Computing the posterior distribution involves combining the likelihood and priors, which can be computationally intensive for large datasets.
- **Approximation Errors:** In practice, the posterior is approximated using gradient descent or other optimization methods, which may lead to suboptimal solutions.

Gradient Descent

To find the optimal *U* and *V*, PMF employs iterative updates using gradient descent.

Updates parameters by computing gradients of the log-posterior

```
In [27]: def gradient_descent_step(learning_rate):
             """
             Performs a single gradient descent step to update U and V matrices.
             Gradients are computed with respect to the log-posterior.

             """
             U = parameters['U']
             V = parameters['V']
             lambda_U = parameters['lambda_U']
             lambda_V = parameters['lambda_V']

             # Gradients for U and V
             U_grad = -np.dot(V, (R - np.dot(U.T, V)).T) + lambda_U * U
             V_grad = -np.dot(U, (R - np.dot(U.T, V))) + lambda_V * V

             # Update U and V
             U -= learning_rate * U_grad
             V -= learning_rate * V_grad

             parameters['U'] = U
             parameters['V'] = V
```

Code 4 Snippet of gradient descent

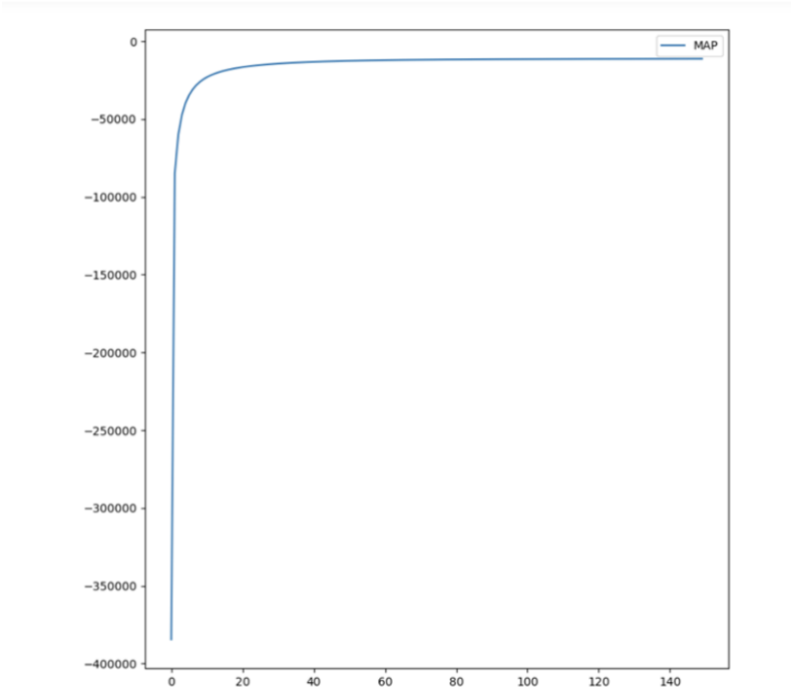
Why gradient descent?

- PMF deals with high-dimensional data (e.g., millions of users and items). Gradient-based methods efficiently optimize such large-scale problems.
- It allows for iterative improvements, enabling convergence to a solution over time.

Limitations:

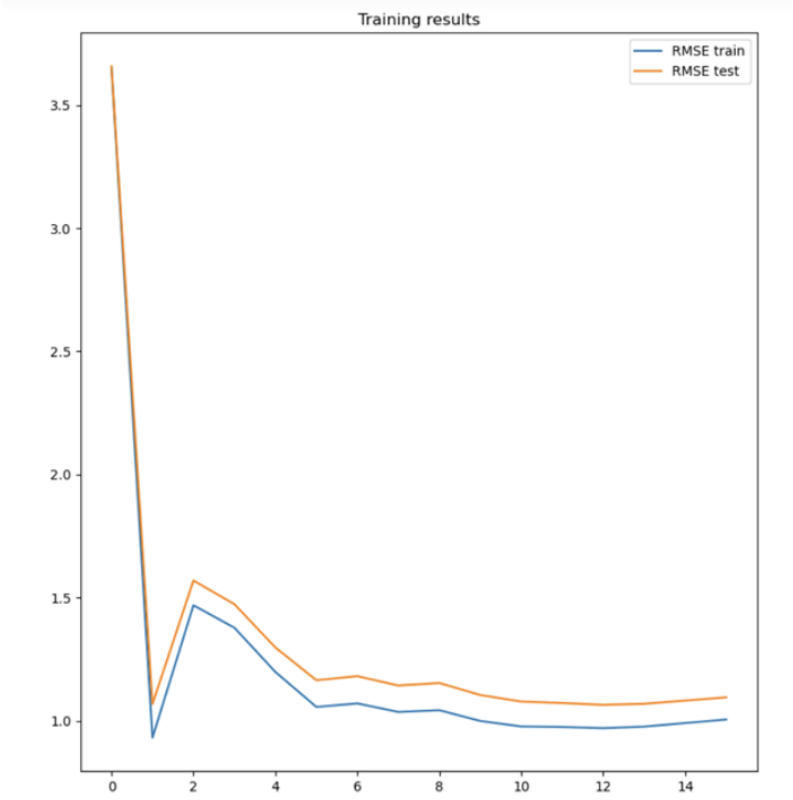
- **Non-Convex Optimization:** The log-posterior function is non-convex, meaning optimization may converge to local minima instead of the global minimum.
- **Sensitivity to Initialization:** The optimization process heavily depends on the initial values of *U* and *V*. Poor initialization can result in slow convergence or suboptimal solutions.
- **Tuning Requirements:** Choosing appropriate learning rates and regularization hyperparameters is challenging and often requires trial and error.

Graphs evaluating performance



Graph 1 Evaluating MAP value of model

The curve shows how the MAP value increases over iterations, indicating the optimization process. The model converges as the curve flattens.



Graph 2 Comparing RMSE train and RMSE test

RMSE Training and Testing The two lines compare the RMSE on the training set (blue) and the test set (orange). The test RMSE initially decreases but later stabilizes or slightly increases, which may indicate overfitting beyond a certain number of iterations.

```
In [35]: print('RMSE of training set:', evaluate(train_set))
         print('RMSE of testing set:', evaluate(test_set))

RMSE of training set: 1.0053875826550782
RMSE of testing set: 1.095324281943083
```

Code 5 Printing RMSE training and test value

RMSE of training set: 1.0053
RMSE of testing set: 1.0953
These values show that the model performs slightly better on the training set compared to the testing set, which is expected due to overfitting to the training data.

```
In [39]: predictions = np.zeros((n_movies, 1))
movie_to_column_items = np.array(list(movie_to_column.items()))
df_rows = []  # List to store rows for the DataFrame

# Generate predictions for all movies for the given user
for i, movie in enumerate(movie_to_column_items):
    predictions[i] = predict(user_id, movie[0])

# Sort predictions in descending order
indices = np.argsort(-predictions, axis=0)

# Get top 10 movies
for j in range(10):
    movie_id = int(movie_to_column_items[np.where(movie_to_column_items[:, 1] == indices[j])][0][0])
    df_row = {
        'UserID': user_id,
        'MovieID': movie_id,
        'Movie': df_movies[df_movies['movieId'] == movie_id].iloc[0]['title'],
        'Genres': df_movies[df_movies['movieId'] == movie_id].iloc[0]['genres'],
        'Prediction': predictions[indices[j]][0][0]
    }
    df_rows.append(df_row)

# Create a DataFrame from the collected rows
df_result = pd.DataFrame(df_rows)

df_result
```

Code 6 Snippet of predicting movie rating

Out[39]:

	UserID	MovieID	Movie	Genres	Prediction
0	45	1378	Young Guns (1988)	Action|Comedy|Western	3.600125
1	45	102903	Now You See Me (2013)	Crime|Mystery|Thriller	3.594754
2	45	133419	Pitch Perfect 2 (2015)	Comedy	3.568748
3	45	3441	Red Dawn (1984)	Action|Drama|War	3.566968
4	45	89904	The Artist (2011)	Comedy|Drama|Romance	3.546616
5	45	30707	Million Dollar Baby (2004)	Drama	3.528642
6	45	106487	The Hunger Games: Catching Fire (2013)	Action|Adventure|Sci-Fi|IMAX	3.528182
7	45	55721	Elite Squad (Tropa de Elite) (2007)	Action|Crime|Drama|Thriller	3.512346
8	45	3969	Pay It Forward (2000)	Drama	3.490879
9	45	67255	Girl with the Dragon Tattoo, The (Män som hata...	Crime|Drama|Mystery|Thriller	3.487619

Code 7 Snippet of table containing predicted ratings for different movie.

Advantages

Why PMF over Other Methods?

1. **Scalability:**
    - PMF models R using low-rank matrices U and V, making it computationally efficient and suitable for large-scale datasets like Netflix or Amazon.
2. **Handling Sparsity:**
    - Collaborative filtering inherently faces sparse data (many users rate only a few items). PMF leverages the probabilistic framework to fill these gaps effectively by utilizing priors and Gaussian likelihoods.
3. **Bayesian Framework:**
    - Unlike traditional matrix factorization (e.g., SVD), PMF incorporates uncertainty handling through Bayesian inference, making it more robust to noise and sparse data.
4. **Regularization:**
    - PMF uses priors to prevent overfitting, which is critical when user-item matrices are sparse or datasets are small.
5. **Interpretability:**
    - Probabilistic modelling provides interpretable results, allowing us to quantify uncertainties in predictions.

Applications

1. Recommendation Systems : PMF is widely used in recommendation systems to predict user preferences for items they have not yet interacted with. By analysing patterns in user-item interaction matrices, PMF fills in missing entries to make personalized recommendations.

Example

- **Netflix**: Suggesting movies or TV shows based on a user's previous viewing history and ratings.
- **Spotify**: Recommending playlists or songs based on a user's listening habits and preferences.
- **Amazon**: Suggesting products to users based on their browsing or purchasing history.

2. Rating Prediction : PMF can predict how a user might rate an item (e.g., a movie, book, or product) they haven't yet interacted with. This is achieved by approximating user preferences and item characteristics.

Example:

- **IMDB**: Predicting how a user might rate a movie they haven't watched, based on similar users' ratings.

- **Goodreads**: Estimating how a reader might rate a book they haven't read yet.

Further areas where this can be applied :

3. Education

- **Use**: Adaptive learning platforms like Coursera or Khan Academy can use PMF to recommend courses or study materials tailored to a student's strengths and weaknesses.
- **Example**: Predicting which topics, a student might struggle with based on their interaction with previous topics.

4. Healthcare

- **Use**: Recommending personalized treatments or predicting the effectiveness of medications for specific patients based on historical patient data.
- **Example**: Suggesting alternative therapies to patients with similar medical profiles.

References

[1] https://towardsdatascience.com/pmf-for-recommender-systems-cbaf20f102f0

[2] https://towardsdatascience.com/probabilistic-matrix-factorization-b7852244a321

[3] **Probabilistic Matrix Factorization Ruslan Salakhutdinov and Andriy Mnih**
Department of Computer Science, University of Toronto 6 King's College Rd, M5S 3G4, Canada {rsalakhu,amnih}@cs.toronto.edu

---

GitHub Link : https://github.com/Imparth2803/ML_NN