Advanced Recommender Systems 1

Agenda

1. Introduction to Recommender Systems
    - Importance of recommender Systems
    - Types of Recommender System
2. Collaborative Filtering
    - Types of Collaborative Filtering
        - User-Based Collaborative Filtering
        - Item-Based Collaborative Filtering
3. Matrix Factorization Techniques
    - Singular Value Decomposition
    - Alternating Least Square
    - Stochastics Gradient Descent
    - Non-Negative Matrix Factorization
    - Probabilistics Matrix Factorization
4. Evaluation Metrics for Recommender Systems
    - (Precision, Recall, F1-Score)
    - (MAE, RMSE, MRR, NDCG, Coverage)

Introduction to Recommender Systems

- Recommender Systems (RS) are sophisticated algorithms designed to provide personalized item suggestions to users based on factors such as user preferences, past behavior, and item characteristics.
- They are critical in enhancing user experience by offering relevant content, improving engagement, and driving customer satisfaction. Platforms like Netflix, Amazon, Spotify, and YouTube leverage recommender systems to suggest movies, products, music, and videos tailored to individual tastes.
- By analyzing large datasets, RS can predict what users may like, leading to increased interaction and business growth.
- The key approaches include collaborative filtering, content-based filtering, and hybrid methods, making RS essential for personalization in digital services.

Importance of Recommender Systems

Recommender systems have gained immense importance across industries, providing businesses with significant advantages by enhancing user engagement, boosting sales, and improving customer satisfaction. Their primary role is to help users discover items of interest more easily, which directly influences how businesses deliver personalized experiences. Here's a detailed look at why recommender systems are crucial:

1. **Personalization of User Experience**
    - Recommender systems tailor content, products, or services based on individual user preferences and behavior, making the experience more personalized. Instead of displaying generic recommendations, they provide curated content that matches users' interests.
        - Impact: This personalization drives customer engagement, leading to increased interaction with the platform, retention, and overall user satisfaction.
        - Example: Netflix recommending movies and TV shows based on the user's viewing history and preferences ensures that viewers remain engaged for longer periods.
2. **Increased User Engagement**
    - By providing relevant suggestions, recommender systems keep users engaged and interested in the platform. Users are more likely to explore additional items when they find that the system accurately predicts their preferences.
        - Impact: Sustained engagement translates into more time spent on the platform, fostering loyalty and increasing the likelihood of future purchases or interactions.

- Example: YouTube's recommendation algorithm suggests videos that users are more likely to watch based on their browsing history, increasing video views and session durations.

3. **Boosting Sales and Revenue**
   - Recommender systems play a significant role in increasing sales, especially in e-commerce. By suggesting products that users are more likely to purchase, businesses can see a significant uptick in revenue. Recommender systems can also expose users to products they might not have considered but find useful or interesting.
     - Impact: Businesses experience increased conversion rates, cross-selling, and upselling, directly contributing to higher profits.
     - Example: Amazon's "Customers who bought this item also bought" recommendation leads to additional purchases, driving overall sales growth.
4. **Handling Information Overload**
   - The internet provides an overwhelming amount of information, products, and services. Recommender systems simplify decision-making by filtering vast datasets and presenting users with the most relevant options.
     - Impact: By reducing information overload, recommender systems allow users to make faster, more informed choices, improving the overall user experience.
     - Example: Spotify helps users navigate through millions of songs by recommending music that aligns with their tastes, reducing the challenge of manually searching for songs.
5. **Enhanced Customer Retention and Loyalty**
   - Effective recommendations increase user satisfaction, which in turn fosters long-term customer relationships. When users consistently find value in recommendations, they tend to stay loyal to the platform.
     - Impact: Increased retention and loyalty lead to higher lifetime value (LTV) of customers, which is crucial for business growth.
     - Example: On platforms like Netflix or Amazon Prime, users are more likely to renew their subscriptions when they consistently find relevant and engaging content.

Types of recommender systems

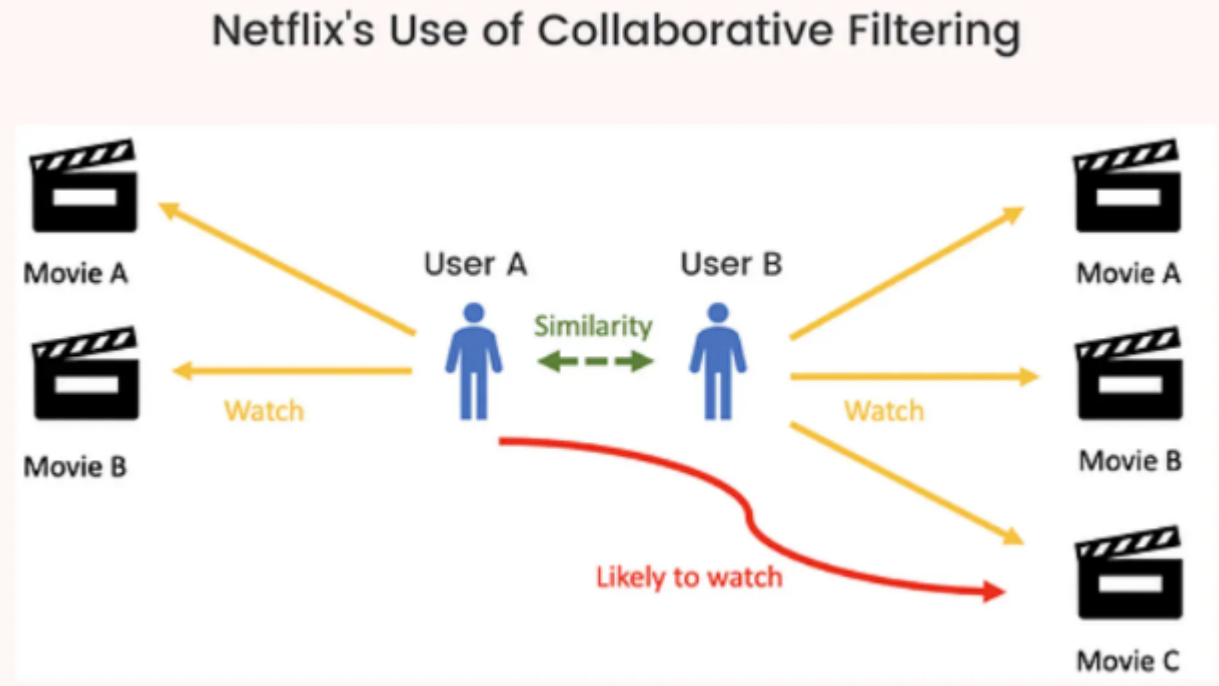
1. Content-Based Filtering
   - Recommends items similar to those a user has liked or interacted with in the past. The algorithm focuses on the attributes of items and matches them with the user's

profile.
- How it works: It calculates the similarity between items based on features (like genre, actors, or product specifications). For example, if a user watched action movies, the system will recommend more action films.

2. Collaborative Filtering
   - Based on the idea that users who agreed in the past will agree in the future. It focuses on identifying users with similar tastes.
   - How it works:
     - User-User Collaborative Filtering: Finds similar users based on their interactions and recommends items liked by those users.
     - Item-Item Collaborative Filtering: Recommends items based on similarity between items, using user ratings or interactions.
3. Hybrid Methods
   - Combines multiple recommendation approaches (content-based, collaborative filtering, etc.) to overcome the limitations of each individual method.
   - How it works: Different strategies like weighted hybrid (combining scores from different models) or switching hybrid (switching between models) are used.
4. Knowledge-Based Recommender Systems
   - Recommends items based on explicit knowledge about user preferences and item features. It requires domain-specific information and user requirements.
   - How it works: Relies on a knowledge base and uses rules or constraints to filter the items.
5. Deep Learning-Based Recommender Systems
   - Uses deep learning models (such as neural networks) to capture complex patterns in user-item interactions.
   - How it works: Neural networks, especially Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), are used to model sequential behavior or content features for better recommendation accuracy.
6. Context-Aware Recommender Systems
   - Considers the context in which recommendations are made, such as location, time, device, or user's current activity.
   - How it works: Contextual information is incorporated into traditional recommender algorithms to tailor recommendations to a user's specific situation.

Collaborative Filtering

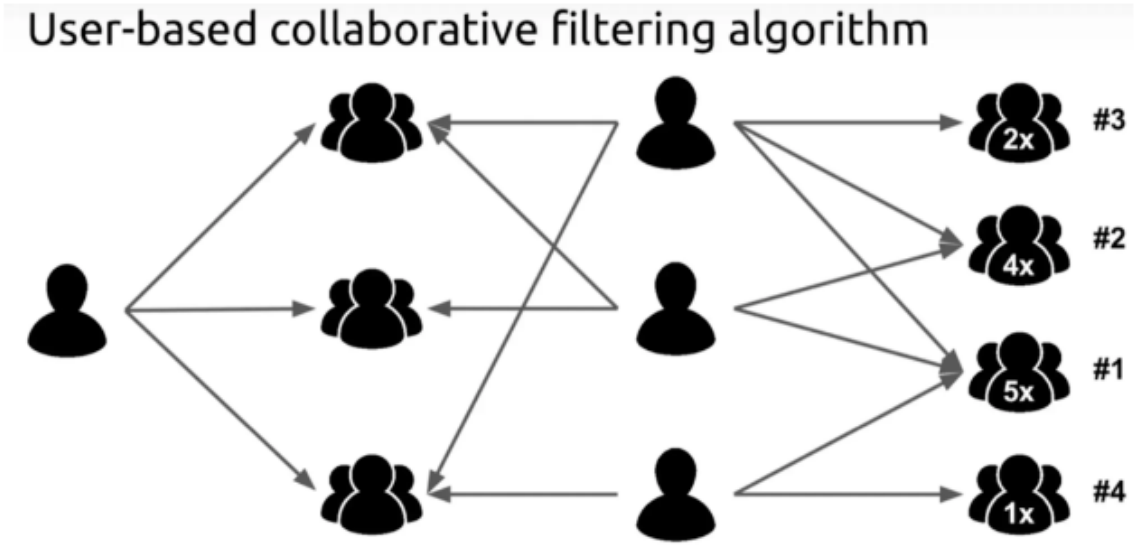
- Collaborative filtering (CF) is one of the most widely used techniques in recommender systems, relying on the idea that users who have agreed in the past (i.e., have similar preferences or behaviors) will agree again in the future.
- It aims to predict the preferences of a user based on the preferences of similar users or items. Collaborative filtering doesn't require domain knowledge about the items being recommended but leverages large amounts of user-item interaction data, such as user ratings or behaviors.
- Collaborative filtering can be broken down into two primary types: user-based collaborative filtering and item-based collaborative filtering.
- Additionally, there are matrix factorization methods that focus on reducing the dimensionality of user-item interactions to make predictions.



Types of Collaborative Filtering

User-Based Collaborative Filtering (UBCF)

- It is a popular technique in recommender systems that suggests items to users based on the preferences of similar users.
- The underlying assumption is that if two users have shown similar preferences in the past, they are likely to have similar tastes in the future.
- The goal of UBCF is to find "neighboring" users whose past behaviors (e.g., ratings, purchases) closely match the target user's behavior, and then recommend items that those neighboring users liked but the target user hasn't interacted with yet.



Steps for User-Based Collaborative Filtering (UBCF):

- Identify the Target User:
- Select the user for whom you want to generate recommendations (referred to as the target user). This user may have already interacted with some items (e.g., provided ratings, made purchases, or clicked on items), and the goal is to recommend new items that they may like.
    - Create a User-Item Matrix:
- Construct a matrix that captures the interactions between users and items. Each row represents a user, and each column represents an item. The entries in the matrix are the user's interaction values (ratings, clicks, purchases). Missing values indicate that the user has not interacted with that item yet.
- Find Similar Users (Neighbors):
    - Calculate the similarity between the target user and all other users based on their interactions with items. Common similarity metrics include:
        - Cosine Similarity: Measures the cosine of the angle between the two users' rating vectors.
        - Pearson Correlation: Measures the linear correlation between the users' ratings.
    - Example: Using cosine similarity to find how similar User B and the target user are.
- Select the Nearest Neighbors:
    - Choose the top N most similar users (neighbors) based on the calculated similarity scores. These users will be the ones whose preferences are most aligned with the target user.
- Aggregate the Neighbors’ Preferences:

    - Identify items that these similar users (neighbors) have interacted with but that the target user has not. The neighbors' ratings for these items are considered when generating recommendations.
    - Example: If User B (neighbor) has highly rated Item 2, which the target user has not rated, Item 2 becomes a candidate for recommendation.

- Generate Recommendations:
    - Recommend the top items that similar users have liked but the target user hasn’t interacted with yet. The recommendations can be made by calculating a weighted average of the neighbors' ratings for each item or simply selecting the most popular items among the neighbors.
    - Example: If neighbors like Item 2 and Item 4, recommend these items to the target user.
- Present Recommendations:
    - Show the list of recommended items to the target user, ideally ranked in order of expected preference based on the aggregated neighbor preferences.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Sample user-item interaction matrix (ratings)
R = np.array([[5, 3, 0, 1],
              [4, 0, 0, 1],
              [1, 1, 0, 5],
              [1, 0, 0, 4],
              [0, 1, 5, 4]])

# Calculate cosine similarity between users
user_similarity = cosine_similarity(R)

# Function to get recommendations for a target user
def get_user_based_recommendations(target_user_id, R, user_similarity, n_recommen
    # Get similar users
    similar_users = np.argsort(user_similarity[target_user_id])[-2:][::-1]

    # Aggregate ratings from similar users
    recommended_items = {}
    for user in similar_users:
        for item_id, rating in enumerate(R[user]):
            if rating > 0 and R[target_user_id][item_id] == 0:  # Recommend items
                if item_id not in recommended_items:
                    recommended_items[item_id] = 0
                recommended_items[item_id] += rating

    # Sort recommendations
    recommended_items = sorted(recommended_items.items(), key=lambda x: x[1], rev
    return recommended_items[:n_recommendations]

# Get recommendations for user 0
recommendations = get_user_based_recommendations(target_user_id=0, R=R, user_simi
print("User-Based Recommendations for User 0:", recommendations)
```

```
User-Based Recommendations for User 0: []
```

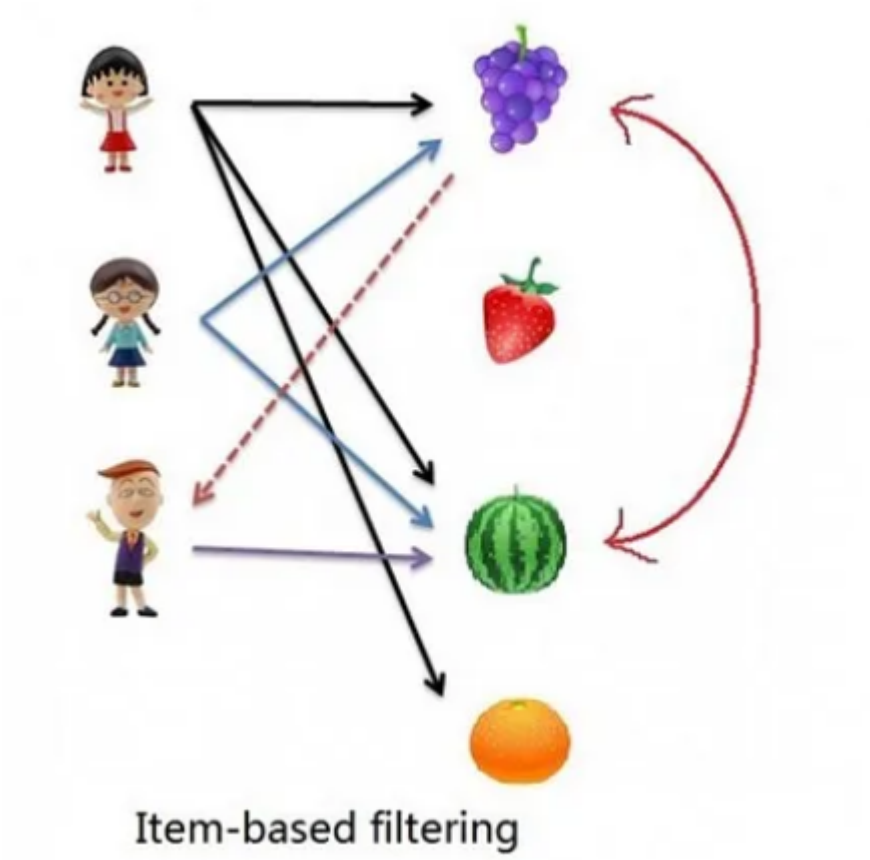
Item-Based Collaborative Filtering

- Item-Based Collaborative Filtering (IBCF) is a recommendation technique that suggests items to users by examining the relationships between items rather than between users.
- It assumes that users will prefer items similar to those they have already interacted with (liked, purchased, rated highly).
- Instead of finding similar users to make recommendations, IBCF identifies items that are similar to the items the target user has engaged with and recommends those similar items.

How It Works:

- IBCF relies on the idea that if a user likes an item, they will likely like other items that are similar to it.

- The system calculates item-to-item similarities, often using item interaction data from multiple users, and then makes recommendations based on these similarities.



Steps Involved in Item-Based Collaborative Filtering:

1. Create an Item-User Matrix:
    - The first step is to construct an item-user interaction matrix. In this matrix, rows represent items, columns represent users, and the entries indicate how much a user interacted with a particular item (e.g., ratings, purchases, views, or clicks).
2. Compute Item Similarities:
    - Calculate similarity scores between each pair of items based on user interactions. Common similarity measures include:
        - Cosine Similarity: Measures the cosine of the angle between two item vectors, treating each item's interactions with users as a vector.
        - Pearson Correlation: Measures the linear correlation between the ratings or interactions of two items.
3. Form an Item Similarity Matrix:
    - Construct a matrix of item-to-item similarities. In this matrix, rows and columns represent items, and each entry shows the similarity score between the corresponding pair of items.
4. Identify Items the Target User Has Interacted With:

- For the target user, identify the items they have already interacted with (e.g., items they have rated, purchased, or clicked on). For instance, if the target user has rated Item 1 highly, we can use that to find similar items.

5. Find Similar Items:
    - For each item the user has interacted with, find other items that are highly similar based on the item similarity matrix. These similar items will be considered candidates for recommendation.
6. Generate Recommendations:
    - Rank the similar items based on their similarity scores and make recommendations to the user. You can also use a weighted average of the ratings of similar items to provide a more accurate recommendation score.
7. Present Recommendations:
    - Finally, show the user the top recommended items that they have not interacted with, based on item similarity. This personalized list of items is intended to reflect the user's preferences based on their past interactions.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Calculate cosine similarity between items
item_similarity = cosine_similarity(R.T)

# Function to get recommendations for a target user
def get_item_based_recommendations(target_user_id, R, item_similarity, n_recommen
    user_ratings = R[target_user_id]
    recommended_items = {}

    # Get items liked by the user
    for item_id, rating in enumerate(user_ratings):
        if rating > 0:  # Only consider rated items
            # Get similar items
            for similar_item_id, similarity_score in enumerate(item_similarity[it
                if similar_item_id not in recommended_items:
                    recommended_items[similar_item_id] = 0
                # Weight the recommendation by the similarity and the user's rati
                recommended_items[similar_item_id] += similarity_score * rating

    # Remove items already rated by the user
    for item_id in range(len(user_ratings)):
        if user_ratings[item_id] > 0:
            recommended_items.pop(item_id, None)

    # Sort recommendations
    recommended_items = sorted(recommended_items.items(), key=lambda x: x[1], rev
    return recommended_items[:n_recommendations]

# Get recommendations for user 0
recommendations = get_item_based_recommendations(target_user_id=0, R=R, item_simi
print("Item-Based Recommendations for User 0:", recommendations)
```

```
Item-Based Recommendations for User 0: [(2, 1.4252896776565864)]
```

Matrix Factorization Techniques

- Matrix factorization is a robust and widely used technique in recommender systems, particularly in the context of collaborative filtering.
- This approach is essential for extracting latent factors that explain the patterns of user-item interactions observed in large datasets.
- The fundamental idea behind matrix factorization is to decompose the user-item interaction matrix into lower-dimensional matrices that capture the underlying structure of the data while preserving significant information about user preferences and item characteristics.

Key Matrix Factorization Techniques

Matrix factorization techniques are essential in building effective recommender systems. They help uncover latent features that explain user-item interactions. Here are some key matrix factorization techniques commonly used in recommender systems:

1. Singular Value Decomposition (SVD)

- Singular Value Decomposition (SVD) is a fundamental matrix factorization technique widely used in various applications, including signal processing, natural language processing, and, notably, recommender systems. It provides a way to decompose a matrix into simpler, interpretable components, enabling tasks such as dimensionality reduction, data compression, and noise reduction.
    - Mathematical Definition Given a matrix A of dimensions m×n (where m is the number of rows and n is the number of columns), SVD states that A can be decomposed into three matrices:

$A = U\Sigma V^T$

- U: An m×m orthogonal matrix whose columns are the left singular vectors of A.
- Σ: An m×n diagonal matrix containing the singular values of A on its diagonal, ordered from largest to smallest.
- V^T : An n×n orthogonal matrix whose rows are the right singular vectors of A.

Applications of SVD

- Recommender Systems: SVD is extensively used in collaborative filtering to extract latent factors that can represent user preferences and item characteristics. By reducing the dimensionality of the user-item interaction matrix, SVD helps in making predictions for unobserved interactions.
- Dimensionality Reduction: In machine learning, SVD is employed for reducing the dimensionality of datasets while preserving important information. This is particularly useful for preprocessing data before applying machine learning algorithms.

- Image Compression: SVD can be used to compress images by representing them with fewer singular values, significantly reducing the file size while maintaining a reasonable quality of the image.
- Natural Language Processing: In text mining, SVD is applied to reduce the dimensionality of term-document matrices, helping to uncover latent semantic structures in the data. This is commonly referred to as Latent Semantic Analysis (LSA).
- Noise Reduction: SVD can help filter out noise in data by reconstructing matrices with only the most significant singular values, effectively smoothing the data.

```
pip install numpy scipy scikit-learn
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3
```

```
import numpy as np
from numpy.linalg import svd

# Sample user-item interaction matrix (ratings)
R = np.array([[5, 3, 0, 1],
              [4, 0, 0, 1],
              [1, 1, 0, 5],
              [1, 0, 0, 4],
              [0, 1, 5, 4]])

# Perform SVD
U, Sigma, Vt = svd(R, full_matrices=False)

# Create a diagonal matrix for Sigma
Sigma = np.diag(Sigma)

# Reconstruct the original matrix
R_approx = np.dot(U, np.dot(Sigma, Vt))

print("Original Ratings:\n", R)
print("\nApproximated Ratings using SVD:\n", R_approx)
```

```
Original Ratings:
 [[5 3 0 1]
 [4 0 0 1]
 [1 1 0 5]
 [1 0 0 4]
 [0 1 5 4]]

Approximated Ratings using SVD:
 [[ 5.00000000e+00  3.00000000e+00  1.55140666e-15  1.00000000e+00]
 [ 4.00000000e+00 -5.41755501e-16  1.25914820e-15  1.00000000e+00]
 [ 1.00000000e+00  1.00000000e+00 -1.49309448e-15  5.00000000e+00]
```

```
[ 1.00000000e+00  1.09333219e-15 -5.34909636e-16  4.00000000e+00]
[ 3.27352245e-15  1.00000000e+00  5.00000000e+00  4.00000000e+00]]
```

2. Alternating Least Squares (ALS)

- Alternating Least Squares (ALS) is an optimization technique commonly used in collaborative filtering and recommender systems, particularly for matrix factorization tasks. It aims to learn latent factors for users and items from a user-item interaction matrix, effectively allowing for personalized recommendations. The primary advantage of ALS is its efficiency in handling large, sparse datasets, making it suitable for real-world applications such as Netflix and Spotify.
    - Mathematical Formulation:

      Given a user-item interaction matrix R, ALS decomposes it into two matrices U (user features) and V (item features) such that:

      $R \approx UV^T$

- Where:
    - U: An m×k matrix, where m is the number of users and k is the number of latent factors (features) for each user.
    - V: An n×k matrix, where n is the number of items.

The goal is to minimize the loss function:

$$L(U, V) = \sum_{(i,j)\in\mathcal{O}} (R_{ij} - U_i V_j^T)^2 + \lambda(\|U\|^2 + \|V\|^2)$$

Where:

- O is the set of observed ratings in the matrix R.
- λ is the regularization parameter to prevent overfitting.

Applications of ALS:

- Recommender Systems: Widely used in systems that provide personalized recommendations for movies, music, and products based on user interactions.
- Content Recommendation: Employed in content delivery platforms to suggest articles, videos, and music tracks tailored to user preferences.
- Collaborative Filtering: Used in collaborative filtering algorithms to identify similar users and items based on latent features learned from the data.

```
from scipy.sparse import csr_matrix
from sklearn.decomposition import NMF

# Convert to sparse matrix
R_sparse = csr_matrix(R)

# Fit the ALS model
model = NMF(n_components=2, init='random', random_state=0)
W = model.fit_transform(R_sparse)
H = model.components_

# Reconstruct the original matrix
R_approx = np.dot(W, H)

print("Original Ratings:\n", R)
print("\nApproximated Ratings using ALS:\n", R_approx)
```

```
Original Ratings:
 [[5 3 0 1]
 [4 0 0 1]
 [1 1 0 5]
 [1 0 0 4]
 [0 1 5 4]]

Approximated Ratings using ALS:
 [[5.25583751 1.99314304 0.         1.45510614]
 [3.50429883 1.32891643 0.         0.97018348]
 [1.31291255 0.9441558  1.94957474 3.94614513]
 [0.98126695 0.72179626 1.52760301 3.0788861 ]
 [0.         0.65008539 2.83998144 5.21892451]]
```

3. Stochastic Gradient Descent (SGD)

- Stochastic Gradient Descent (SGD) is a widely-used optimization algorithm in machine learning and deep learning for minimizing a loss function.
- Unlike traditional gradient descent, which uses the entire dataset to compute the gradient, SGD updates the model parameters using a single training example (or a small subset) at a time.
- This approach allows for faster convergence, particularly when dealing with large datasets.
- The main goal of any optimization algorithm is to minimize the loss function L(w), where w represents the model parameters.
- In SGD, instead of computing the gradient of the loss function over the entire dataset, the algorithm updates the parameters based on the gradient computed from a single or a mini-batch of examples.
- This method introduces randomness into the optimization process, which can help escape local minima and improve generalization.

Mathematical Formulation

- The basic update rule for SGD can be expressed as follows:

$$w_{t+1} = w_t - \eta \nabla L(w_t; x_i, y_i)$$

Where:

- w_t : The model parameters at iteration t.
- η: The learning rate, which controls the step size of each update.
- $\nabla L(w_t ; x_i , y_i)$: The gradient of the loss function with respect to the parameters w_t , calculated using a single training example (x_i , y_i).

Applications of SGD

- Linear Regression: SGD can be applied to optimize the coefficients of a linear regression model efficiently, especially with large datasets.
- Neural Networks: SGD is the backbone of training neural networks, where it updates the weights of the model based on the computed gradients during backpropagation.
- Support Vector Machines: SGD can be used to optimize the hinge loss function for SVMs, enabling the training of large-scale classifiers.
- Reinforcement Learning: In some reinforcement learning algorithms, SGD is utilized to update policies based on the received rewards.

```
import numpy as np

class SGDRecommender:
    def __init__(self, n_users, n_items, n_factors, alpha=0.01, beta=0.01, n_iter
        self.n_users = n_users
        self.n_items = n_items
        self.n_factors = n_factors
        self.alpha = alpha
        self.beta = beta
        self.n_iter = n_iter
        self.user_factors = np.random.normal(scale=1./self.n_factors, size=(self.
        self.item_factors = np.random.normal(scale=1./self.n_factors, size=(self.

    def fit(self, R):
        for _ in range(self.n_iter):
            for i in range(self.n_users):
                for j in range(self.n_items):
                    if R[i][j] > 0:  # Only update for non-zero entries
                        error = R[i][j] - np.dot(self.user_factors[i], self.item_
                        self.user_factors[i] += self.alpha * (error * self.item_f
                        self.item_factors[j] += self.alpha * (error * self.user_f

    def predict(self):
        return np.dot(self.user_factors, self.item_factors.T)


n_users, n_items = R.shape
sgd_recommender = SGDRecommender(n_users, n_items, n_factors=2)
sgd_recommender.fit(R)

# Get predicted ratings
R_approx = sgd_recommender.predict()

print("Original Ratings:\n", R)
print("\nApproximated Ratings using SGD:\n", R_approx)
```

```
Original Ratings:
 [[5 3 0 1]
 [4 0 0 1]
 [1 1 0 5]
 [1 0 0 4]
 [0 1 5 4]]

Approximated Ratings using SGD:
 [[4.97702448 2.98273346 4.24944753 1.00231811]
 [3.98216893 2.40270773 3.61090253 1.00083456]
 [1.00364041 0.98986029 5.91502209 4.97224419]
 [0.99841728 0.9063534  4.86376467 3.98399307]
 [1.17570019 1.01030258 4.98522139 3.99148907]]
```

4. Non-Negative Matrix Factorization (NMF)

- Non-Negative Matrix Factorization (NMF) is a mathematical technique used for factorizing a non-negative matrix into two lower-dimensional non-negative matrices.
- It is particularly useful in the fields of data mining, machine learning, and signal processing for extracting latent features from data, especially when the data consists of non-negative values, such as images, text, and audio.
- NMF aims to decompose a given non-negative matrix V into two non-negative matrices W and H such that:

$$V \approx WH$$

Where:

- V: An m×n non-negative matrix (e.g., user-item interactions, images).
- W: An m×k non-negative matrix representing the features (or basis).
- H: A k×n non-negative matrix representing the weights or coefficients for the features.

The number of features k is typically much smaller than m and n, allowing for dimensionality reduction and feature extraction.

Mathematical Formulation

- The objective of NMF is to minimize the reconstruction error between the original matrix V and the product of W and H. This is typically done using a cost function such as:

$$L(W, H) = \|V - WH\|_F^2$$

Where

- $\|\cdot\|_F$ denotes the Frobenius norm, which calculates the difference between the original and reconstructed matrices.

Additionally, NMF enforces the non-negativity constraints on W and H:

$$W \geq 0 \quad \text{and} \quad H \geq 0$$

Applications of NMF

- Image Processing: NMF is used for image representation and compression by decomposing images into non-negative components.
- Text Mining: In natural language processing, NMF can be used for topic modeling to identify latent topics in a collection of documents.

- Recommendation Systems: NMF can help in collaborative filtering to identify hidden features that influence user preferences.
- Biological Data Analysis: NMF is utilized in genomics and proteomics to identify patterns in high-dimensional biological datasets.

```
from sklearn.decomposition import NMF

# Fit the NMF model
model = NMF(n_components=2, init='random', random_state=0)
W = model.fit_transform(R)
H = model.components_

# Reconstruct the original matrix
R_approx = np.dot(W, H)

print("Original Ratings:\n", R)
print("\nApproximated Ratings using NMF:\n", R_approx)
```

```
Original Ratings:
 [[5 3 0 1]
 [4 0 0 1]
 [1 1 0 5]
 [1 0 0 4]
 [0 1 5 4]]

Approximated Ratings using NMF:
 [[5.25583751 1.99314304 0.         1.45510614]
 [3.50429883 1.32891643 0.         0.97018348]
 [1.31291255 0.9441558  1.94957474 3.94614513]
 [0.98126695 0.72179626 1.52760301 3.0788861 ]
 [0.         0.65008539 2.83998144 5.21892451]]
```

5. Probabilistic Matrix Factorization (PMF)

- Probabilistic Matrix Factorization (PMF) is a statistical approach to matrix factorization, which is widely used in collaborative filtering for recommender systems.
- Unlike traditional matrix factorization techniques that rely on deterministic optimization methods, PMF incorporates probabilistic models, allowing it to capture uncertainty and make predictions about missing data in a principled way.
- In PMF, we aim to model a user-item interaction matrix V V with missing entries, representing user preferences for items. The matrix is decomposed into two latent feature matrices: a user matrix U and an item matrix M, such that:

  $V \approx UM^T$

Where:

- V: An m×n matrix, with m users and n items.
- U: An m×k matrix representing latent features for users.

- M: An n×k matrix representing latent features for items.
- k: The number of latent factors (features).

The goal is to learn the parameters U and M that minimize the difference between the observed entries in V and the predicted values based on the latent factors.

Applications of PMF

- Recommender Systems: PMF is widely used in various recommender systems to suggest products, movies, or music based on user preferences.
- Collaborative Filtering: It is a fundamental technique in collaborative filtering methods, enabling personalized recommendations by uncovering hidden relationships in user-item interactions.
- Data Imputation: PMF can be applied to fill in missing values in datasets, providing a means to estimate unobserved ratings or preferences.
- Anomaly Detection: PMF can help identify outliers in user behavior by analyzing deviations from the expected ratings.

```
import numpy as np

class PMF:
    def __init__(self, n_users, n_items, n_factors, alpha=0.01, beta=0.02, n_iter
        self.n_users = n_users
        self.n_items = n_items
        self.n_factors = n_factors
        self.alpha = alpha
        self.beta = beta
        self.n_iter = n_iter
        self.user_factors = np.random.normal(0, 0.1, (n_users, n_factors))
        self.item_factors = np.random.normal(0, 0.1, (n_items, n_factors))

    def fit(self, R):
        for _ in range(self.n_iter):
            for i in range(self.n_users):
                for j in range(self.n_items):
                    if R[i][j] > 0:  # Only update for non-zero entries
                        error = R[i][j] - np.dot(self.user_factors[i], self.item_
                        self.user_factors[i] += self.alpha * (error * self.item_f
                        self.item_factors[j] += self.alpha * (error * self.user_f

    def predict(self):
        return np.dot(self.user_factors, self.item_factors.T)

n_users, n_items = R.shape
pmf = PMF(n_users, n_items, n_factors=2)
pmf.fit(R)

# Get predicted ratings
R_approx = pmf.predict()

print("Original Ratings:\n", R)
print("\nApproximated Ratings using PMF:\n", R_approx)
```

```
Original Ratings:
 [[5 3 0 1]
 [4 0 0 1]
 [1 1 0 5]
 [1 0 0 4]
 [0 1 5 4]]

Approximated Ratings using PMF:
 [[4.95414966 2.96473471 4.8690594  1.00451513]
 [3.96373642 2.38755806 4.09515893 1.00139799]
 [1.00872648 0.97583193 5.77523886 4.94519596]
 [0.99589414 0.89167083 4.77955071 3.96838999]
 [1.22478336 1.02602243 4.9707851  3.98138093]]
```

Applications of Matrix Factorization

- Movie Recommendations: Platforms like Netflix use matrix factorization techniques to recommend movies based on user viewing history and preferences.

- E-commerce: Online retailers such as Amazon utilize matrix factorization to suggest products to customers based on their purchase history and similarities with other users.
- Music Recommendations: Music streaming services like Spotify leverage matrix factorization to create playlists based on user listening behavior and similarities with other listeners.
- Social Media: Social networking sites use matrix factorization for friend recommendations, suggesting connections based on shared interests and interactions.

Evaluation Metrics for Recommender Systems

- Evaluating recommender systems is essential for measuring how well they fulfill their goals, whether it's recommending items, predicting ratings, or aligning with user preferences.
- Common evaluation metrics include precision and recall, which assess the relevance of recommendations, and F1-score, which balances precision and recall.
- Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) measure the accuracy of predicted ratings by comparing them to actual ratings. Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain (nDCG) evaluate the ranking quality of recommendations, ensuring top items are more relevant.
- For personalized recommendations, diversity and novelty metrics ensure that the system introduces users to new and varied items.
- Ultimately, the choice of metrics depends on the recommender system's objectives, such as accuracy, relevance, or user satisfaction.

Precision

- Precision measures the proportion of recommended items that are relevant to the user, focusing on the accuracy of the recommendations.
- It calculates the ratio of relevant items retrieved to the total number of items recommended.
- A high precision score indicates that most of the recommended items are useful to the user, but it does not consider whether all relevant items were retrieved.
- Precision is particularly important when the system needs to ensure that only highly relevant items are recommended, minimizing irrelevant suggestions.
- However, it may need to be balanced with recall, which measures the completeness of relevant items retrieved.

$$\text{Precision} = \frac{\text{Number of relevant items recommended}}{\text{Total number of recommended items}}$$

Purpose: Precision is useful when the goal is to recommend a few highly relevant items rather than aiming for comprehensive coverage. It focuses on recommendation accuracy.

Use Case: In scenarios where presenting relevant items (e.g., in the top-5 or top-10) is more important than covering a broader set of recommendations, like in movie recommendations.

```
pip install numpy scikit-learn
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3
```

```
from sklearn.metrics import precision_score

# Sample data: true positives (1 for relevant, 0 for not)
y_true = [1, 0, 1, 1, 0, 0, 1]
y_pred = [1, 0, 0, 1, 0, 0, 1]  # Predicted relevant items

# Calculate precision
precision = precision_score(y_true, y_pred)
print(f"Precision: {precision:.2f}")
```

```
Precision: 1.00
```

Recall

- Recall measures the proportion of relevant items that were successfully recommended out of the total relevant items available in the dataset.
- It evaluates how well the system retrieves all the useful items for the user.
- A high recall means that the system is able to recommend most of the relevant items, even if some non-relevant items are also included.
- While recall emphasizes completeness, it can result in recommending more irrelevant items if not balanced with precision.
- In recommender systems, optimizing recall is important when the goal is to provide a comprehensive set of recommendations, ensuring that users don't miss out on relevant content.

$$\text{Recall} = \frac{\text{Number of relevant items recommended}}{\text{Total number of relevant items in the data}}$$

Purpose: Recall is important when the goal is to cover all possible relevant items, ensuring that the user sees everything of interest.

Use Case: Recall is critical in domains where missing out on relevant items is costly, such as in e-commerce when ensuring users see products they're interested in.

```
from sklearn.metrics import recall_score

# Calculate recall
recall = recall_score(y_true, y_pred)
print(f"Recall: {recall:.2f}")
```

```
Recall: 0.75
```

F1-Score

- The F1-Score combines both precision and recall into a single metric by calculating their harmonic mean, providing a balanced measure of a model's performance.
- It is particularly useful when there is an uneven distribution between relevant and non-relevant items, as it ensures neither precision nor recall is favored disproportionately.
- A high F1-Score indicates that the system has a good balance between returning relevant recommendations (precision) and retrieving all relevant items (recall).
- It is especially important in cases where both precision and recall are critical for the effectiveness of the recommender system.
- The F1-Score ranges from 0 to 1, with 1 indicating perfect precision and recall.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Purpose: The F1-Score is used when there is a need to balance the accuracy of recommendations (precision) with the coverage of relevant items (recall).

Use Case: It is useful when neither precision nor recall can be sacrificed, especially in hybrid recommendation systems that prioritize both relevance and diversity.

```
from sklearn.metrics import f1_score

# Calculate F1-Score
f1 = f1_score(y_true, y_pred)
print(f"F1-Score: {f1:.2f}")
```

```
F1-Score: 0.86
```

Mean Absolute Error (MAE)

- Mean Absolute Error (MAE) measures the average magnitude of errors between the predicted ratings and the actual ratings, regardless of whether the error is positive or negative.
- It calculates the absolute differences between predicted and true ratings, then averages these differences across all predictions.
- MAE provides a clear sense of how far off the predictions are from the actual values, making it easy to interpret.
- A lower MAE indicates more accurate predictions, meaning the system's predicted ratings are closer to the actual user ratings.
- It is a straightforward and commonly used metric in evaluating recommender systems for rating prediction tasks.

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |r_i - \hat{r}_i|$$

Where r_i is the actual rating, $\hat{r}_i$ is the predicted rating, and N is the total number of predictions.

Purpose: MAE is useful for understanding the overall accuracy of the predicted ratings in comparison to the actual user ratings.

Use Case: It is commonly used in rating prediction tasks, such as when predicting user ratings for movies or products on a scale (e.g., 1 to 5).

```
from sklearn.metrics import mean_absolute_error

# Sample actual and predicted ratings
actual_ratings = [4, 3, 5, 1, 2]
predicted_ratings = [4.5, 2.5, 4, 2, 1]

# Calculate MAE
mae = mean_absolute_error(actual_ratings, predicted_ratings)
print(f"Mean Absolute Error (MAE): {mae:.2f}")
```

```
Mean Absolute Error (MAE): 0.80
```

Root Mean Square Error (RMSE)

- Root Mean Squared Error (RMSE) measures the square root of the average squared differences between the predicted ratings and the actual ratings.
- By squaring the errors before averaging, RMSE gives more weight to larger errors, making it sensitive to outliers or significant deviations in predictions.
- This means that higher RMSE values indicate that the system's predictions are further off from actual ratings, particularly in cases where large errors occur.
- RMSE is commonly used in recommender systems to evaluate prediction accuracy, as it highlights not only the magnitude of errors but also their impact, making it suitable for scenarios where avoiding large errors is crucial.
- A lower RMSE value signifies more accurate predictions.

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(r_i - \hat{r}_i)^2}$$

Purpose: RMSE penalizes large prediction errors more than MAE, making it useful in scenarios where avoiding large mistakes is important.

Use Case: RMSE is often used in rating-based recommendation systems where minimizing large prediction errors is crucial, such as in platforms like Netflix.

```
from sklearn.metrics import mean_squared_error
import numpy as np

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(actual_ratings, predicted_ratings))
print(f"Root Mean Square Error (RMSE): {rmse:.2f}")
```

```
Root Mean Square Error (RMSE): 0.84
```

Mean Reciprocal Rank (MRR)

- Mean Reciprocal Rank (MRR) is a metric used to evaluate the ranking quality of a recommender system.
- It calculates the average of the reciprocal of the rank at which the first relevant item appears in the list of recommendations for each user.
- In simpler terms, for each user, the system looks at the rank of the first relevant recommendation and takes the reciprocal (e.g., if the relevant item is ranked 1st, the reciprocal is 1; if ranked 3rd, the reciprocal is 1/3).
- The MRR is then computed by averaging these reciprocals across all users.
- A higher MRR indicates that relevant items are ranked closer to the top of the recommendation list, providing users with quick access to useful suggestions.

$$\text{MRR} = \frac{1}{|U|} \sum_{u=1}^{|U|} \frac{1}{\text{rank}_u}$$

Where rank_u is the position of the first relevant item in the recommendation list for user u.

Purpose: MRR is used to evaluate the ranking quality of recommender systems, particularly when only the first relevant item is of interest.

Use Case: MRR is common in applications where showing the most relevant item at the top of the recommendation list is crucial, like search engines.

```
def mean_reciprocal_rank(relevant_items, recommended_items):
    mrr = 0.0
    for user_relevant, user_recommend in zip(relevant_items, recommended_items):
        for rank, item in enumerate(user_recommend, start=1):
            if item in user_relevant:
```