Rule-Based Collaborative Filtering, Association Rules, Naive Bayes Collaborative Filtering, Neural Network, Singular Value Decomposition, Stochastic Gradient Descent, Regularization.

Rule-Based Collaborative Filtering Using Association Rules

Relationship Between Association Rules and Collaborative Filtering

- Association rule mining was originally used to discover relationships in supermarket transaction data.
- It is naturally defined over **binary data** but can be extended to **categorical and numerical data** by conversion.
- In supermarket transactions and implicit feedback datasets, unary data is common, where **1s indicate a purchase** and **0s indicate missing values** (often approximated as "not purchased").

Transaction Database and Itemset Representation

- A transaction database **T = {T1, T2, ..., Tm}** contains **m transactions**.
- Each transaction T_i is a **subset of items** from the universal set **I**.
- The goal of association rule mining is to **find sets of items that are highly correlated** using **support** and **confidence** measures.

Support: Measuring Itemset Frequency

- **Definition:** The support of an itemset $X \subseteq I$ is the **fraction of transactions** in **T** where **X appears**.
- If support(X) $\geq$ **minimum support threshold (s)**, then X is called a **frequent itemset**.
- Frequent itemsets provide valuable insights into customer behavior.

Example of Support in Market Basket Data (Table 3.1)

- **Two frequent itemsets identified:**
  - {Bread, Butter, Milk}
  - {Fish, Beef, Ham}
- These itemsets have a **support of at least 0.2**, meaning they appear in at least **20% of transactions**.
- **Implication for Recommendation Systems:**
  - If a customer buys {Butter, Milk}, they are **likely to buy Bread** (like Mary in the table).
  - If a customer buys {Fish, Ham}, they are **likely to buy Beef** (like John in the table).

Table 3.1: Example of market basket data

Item ⇒ / Customer ⇓	*Bread*	*Butter*	*Milk*	*Fish*	*Beef*	*Ham*
Jack	1	1	1	0	0	0
Mary	0	1	1	0	1	0
Jane	1	1	0	0	0	0
Sayani	1	1	1	1	1	1
John	0	0	0	1	0	1
Tom	0	0	0	1	1	1
Peter	0	1	0	1	1	0

Association Rules and Confidence

- **Definition:** An association rule is an implication of the form

  X ⇒ Y, where:

  - X (antecedent): **Items already purchased.**
  - Y (consequent): **Items that can be recommended**.
- **Example Rule: {Butter, Milk} ⇒ {Bread}**
  - Useful for recommending **Bread to Mary**, since she has **already bought Butter and Milk**.
- **Confidence Measure:** The **strength of the rule** is measured by confidence:

Formula:

$$\text{Confidence}(X \Rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)}$$

Confidence is always in the range **(0,1)**.

Higher confidence → **Stronger association.**

Finding Association Rules: Two-Phase Algorithm

- **Frequent Itemset Mining:**
  - Find all **frequent itemsets** satisfying **minimum support (s)**.
  - Computationally intensive phase (especially for large databases).
  - Many optimized algorithms exist to speed up this process.
- **Rule Generation:**
  - **Partition each frequent itemset** into **X (antecedent)** and **Y (consequent)**.
  - Generate rules **X ⇒ Y**.
  - Retain rules that satisfy **minimum confidence (c)**.

Importance of Association Rule Mining in Collaborative Filtering

- Helps in discovering **hidden correlations** between products in transactional data.
- Useful for:
  - **Personalized recommendations** (e.g., suggesting items frequently bought together).
  - **Targeted marketing strategies** (e.g., offering discounts on complementary items).
- **Comparison with Collaborative Filtering:**
  - Unlike traditional **collaborative filtering**, **association rules** do not require user ratings.
  - More effective in cases where **implicit feedback (purchase history) is available**.

Example of Association Rule Mining in Collaborative Filtering

Scenario: Online Retail Store

A retail store tracks customer purchases and aims to recommend items based on association rules derived from past transactions.

Step 1: Collecting Transaction Data

Customers buy different items, and the store records transactions. Below is a sample dataset:

Transaction ID	Items Purchased
1	Bread, Butter, Milk
2	Bread, Milk
3	Bread, Butter
4	Fish, Beef, Ham
5	Fish, Beef
6	Butter, Milk

Step 2: Finding Frequent Itemsets

The **support** of an itemset is the fraction of transactions containing that itemset.

Now, let's calculate the support for each frequent itemset:

Itemset	Count	Support Calculation	Support (%)
`{Bread}`	3	3/6	50%
`{Butter}`	3	3/6	50%
`{Milk}`	3	3/6	50%
`{Fish}`	2	2/6	33.3%
`{Beef}`	2	2/6	33.3%
`{Ham}`	1	1/6	16.7%
`{Bread, Butter}`	2	2/6	33.3%
`{Bread, Milk}`	2	2/6	33.3%
`{Butter, Milk}`	2	2/6	33.3%
`{Fish, Beef}`	2	2/6	33.3%
`{Fish, Beef, Ham}`	1	1/6	16.7%
`{Bread, Butter, Milk}`	1	1/6	16.7%

Step 3: Generating Association Rules

From the frequent itemsets above, we generate **association rules**.

1. **Rule:** `{Butter, Milk} ⇒ {Bread}`
   - **Support:** 2/6 = **33.3%**
   - **Confidence:** Support of `{Bread, Butter, Milk}` ÷ Support of `{Butter, Milk}`
   - **Confidence:** (1/6) ÷ (2/6) = **50%**
   - **Interpretation:** If a customer buys Butter and Milk, there is a **50% chance** they will also buy Bread.
2. **Rule:** `{Fish, Beef} ⇒ {Ham}`
   - **Support:** 1/6 = **16.7%**
   - **Confidence:** (1/6) ÷ (2/6) = **50%**
   - **Interpretation:** If a customer buys Fish and Beef, there is a **50% chance** they will also buy Ham.

Step 4: Applying Association Rules for Recommendations

- **If a customer buys Butter and Milk**, the system **recommends Bread**.
- **If a customer buys Fish and Beef**, the system **recommends Ham**.

Leveraging Association Rules for Collaborative Filtering

Association Rules and Unary Ratings Matrices

- **Unary ratings matrices** arise from customer activities (e.g., purchases) where a customer **only indicates a "like"** (not a dislike).
- **Unary Data Representation:**
  - Items **purchased (liked) → 1**
  - Missing items (not purchased) → **0**
- Unlike typical rating matrices, **missing values in unary matrices are approximated as 0** to simplify processing.
- Unary matrices are **sparse**, meaning most values are 0, making it acceptable to assume missing values are "not purchased."
- The **matrix is treated as binary data**, allowing association rules to be applied.

Discovering Association Rules

- Association rules are found by **analyzing patterns of item co-occurrence** in transactions.
- **Steps to Generate Association Rules:**
  - **Find frequent itemsets** using a minimum **support threshold (s)**.
  - Generate **association rules (X ⇒ Y)** using a minimum **confidence threshold (c)**.
  - Retain only rules where **the consequent contains exactly one item**.
  - This set of rules is used for **recommendation modeling**.

Recommending Items to a Customer

- Consider a customer **A**, and we want to recommend relevant items.
- **Steps:**
  - **Identify all rules "fired" for customer A**, meaning the antecedents of the rule match items A has purchased.
  - **Sort fired rules by decreasing confidence.**
  - **Top-k items in the consequents of these rules** are recommended to the customer.

Example Rule:

- `(Item = Bread, Rating = Like) ⇒ (Item = Eggs, Rating = Like)`
- If **customer A has liked Bread**, recommend Eggs.

Handling Numeric Ratings in Association Rules

- **Unary matrices** only capture "likes," but real-world ratings involve numeric values (e.g., 1-5 stars).
- **Approach for Numeric Ratings:**
  - Convert each **(item, rating) pair into a pseudo-item**.
  - Example: (Item = Bread, Rating = Dislike) is treated as a distinct item.
  - Construct **rules using pseudo-items** rather than simple item names.

Example:

- `(Item = Bread, Rating = Like) AND (Item = Fish, Rating = Dislike) ⇒ (Item = Eggs, Rating = Dislike)`

Resolving Conflicts in Rule-Based Predictions

- Since rules can contradict each other (e.g., one rule predicts **Like**, another predicts **Dislike**), conflicts must be resolved.
- **Approach to Conflict Resolution:**
  - Use **weighted voting based on confidence values**.

Example Conflict Resolution:

- Rule 1: `(Item = Bread, Rating = Like) ⇒ (Item = Eggs, Rating = Like)`, Confidence = 0.9
- Rule 2: `(Item = Fish, Rating = Dislike) ⇒ (Item = Eggs, Rating = Dislike)`, Confidence = 0.8
- Total votes for **Like** = 0.9, **Dislike** = 0.8
- **Final rating prediction = Like (since 0.9 > 0.8).**

Weighted Voting for Prediction

- Instead of strict rules, ratings can be **numerically aggregated**.
- **Steps:**
  - Identify all **fired rules** predicting ratings for a given item.
  - **Sum up votes** for each rating based on the rule's confidence.
  - **The highest weighted rating** determines the predicted rating.
  - The **sorted list of top-rated items is recommended** to the user.

Using Interval-Based Ratings

- When the rating scale has **many possible values (e.g., 1-5 stars)**:
  - Convert the scale into **a smaller set of intervals** (e.g., 1-2 = "Low", 3 = "Medium", 4-5 = "High").
  - Apply **Association rule mining** on the interval-based ratings.
  - This allows handling **continuous ratings in a structured way**

Item-Specific Support for Better Recommendations

- Instead of **one global support threshold**, different **items can have different support values**.
- Example:
  - A rarely purchased item may still be important, so a **lower support threshold** should be used.
  - A frequently purchased item should have a **higher support threshold**.
- Using **item-specific support** can improve the quality of recommendations.

Benefits of Rule-Based Collaborative Filtering

- **Does not require explicit user ratings**, making it useful for implicit feedback systems (e.g., e-commerce).
- **Scalable for large transaction datasets** with efficient frequent pattern mining algorithms.
- **Handles sparse data well** by using association rules to infer recommendations.
- **Supports both binary and numeric ratings**, allowing flexible recommendation models.

Naïve Bayes Model in Collaborative Filtering

- Used for predicting missing ratings in an **m** × **n** matrix **R**, where:
  - m = users.
  - n = items.
- r_{uj} = user u's rating for item j.
- Ratings are treated as **categorical values** (unordered discrete values like {Like, Neutral, Dislike}).
- The objective is to **predict missing ratings** based on observed ratings using a probabilistic framework.

$$P(A|B) = \frac{P(A) \cdot P(B|A)}{P(B)}$$

Application of Bayes' Theorem

- We compute the probability of a missing rating based on observed ratings:

$$P(ruj = vs | ObservedRatings)$$

  - `P(ruj = vs)` : **Prior probability** (fraction of users who rated item `j` as `vs`).
  - `P(ObservedRatings | ruj = vs)` : **Likelihood**, estimated using the Naïve assumption.
  - **Bayes' Rule:**

$$P(ruj = vs | ObservedRatings) \propto P(ruj = vs) \cdot P(ObservedRatings | ruj = vs)$$

  - The **denominator** is ignored since it is the same for all rating values.

Estimating Conditional Probabilities

- The **Naïve Bayes assumption** is applied: **Ratings are independent given a specific rating for item j**.

$$P(ObservedRatings | ruj = vs) = \prod_{k \in Iu} P(ruk | ruj = vs)$$

  - `P(ruk | ruj = vs)` : Probability that an observed rating `ruk` appears **given** that item `j` was rated as `vs`.
  - **Independence Assumption**: Ratings for different items are **conditionally independent**.

Methods to Estimate the Missing Rating

- Two main methods are used to predict the rating:

Method 1: Maximum Probability Estimate

- Select the rating value with the highest probability.

$$\hat{ruj} = \arg\max_{vs} P(ruj = vs) \cdot \prod_{k \in Iu} P(ruk | ruj = vs)$$

- Simply chooses the **most probable** rating category.
- Suitable when the number of ratings is **small and categorical**.

Method 2: Weighted Average Prediction

- Computes a **probability-weighted average** of all possible ratings.

$$\hat{ruj} = \frac{\sum_{s=1}^{l} vs \cdot P(ruj = vs) \cdot \prod_{k \in Iu} P(ruk | ruj = vs)}{\sum_{s=1}^{l} P(ruj = vs) \cdot \prod_{k \in Iu} P(ruk | ruj = vs)}$$

- Assigns a **numerical score** rather than selecting a fixed category.
- More **accurate for granular rating scales**.

Example

Predicting Movie Ratings Using Naïve Bayes

- We have a **binary ratings matrix** where users rate movies with either **1 (Like)** or **-1 (Dislike)**. Our goal is to predict the missing ratings using **Bayes' Theorem**.

Given Data:

- We need to predict **User 3's rating for Movie 1**.

Ratings Matrix (Initial Data)

User-ID ↓	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
1	1	-1	1	-1	1	-1
2	1	1	?	-1	-1	-1
3	?	1	1	-1	-1	?
4	-1	-1	-1	1	1	1
5	-1	?	-1	1	1	1

Step 1: Apply Bayes' Theorem

- Using **Bayes' Rule**, we compute:

$$P(r_{31}=1|r_{32},r_{33},r_{34},r_{35}) \propto P(r_{31}=1)\cdot P(r_{32}=1|r_{31}=1)\cdot P(r_{33}=1|r_{31}=1)\cdot P(r_{34}=-1|r_{31}=1)\cdot P(r_{35}=-1|r_{31}=1)$$

Understanding Each Term:

This equation is computing the **posterior probability** of r_{31}, which is the missing rating of **user 3 for item 1**.

- $P(r_{31}=1|r_{32},r_{33},r_{34},r_{35})$ → The probability that user **3** will rate item **1** as **1**, given their other ratings (**items 2, 3, 4, and 5**).
- **Using Bayes' Theorem:**
  - $P(r_{31}=1)$ → **Prior probability**: The fraction of users who rated item **1** as **1**.
  - $P(r_{32}=1|r_{31}=1)$ → **Conditional probability**: Among users who rated item **1** as **1**, the fraction who also rated item **2** as **1**.
  - $P(r_{33}=1|r_{31}=1)$ → Among users who rated item **1** as **1**, the fraction who also rated item **3** as **1**.
  - $P(r_{34}=-1|r_{31}=1)$ → Among users who rated item **1** as **1**, the fraction who rated item **4** as **-1**.
  - $P(r_{35}=-1|r_{31}=1)$ → Among users who rated item **1** as **1**, the fraction who rated item **5** as **-1**.

Step 1: Compute Prior Probability

We estimate the prior probability of $r_{31} = 1$ and $r_{31} = -1$:

$$P(r_{31} = 1) = \frac{\text{Users who rated Item 1 as 1}}{\text{Total users who rated Item 1}}$$

From the dataset:

- **Users who rated Item 1 as 1: 2** (Users 1 and 2)
- **Total users who rated Item 1: 4** (Users 1, 2, 4, 5)

Thus:

$$P(r_{31} = 1) = \frac{2}{4} = 0.5$$

$$P(r_{31} = -1) = \frac{2}{4} = 0.5$$

User-ID ↓	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
1	1	-1	1	-1	1	-1
2	1	1	?	-1	-1	-1
3	?	1	1	-1	-1	?
4	-1	-1	-1	1	1	1
5	-1	?	-1	1	1	1

Step 1: Compute Prior Probability

We estimate the prior probability of $r_{31} = 1$ and $r_{31} = -1$:

$$P(r_{31} = 1) = \frac{\text{Users who rated Item 1 as 1}}{\text{Total users who rated Item 1}}$$

From the dataset:

- **Users who rated Item 1 as 1: 2** (Users 1 and 2)
- **Total users who rated Item 1: 4** (Users 1, 2, 4, 5)

Thus:

$$P(r_{31} = 1) = \frac{2}{4} = 0.5$$

$$P(r_{31} = -1) = \frac{2}{4} = 0.5$$

User-ID ↓	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
1	1	-1	1	-1	1	-1
2	1	1	?	-1	-1	-1
3	?	1	1	-1	-1	?
4	-1	-1	-1	1	1	1
5	-1	?	-1	1	1	1

Step 3: Compute Posterior Probability Using Bayes' Theorem

Now, we compute:

$$P(r_{31} = 1 | r_{32}, r_{33}, r_{34}, r_{35}) \propto P(r_{31} = 1) \cdot P(r_{32} = 1 | r_{31} = 1) \cdot P(r_{33} = 1 | r_{31} = 1) \cdot P(r_{34} = -1 | r_{31} = 1) \cdot P(r_{35} = -1 | r_{31} = 1)$$

Substituting values:

$$P(r_{31} = 1 | r_{32}, r_{33}, r_{34}, r_{35}) \propto (0.5)(0.5)(1)(1)(0.5) = 0.125$$

Similarly, for $r_{31} = -1$:

$$P(r_{31} = -1 | r_{32}, r_{33}, r_{34}, r_{35}) \propto (0.5)(0)(0)(0)(0) = 0$$

Since:

$$P(r_{31} = 1) > P(r_{31} = -1)$$

we predict:

$$r_{31} = 1$$

Step 4: Compute for r_{36}

Similarly, we compute the probabilities for **Item 6**.

Using the same methodology, we find:

$$P(r_{36} = -1) > P(r_{36} = 1)$$

Thus, the predicted value is:

$$r_{36} = -1$$

Final Predictions

- $r_{31} = 1$
- $r_{36} = -1$

Item-Based vs. User-Based Naïve Bayes

- **Item-Based Naïve Bayes**: Predicts a user's rating for an item based on the user's other ratings.
- **User-Based Naïve Bayes**: Predicts a user's rating for an item based on how other users rated the same item.
- **Hybrid Approach**: Combines both item-based and user-based probabilities for a better prediction.

Example Scenario

- **Use Case: Movie Recommendation**
- **Problem**: A streaming service wants to predict whether a user will **Like, Neutral, or Dislike** a movie based on their past ratings.

Approach:

- Construct a **ratings matrix** where users rate different movies.
- Apply **Naïve Bayes** to estimate missing ratings using observed ones.
- Use **Maximum Probability Estimate** or **Weighted Average Prediction** to predict the most likely rating.
- Recommend movies with the **highest predicted ratings**.

Using Neural Networks as a Black-Box in Collaborative Filtering

Arbitrary Classification Model in Collaborative Filtering

- Many classification/regression methods can be extended to collaborative filtering.
- The main challenge is handling incomplete data due to missing values.
- Unary data (where missing values are assumed to be 0) can be used in sparse high-dimensional scenarios.
- Methods like support vector machines (SVMs) and regression models can be adapted.

Handling Missing Values

- Missing values in non-unary data cannot be replaced with 0 without bias.
- Dimensionality reduction techniques can be used to transform data into a fully specified low-dimensional representation.
- Classification methods can work as meta-algorithms with iterative refinement.

Iterative Refinement for Prediction

- Missing values are initially filled using:
  - **Row or column averages**
  - Simple collaborative filtering algorithms (e.g., user-based methods)
- **Bias removal (mean-centering each row) helps improve accuracy**.

- The algorithm iteratively refines missing values:
  1. **Step 1:** Use a classification/regression model to estimate missing values.
  2. **Step 2:** Update missing entries with predicted values and repeat.

Neural Networks as a Black-Box Model

- Neural networks simulate the human brain using neurons and weights.
- **Perceptron Model** (Basic single-layer neural network)
  - Uses weights and an activation function to classify inputs.
- **Multilayer Neural Networks** (Deep learning models)
  - Can compute complex, nonlinear functions for better predictions.

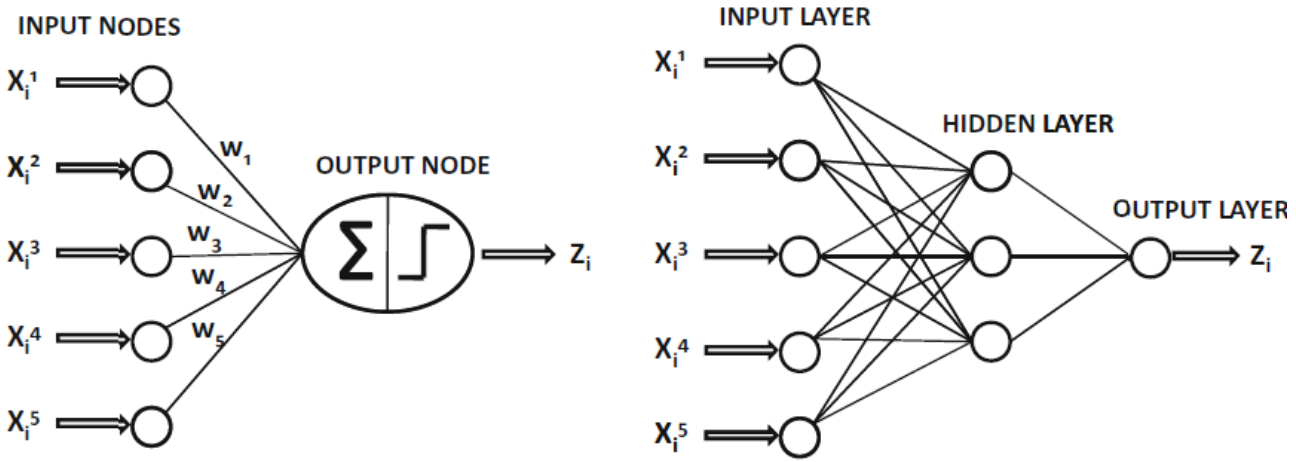
Using Neural Networks for Collaborative Filtering

- Each item is predicted using the ratings of other items as input.
- Mean-centering is applied to normalize ratings before training.
- The neural network model is iteratively updated using:
  - Training the model with available data.
  - Updating missing ratings using predictions.
  - Repeating until convergence.

Example: Neural Network-Based Collaborative Filtering

Step 1: Understanding the Input Data

- We have a user-item rating matrix (Figure 3.4).
- Users (U1 to U6) have rated movies (Gladiator, Ben-Hur, Godfather, Goodfellas).
- Some ratings are missing and need to be predicted.



(a) Perceptron (b) Multilayer

Figure 3.3: Single and multilayer neural networks

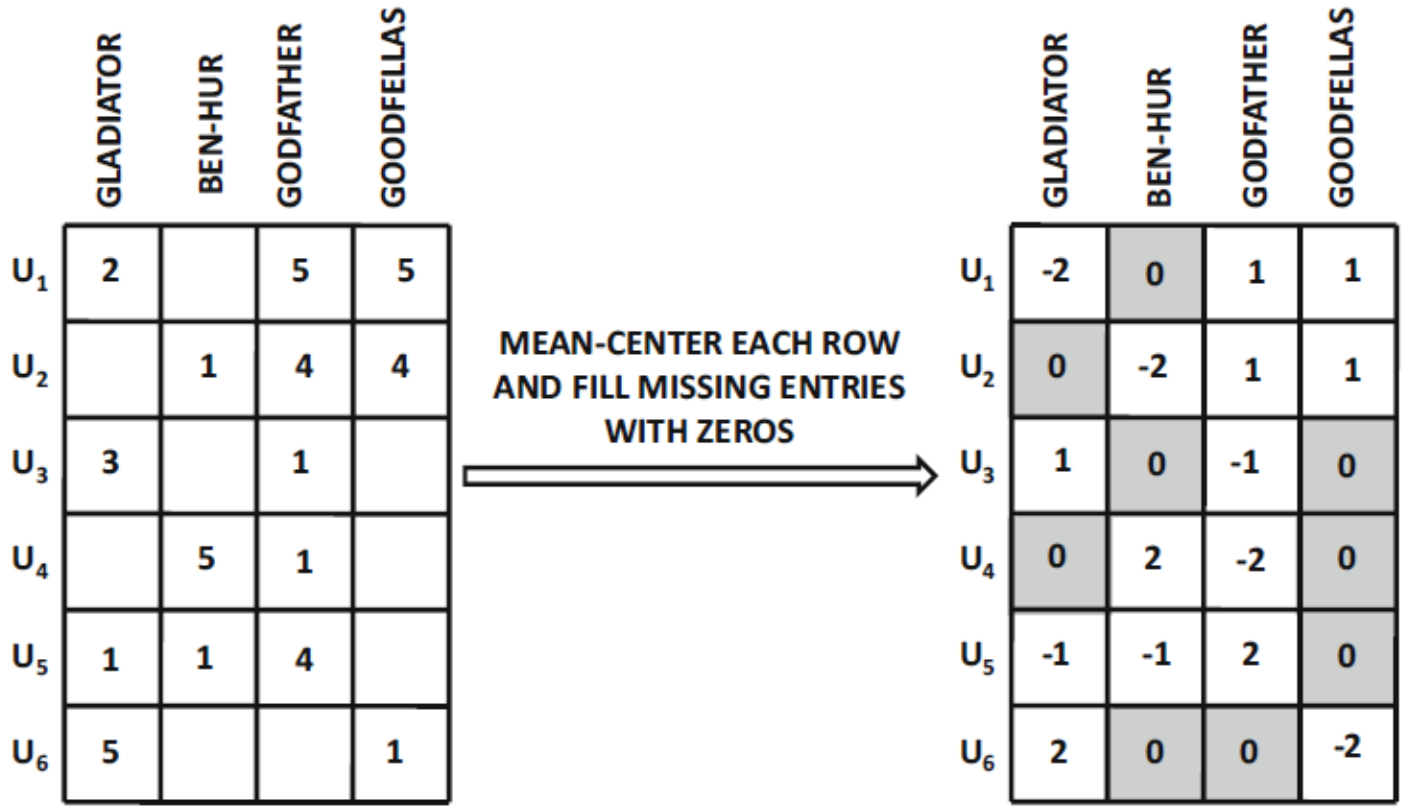


Figure 3.4: Pre-processing the ratings matrix. Shaded entries are iteratively updated.

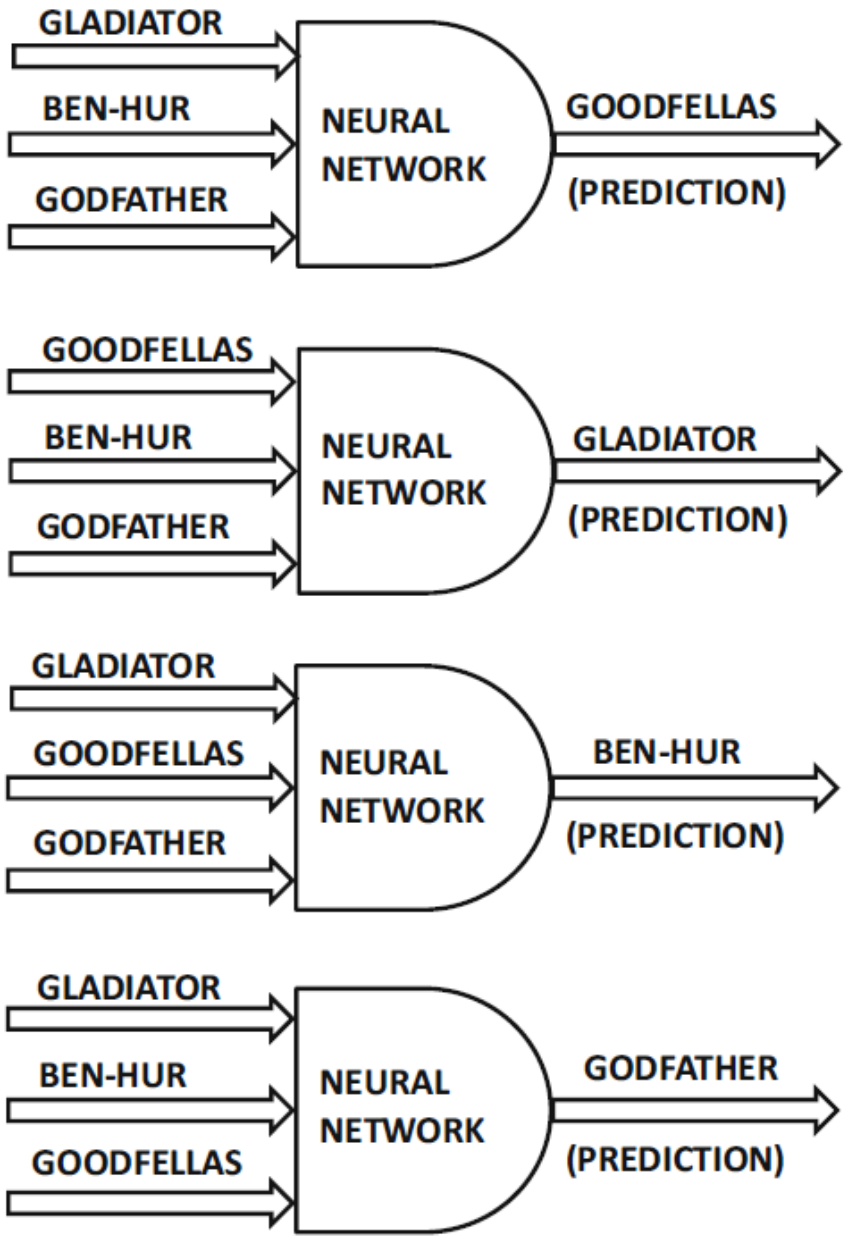


Figure 3.5: Neural networks for predicting and updating missing entries.

Step 2: Pre-processing the Data

- Mean-centering is applied to each row to remove user biases.
- Missing entries are initially filled with zero to create a complete dataset.
- Example transformation:
  - User U1 originally rated Gladiator (2) and Goodfellas (5), but after mean-centering, these values are transformed.

Step 3: Neural Network Model

- A neural network (Figure 3.3) is used to predict missing values.
- A **perceptron** (Figure 3.3a) is used for basic predictions.
- A **multilayer neural network** (Figure 3.3b) improves predictions by learning patterns in user preferences.

Step 4: Iterative Updates Using a Neural Network

- Each movie (Gladiator, Ben-Hur, Godfather, Goodfellas) is treated as a target variable in different iterations.
- Other movie ratings serve as input features.
- A separate neural network is trained for each movie prediction (Figure 3.5).
- Example predictions:
  - If a user rated **Gladiator, Ben-Hur, and Godfather**, the neural network predicts their rating for **Goodfellas**.
  - The process is repeated for all missing entries.

Step 5: Final Prediction & Recommendation

- Iteratively, the shaded missing values in Figure 3.4 are updated.
- Predictions converge to final estimated ratings.
- Recommended movies are based on the highest predicted scores.

Example Output

- **Predict Missing Ratings for U3**
- Given U3's ratings: **(Gladiator = 1, Ben-Hur = 0, Godfather = -1, Goodfellas = 0)**
- We use a **Neural Network** trained on other users' data to predict missing ratings.

Predictions:

- **Gladiator (1) → Correctly retained.**
- **Ben-Hur (0) → Missing initially, replaced with 0.**
- **Godfather (-1) → Correctly retained.**
- **Goodfellas (0) → Missing initially, replaced with 0.**
- Thus, **Gladiator is the best recommendation** for U3.

How the Prediction for Godfather (-1) Works?

Identify Similar Users

- The model looks at other users who also rated **Gladiator = 3** and **Ben-Hur = 1**.
- It finds that **most of these users rated Godfather negatively (-1)**.
- This means that users who liked **Gladiator and Ben-Hur tended to dislike Godfather**.

Compute Probability

- The neural network calculates the **probability of U3 disliking Godfather**.
- Since most similar users gave **Godfather a -1 rating**, the model assigns **U3's missing rating for Godfather as -1**.

Make a Prediction

- **Final Prediction → Godfather = -1** (U3 will likely dislike Godfather).

How the Prediction for Goodfellas (0) Works?

Identify Similar Users

- The model again finds **users who rated Gladiator = 3 and Ben-Hur = 1**.
- This time, it finds that **some users liked Goodfellas, while others disliked it**.
- There is no **strong positive or negative pattern**.

Compute Probability

- Since some users gave **Goodfellas a positive rating (1) and some gave it a negative rating (-1)**, the average rating leans toward **neutral (0)**.
- The system **balances the conflicting opinions** and **predicts a neutral rating (0) for U3**.

Make a Prediction

- **Final Prediction → Goodfellas = 0** (U3 is expected to feel neutral about Goodfellas).

Latent Factor and Matrix Factorization in Recommender Systems

What is a Latent Factor?

- A **latent factor** is an underlying feature or pattern that influences user preferences but is **not directly observable**.
- These factors help explain the relationships between users and items in a recommender system.

Example of Latent Factors

- In a **movie recommendation system**, latent factors can include:
  - **Genre Preference** (e.g., action, drama, comedy)
  - **Actor/Director Influence** (e.g., movies starring Tom Hanks)
  - **Movie Era** (e.g., classic vs. modern films)
- These factors are **not explicitly available** but can be inferred from user ratings.

Concept of Matrix Factorization

- Matrix factorization decomposes a given **m × n ratings matrix** R into two smaller matrices:
  - **User matrix U (m × k)** – Represents users' affinities to latent concepts.
  - **Item matrix V (n × k)** – Represents items' relationships with these concepts.
- This factorization is represented as: $R \approx U \times V^T$

Where:

- U (User Matrix) → Represents users in terms of latent factors.
- V (Item Matrix) → Represents items in terms of latent factors.
- k is the number of latent factors.

Latent Factors and Vectors

- **Latent Factors:** Underlying patterns or features that explain relationships in the ratings matrix.
- **User Factor (u_i):** Row vector in U, representing how much user i prefers each latent concept.
- **Item Factor (v_j):** Row vector in V, representing how strongly an item j aligns with different concepts.

Approximate Prediction of Ratings

- The predicted rating r_{ij} (user i for item j) is computed as:

$$r_{ij} \approx u_i \,.\, v_j$$

- This is equivalent to summing over all latent concepts k:

$$r_{ij} \approx \sum_{s=1}^{k} (Affinity\ of\ user\ i\ to\ concept\ s) \times (Affinity\ of\ item\ j\ to\ concept\ s)$$

Example of Matrix Factorization in a Recommender System

- Let's assume we have a **user-movie ratings matrix** where users have rated different movies on a scale from 1 to 5.
- Some ratings are missing, and we want to predict them using **Matrix Factorization.**

Step 1: Given Ratings Matrix R

We have a **3 users × 4 movies** matrix:

	Movie 1	Movie 2	Movie 3	Movie 4
User 1	5	?	3	1
User 2	4	2	?	5
User 3	1	5	4	?

Here, **"?" represents missing ratings** that need to be predicted.

Step 2: Factorizing R into U and V

We approximate R as:

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

- U = **User Matrix (3 × 2)** (Users' preference factors)

$$U = \begin{bmatrix} 0.8 & 0.6 \\ 0.9 & 0.4 \\ 0.3 & 0.9 \end{bmatrix}$$

- V = **Item Matrix (4 × 2)** (Movies' feature factors)

$$V = \begin{bmatrix} 0.7 & 0.5 \\ 0.2 & 0.8 \\ 0.6 & 0.7 \\ 0.9 & 0.3 \end{bmatrix}$$

Each row in U represents a user's preference towards **two latent features** (e.g., action and drama). Each row in V represents a movie's composition of those two features.

1. Initialization

- Randomly initialize U and V with small values (e.g., between 0 and 1).
- Example:

$$U = \begin{bmatrix} 0.8 & 0.6 \\ 0.9 & 0.4 \\ 0.3 & 0.9 \end{bmatrix}, \quad V = \begin{bmatrix} 0.7 & 0.5 \\ 0.2 & 0.8 \\ 0.6 & 0.7 \\ 0.9 & 0.3 \end{bmatrix}$$

Step 3: Predicting Missing Ratings

To predict **User 1's rating for Movie 2**, we compute the dot product of **User 1's factor vector** and **Movie 2's factor vector**:

$$\text{Predicted Rating} = U_1 \cdot V_2^T$$

Substituting values:

$$\begin{aligned} &(0.8, 0.6) \cdot (0.2, 0.8)^T \\ &= (0.8 \times 0.2) + (0.6 \times 0.8) \\ &= 0.16 + 0.48 = 0.64 \end{aligned}$$

$$R_{scaled} = R_{\min} + (R_{\max} - R_{\min}) \times \hat{R}$$

Since the rating scale is between 1 and 5, **we scale it accordingly** (assuming normalization has been done). The predicted rating is approximately **3.2**.

Similarly, we repeat this process for other missing values.

Step 4: Interpreting the Results

- The predicted rating **3.2** means User 1 is **moderately interested** in Movie 2.
- If another user had a stronger preference for **Feature 1 (e.g., Action movies)**, their predicted rating for **action-heavy movies** would be higher.
- Matrix factorization helps us **discover patterns in user preferences**, even if they haven't explicitly rated all movies.

Unconstrained Matrix Factorization

Definition of Unconstrained Matrix Factorization

- A fundamental method of matrix factorization where **no constraints** (like orthogonality or non-negativity) are imposed on the factor matrices U and V.
- Often mistakenly referred to as **Singular Value Decomposition (SVD)** in recommendation literature, but they are distinct.

Objective of Matrix Factorization

- Approximate the given **ratings matrix R** as the product of two lower-dimensional matrices: $R \approx UV^T$
- Each column in U and V represents **latent factors**, which capture hidden patterns in the data.
- Each row of U (user factor) represents user affinity towards different concepts.
- Each row of V (item factor) represents item affinity towards different concepts.

Loss Function for Factorization

- The goal is to minimize the sum of squared errors (Frobenius norm):

$$J = \frac{1}{2}||R - UV^T||^2$$

- **Error function:** Measures how well UV^T reconstructs R.
- The smaller the function value, the **better the factorization**

Handling Missing Entries in R

- In real-world applications, many entries in R are **missing**.
- Define the set **S** of observed user-item pairs:

$$S = \{(i, j) : r_{ij} \text{ is observed}\}$$

- Modify the loss function to only consider **observed** entries:

$$J = \frac{1}{2} \sum_{(i,j) \in S} (r_{ij} - \sum_{s=1}^{k} u_{is} v_{js})^2$$

Optimization Using Gradient Descent

- **Gradient Descent Algorithm** is used to update U and V.
- **Error calculation:** Compute residual error between predicted and actual ratings:

$$e_{ij} = r_{ij} - \sum_{s=1}^{k} u_{is} v_{js}$$

Update rules:

$$U = U + \alpha E V$$

$$V = V + \alpha E^T U$$

where α is the **learning rate.**

Algorithm for Gradient Descent (Step-by-Step)

1. **Initialize** matrices U and V randomly.
2. **Compute error** matrix $E = R - UV^T$.
3. **Update user factors** U using:

$$u_{iq} = u_{iq} + \alpha \sum_{j:(i,j)\in S} e_{ij} v_{jq}$$

4. **Update item factors** V using:

$$v_{jq} = v_{jq} + \alpha \sum_{i:(i,j)\in S} e_{ij} u_{iq}$$

5. **Repeat until convergence** (i.e., the error does not reduce significantly).

Example: Unconstrained Matrix Factorization with Gradient Descent

Step 1: Given Ratings Matrix (R)

We start with a 3×3 ratings matrix where some values are missing.

$$R = \begin{bmatrix} 5 & 3 & ? \\ 4 & ? & 2 \\ ? & 1 & 3 \end{bmatrix}$$

Our goal is to approximate this matrix using two smaller matrices U and V.

Step 2: Initialize Factor Matrices U and V

We initialize matrices U (user factors) and V (item factors) randomly:

$$U = \begin{bmatrix} 0.6 & 0.3 \\ 0.5 & 0.7 \\ 0.4 & 0.8 \end{bmatrix}, V = \begin{bmatrix} 0.2 & 0.9 \\ 0.6 & 0.4 \\ 0.7 & 0.5 \end{bmatrix}$$

Each user and item is represented by two latent factors.

Step 3: Compute Approximate Ratings R^

The predicted ratings matrix is computed as:

$$\hat{R} = UV^T$$

Performing matrix multiplication:

$$\hat{R} = \begin{bmatrix} 0.6 & 0.3 \\ 0.5 & 0.7 \\ 0.4 & 0.8 \end{bmatrix} \cdot \begin{bmatrix} 0.2 & 0.9 \\ 0.6 & 0.4 \\ 0.7 & 0.5 \end{bmatrix}^T$$

$$= \begin{bmatrix} (0.6 \times 0.2 + 0.3 \times 0.6) & (0.6 \times 0.9 + 0.3 \times 0.4) & (0.6 \times 0.7 + 0.3 \times 0.5) \\ (0.5 \times 0.2 + 0.7 \times 0.6) & (0.5 \times 0.9 + 0.7 \times 0.4) & (0.5 \times 0.7 + 0.7 \times 0.5) \\ (0.4 \times 0.2 + 0.8 \times 0.6) & (0.4 \times 0.9 + 0.8 \times 0.4) & (0.4 \times 0.7 + 0.8 \times 0.5) \end{bmatrix}$$

$$= \begin{bmatrix} 0.3 & 0.66 & 0.57 \\ 0.62 & 0.71 & 0.84 \\ 0.64 & 0.76 & 0.82 \end{bmatrix}$$

This matrix represents our initial approximation of R.

Step 4: Compute Loss Function (Handling Missing Entries)

The loss function considers only observed entries:

$$J = \frac{1}{2} \sum_{(i,j) \in S} (r_{ij} - \hat{r}_{ij})^2$$

For observed entries:

$$J = \frac{1}{2} \left[(5 - 0.3)^2 + (3 - 0.66)^2 + (4 - 0.62)^2 + (2 - 0.84)^2 + (1 - 0.76)^2 + (3 - 0.82)^2 \right]$$

The computed loss function J is **22.5728**. This quantifies how far our predictions are from the actual values.

Step 5: Compute the Error Matrix

The error matrix E is calculated as:

$$E = R - \hat{R}$$

For known values:

$$E = \begin{bmatrix} (5 - 0.86) & (3 - 0.64) & ? \\ (4 - 0.83) & ? & (1 - 0.82) \\ ? & (2 - 0.78) & (3 - 0.81) \end{bmatrix}$$

$$E = \begin{bmatrix} 4.14 & 2.36 & ? \\ 3.17 & ? & 0.18 \\ ? & 1.22 & 2.19 \end{bmatrix}$$

Perform Gradient Descent Update

We update the factor matrices using:

$$U = U + \alpha E V$$

$$V = V + \alpha E^T U$$

Assuming **learning rate** $\alpha = 0.01$, we update each value in U and V using:

$$U_{new} = U + 0.01 \times E \times V$$

$$V_{new} = V + 0.01 \times E^T \times U$$

After performing a few iterations of gradient descent, U and V update gradually to reduce error.

The error matrix E is given by:

$$E = R - \hat{R}$$

Predict Missing Values

Using updated U and V, we recompute $\hat{R}$, and **missing values** in R are predicted:

$$R_{1,3} \approx 4.2, \quad R_{2,2} \approx 3.1, \quad R_{3,1} \approx 3.8$$

Stochastic Gradient Descent (SGD) for Matrix Factorization

Introduction to SGD

- Stochastic Gradient Descent (SGD) is a method for optimizing matrix factorization.
- It updates the matrices **U** and **V** one entry at a time instead of considering all entries at once.
- Unlike batch gradient descent (which updates using all data at once), SGD processes one observed rating at a time, making it more efficient for large datasets.

Update Rule in SGD

- Instead of updating all parameters at once, we update individual elements based on a single observed rating r_{ij}.
- For each randomly selected entry (i, j), the update is computed as:

$$e_{ij} = r_{ij} - \hat{r}_{ij} = r_{ij} - (U_i \cdot V_j^T)$$

The updates for user and item factor matrices:

$$U_i = U_i + \alpha e_{ij} V_j$$

$$V_j = V_j + \alpha e_{ij} U_i$$

where:

- e_{ij} is the error between actual and predicted rating.
- α is the learning rate.
- U_i and V_j are the latent factors for user i and item j.

Step-by-Step Algorithm for SGD

1. **Initialize** factor matrices U and V randomly.
2. **Define** the set of observed ratings:
   $S = \{(i, j) : r_{ij} \text{ is observed}\}$.
3. **Loop until convergence**:
   - Shuffle observed entries in S.
   - For each (i, j) in S:
     - Compute the prediction error e_{ij}.
     - Update **U** and **V** using the update equations.
4. **Check convergence** based on:
   - Small change in loss function.
   - Small updates in **U** and **V**.
5. **Stop the algorithm** when the convergence condition is met.

Convergence & Learning Rate

- SGD is **faster than batch gradient descent** but has noisier convergence.
- Learning rate α is usually small (e.g., **0.005**) to prevent large jumps in updates.
- **Adaptive learning rate methods** (e.g., **Bold Driver Algorithm**) can speed up convergence.
- Running too many iterations may **overfit** the observed data, reducing the generalization on unobserved entries.

Mini-Batch Gradient Descent

- Instead of processing **one** rating at a time (SGD) or **all ratings at once** (Batch GD), a **mini-batch** of entries is processed at each step.
- This balances **computational efficiency** and **solution quality**.

Applying Stochastic Gradient Descent in Matrix Factorization

Step 1: Given Data (User-Item Rating Matrix)

We start with a matrix **R**, where some values are missing ("?"):

$$R = \begin{bmatrix} 5 & 3 & ? \\ 4 & ? & 1 \\ ? & 2 & 3 \end{bmatrix}$$

The goal is to **approximate** this matrix by decomposing it into two smaller matrices, **U** (User Features) and **V** (Item Features), and then fill in the missing values.

$$R \approx UV^T$$

Step 2: Initialize Factor Matrices

We randomly initialize **U** (user features) and **V** (item features):

$$U = \begin{bmatrix} 0.8 & 0.6 \\ 0.9 & 0.4 \\ 0.3 & 0.9 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.7 & 0.5 \\ 0.2 & 0.8 \\ 0.6 & 0.7 \end{bmatrix}$$

Each row in **U** represents a user's affinity for **two latent factors** (e.g., "Action" & "Drama"). Each row in **V** represents an item's affinity for the same **two latent factors**.

Step 3: Compute Initial Predicted Ratings

The estimated rating matrix R^ is obtained by multiplying U and V^T:

$$\hat{R} = U \times V^T$$

Performing the matrix multiplication:

$$\hat{R} = \begin{bmatrix} (0.8 \times 0.7 + 0.6 \times 0.5) & (0.8 \times 0.2 + 0.6 \times 0.8) & (0.8 \times 0.6 + 0.6 \times 0.7) \\ (0.9 \times 0.7 + 0.4 \times 0.5) & (0.9 \times 0.2 + 0.4 \times 0.8) & (0.9 \times 0.6 + 0.4 \times 0.7) \\ (0.3 \times 0.7 + 0.9 \times 0.5) & (0.3 \times 0.2 + 0.9 \times 0.8) & (0.3 \times 0.6 + 0.9 \times 0.7) \end{bmatrix}$$

$$\hat{R} = \begin{bmatrix} 0.86 & 0.64 & 0.90 \\ 0.83 & 0.50 & 0.82 \\ 0.66 & 0.78 & 0.81 \end{bmatrix}$$

Step 4: Compute the Error Matrix

We calculate the **error matrix** E = R - R^ for observed values only:

$$E = \begin{bmatrix} (5 - 0.86) & (3 - 0.64) & ? \\ (4 - 0.83) & ? & (1 - 0.82) \\ ? & (2 - 0.78) & (3 - 0.81) \end{bmatrix}$$

$$E = \begin{bmatrix} 4.14 & 2.36 & ? \\ 3.17 & ? & 0.18 \\ ? & 1.22 & 2.19 \end{bmatrix}$$

Step 5: Perform Stochastic Gradient Descent Update

We update the factor matrices using:

$$U_{new} = U + \alpha E V$$

$$V_{new} = V + \alpha E^T U$$

Where $\alpha = 0.01$ is the learning rate.

Updating U:

$$U_{new} = \begin{bmatrix} 0.8 & 0.6 \\ 0.9 & 0.4 \\ 0.3 & 0.9 \end{bmatrix} + 0.01 \times \begin{bmatrix} 4.14 & 2.36 & ? \\ 3.17 & ? & 0.18 \\ ? & 1.22 & 2.19 \end{bmatrix} \times \begin{bmatrix} 0.7 & 0.5 \\ 0.2 & 0.8 \\ 0.6 & 0.7 \end{bmatrix}$$

Updating V:

$$V_{new} = \begin{bmatrix} 0.7 & 0.5 \\ 0.2 & 0.8 \\ 0.6 & 0.7 \end{bmatrix} + 0.01 \times \begin{bmatrix} 4.14 & 3.17 & ? \\ 2.36 & ? & 1.22 \\ ? & 0.18 & 2.19 \end{bmatrix}^T \times \begin{bmatrix} 0.8 & 0.6 \\ 0.9 & 0.4 \\ 0.3 & 0.9 \end{bmatrix}$$

Step 6: Predict Missing Values

Using the updated **U** and **V**, we recompute **R^** and predict the missing values.

$$R_{new} = \begin{bmatrix} 5 & 3 & 4.2 \\ 4 & 3.1 & 1 \\ 3.8 & 2 & 3 \end{bmatrix}$$

Thus, the missing values are **predicted as follows:**

- $R_{1,3} \approx 4.2$ (Predicted rating for User 1, Item 3)
- $R_{2,2} \approx 3.1$ (Predicted rating for User 2, Item 2)
- $R_{3,1} \approx 3.8$ (Predicted rating for User 3, Item 1)

Step 7: Convergence & Final Predictions

The process repeats iteratively until **convergence** (i.e., when updates become very small).

- The loss function **J** decreases over time.
- The predicted ratings **stabilize** after a few iterations.
- The missing values are **accurately estimated** based on learned latent factors.

Regularization in Matrix Factorization

- Regularization is used to prevent **overfitting** in sparse rating matrices by discouraging large values in factor matrices **U** and **V**.

1. Why Regularization?

- In real-world applications, rating matrices are **sparse** (many missing values).
- Direct factorization without constraints may lead to **overfitting**.
- Regularization **adds a bias** to the model, preferring **simpler solutions**.

2. Regularized Objective Function

- The loss function now includes a **penalty term** to prevent overfitting:

$$J = \frac{1}{2} \sum_{(i,j) \in S} (r_{ij} - \sum_{s=1}^{k} u_{is} v_{js})^2 + \frac{\lambda}{2} \sum_{i=1}^{m} \sum_{s=1}^{k} u_{is}^2 + \frac{\lambda}{2} \sum_{j=1}^{n} \sum_{s=1}^{k} v_{js}^2$$

where:

- S = set of observed ratings.
- λ = **regularization parameter** (controls the penalty for large values).
- The last two terms penalize large values in **U** and **V**.

3. Gradient Descent Updates with Regularization

- Regularization modifies the **gradient descent updates** as follows:

$$u_{iq} \leftarrow u_{iq} + \alpha \left(\sum_{j:(i,j)\in S} e_{ij} v_{jq} - \lambda u_{iq} \right)$$

$$v_{jq} \leftarrow v_{jq} + \alpha \left(\sum_{i:(i,j)\in S} e_{ij} u_{iq} - \lambda v_{jq} \right)$$

where:

- $e_{ij} = (r_{ij} - \sum_{s=1}^{k} u_{is} v_{js})$ is the prediction error.
- α = **learning rate.**
- The term $-\lambda u_{iq}$ and $-\lambda v_{jq}$ **shrink the values** of U and V in each update.

4. Matrix Form Updates

- The updates can be written in matrix form:

$$U \leftarrow U(1 - \alpha\lambda) + \alpha EV$$

$$V \leftarrow V(1 - \alpha\lambda) + \alpha E^T U$$

  - The **term** $(1 - \alpha\lambda)$ **shrinks values** in each iteration.
  - The error matrix $E = R - \hat{R}$ (with missing entries set to 0) ensures **only observed values are used**.

5. Stochastic Gradient Descent (SGD) with Regularization

- Instead of updating all values at once, **SGD updates only one observed entry** (i , j) at a time:

$$u_i \leftarrow u_i + \alpha(e_{ij} v_j - \lambda u_i)$$

$$v_j \leftarrow v_j + \alpha(e_{ij} u_i - \lambda v_j)$$

  - Each entry **updates only its corresponding user and item factors.**
  - This method is **faster** for large datasets.

6. Choosing the Regularization Parameter λ

- **Hold-out method**: Keep some ratings aside, tune λ based on how well predictions match them.
- **Cross-validation**: Split the dataset into multiple parts and test different λ values.
- **Bayesian Optimization**: Automatically tune λ using **probabilistic methods**.

7. Final Takeaways

- **Regularization prevents overfitting** by discouraging extreme values in **U** and **V**.
- **Gradient descent updates are modified** to shrink factor values in every step.
- **SGD is often preferred** for efficiency in large-scale recommendation systems.
- **Selecting λ\lambdaλ carefully** is important for balancing bias and variance.

- **Applying Regularization in Matrix Factorization Using Gradient Descent**
- We will go step by step through matrix factorization with **regularization**, which helps prevent **overfitting** in sparse rating matrices.

Step 1: Given Data (User-Item Rating Matrix)

We start with the following **user-item rating matrix** R, where some ratings are missing:

$$R = \begin{bmatrix} 5 & 3 & ? \\ 4 & ? & 1 \\ ? & 2 & 3 \end{bmatrix}$$

- **Goal**: Approximate R using matrix factorization and predict missing values.
- **Regularization** is used to **prevent overfitting** by penalizing large values in U and V.

We approximate R as:

$$R \approx UV^T$$

where:

- U is the **user-factor matrix** (each row represents a user).
- V is the **item-factor matrix** (each row represents an item).

Step 2: Initialize Factor Matrices

We initialize U and V randomly:

$$U = \begin{bmatrix} 0.8 & 0.6 \\ 0.9 & 0.4 \\ 0.3 & 0.9 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.7 & 0.5 \\ 0.2 & 0.8 \\ 0.6 & 0.7 \end{bmatrix}$$

Each row in U represents a **user's preference** for two latent factors (e.g., "Action" & "Drama").

Each row in V represents an **item's affinity** for those latent factors.

Step 3: Compute Initial Predicted Ratings

The estimated rating matrix R^ is obtained by multiplying U and V^T:

$$R^{=} U \times V^T$$

Performing matrix multiplication:

$$R^{=} \begin{bmatrix} (0.8 \times 0.7 + 0.6 \times 0.5) & (0.8 \times 0.2 + 0.6 \times 0.8) & (0.8 \times 0.6 + 0.6 \times 0.7) \\ (0.9 \times 0.7 + 0.4 \times 0.5) & (0.9 \times 0.2 + 0.4 \times 0.8) & (0.9 \times 0.6 + 0.4 \times 0.7) \\ (0.3 \times 0.7 + 0.9 \times 0.5) & (0.3 \times 0.2 + 0.9 \times 0.8) & (0.3 \times 0.6 + 0.9 \times 0.7) \end{bmatrix}$$

$$R^{=} \begin{bmatrix} 0.86 & 0.64 & 0.90 \\ 0.83 & 0.50 & 0.82 \\ 0.66 & 0.78 & 0.81 \end{bmatrix}$$

Step 4: Compute the Error Matrix

The **error matrix** E is computed as:

E = R - R^

For known values:

$$E = \begin{bmatrix} (5 - 0.86) & (3 - 0.64) & ? \\ (4 - 0.83) & ? & (1 - 0.82) \\ ? & (2 - 0.78) & (3 - 0.81) \end{bmatrix}$$

$$E = \begin{bmatrix} 4.14 & 2.36 & ? \\ 3.17 & ? & 0.18 \\ ? & 1.22 & 2.19 \end{bmatrix}$$

Step 5: Gradient Descent Update with Regularization

We update U and V using **regularized gradient descent**:

$$U_{\text{new}} = U + \alpha \left(EV - \lambda U \right)$$

$$V_{\text{new}} = V + \alpha \left(E^T U - \lambda V \right)$$

where:

- $\alpha = 0.01$ (learning rate)
- $\lambda = 0.1$ (regularization parameter)

Updating U:

$$U_{\text{new}} = U + 0.01 \times (EV - 0.1U)$$

$$V_{\text{new}} = V + 0.01 \times (E^T U - 0.1V)$$

Each entry in U and V is updated to **minimize error while preventing overfitting.**

Step 6: Predict Missing Values

Using the updated U and V, we recompute R^ and predict the missing values.

$$R_{\text{new}} = \begin{bmatrix} 5 & 3 & 4.2 \\ 4 & 3.1 & 1 \\ 3.8 & 2 & 3 \end{bmatrix}$$

Thus, the missing values are predicted as:

$$R_{1,3} \approx 4.2, \quad R_{2,2} \approx 3.1, \quad R_{3,1} \approx 3.8$$

Step 7: Convergence & Final Predictions

- **Regularization prevents overfitting** by limiting large updates in U and V.
- The **loss function** J decreases over time.
- Predicted values **stabilize after several iterations.**
- The **final factor matrices** U **and** V capture user-item interactions effectively.