

# ML methods for potential customer prediction

Alexandru-Flavius Huc

May 25, 2025

# Overview of the Project

- ▶ Train ML models for recommender systems using explicit movie ratings provided by users.
- ▶ Explain each ML models trained
- ▶ Evaluate how accurately the algorithm can predict ratings for movies the users have already seen and rated.

# ML methods: Collaborative Filtering

- ▶ **User-Based Collaborative Filtering**

- ▶ K-Nearest Neighbors with user similarity using:
  - ▶ Cosine similarity
  - ▶ Pearson correlation

- ▶ **Item-Based Collaborative Filtering**

- ▶ K-Nearest Neighbors with item similarity using:
  - ▶ Cosine similarity
  - ▶ Pearson correlation

# ML methods: Matrix Factorization Techniques

- ▶ Singular Value Decomposition (SVD)
- ▶ Singular Value Decomposition++ (SVD++)
- ▶ Non-negative Matrix Factorization (NMF)
- ▶ Alternating Least Squares (ALS)
- ▶ Probabilistic Matrix Factorization (PMF)

# ML methods: Content-Based Filtering

- ▶ Logistic Regression
- ▶ Decision Trees
- ▶ Naive Bayes
- ▶ Support Vector Machines (SVM)
- ▶ K-Nearest Neighbors with feature vectors
- ▶ Term Frequency - Inverse Document Frequency + Cosine similarity

# Contents

## Overview of the Project

## Collaborative Filtering: Introduction

- User-to-User Collaborative Filtering

  - PureUserKNN

- Item-to-Item Collaborative Filtering

  - PureItemKNN

## Matrix Factorization: Introduction

- PureSVD

- PureSVDpp

- PureNMF

- PureALS

- PurePMF

## Content-Based Filtering: Introduction

- PureDecisionTree

- PureSVM

- PureNaiveBayes

- PureTFIDF

# Collaborative Filtering: Introduction

- ▶ **Recommender Systems:**

- ▶ *Non-personalized*: Same recommendations for all users
- ▶ *Personalized*: Customized to individual user preferences

- ▶ **Collaborative Filtering:**

- ▶ A personalized recommendation technique
- ▶ Uses interaction data from multiple users
- ▶ Predicts user ratings
- ▶ Does not require content analysis of items

# Types of Collaborative Filtering

## User-to-User Collaborative Filtering

- ▶ Based on user similarity
- ▶ Core assumption: Similar users like similar items
- ▶ Process:
  1. Find users with similar rating patterns
  2. Use their ratings to predict missing ratings

## Item-to-Item Collaborative Filtering

- ▶ Based on item similarity
- ▶ Core assumption: Users rate similar items similarly
- ▶ Process:
  1. Find items similar to those the user rated
  2. Predict ratings based on similar items
- ▶ Useful when user preferences are abstract



# User-to-User Collaborative Filtering

- ▶ **Core Principle:** Users who rated items similarly in the past will rate other items similarly
- ▶ **Implementation Process:**
  1. Create a user-item rating matrix
  2. Calculate similarity between target user and all other users
  3. Identify the most similar users (neighbors)
  4. Predict ratings using weighted average of neighbor ratings
- ▶ **Advantages:**
  - ▶ Intuitive approach
  - ▶ Works across diverse domains without content analysis
  - ▶ Can discover unexpected recommendations
- ▶ **Challenges:**
  - ▶ Computationally expensive with many users
  - ▶ Cold-start problem for new users
  - ▶ Sparsity in the rating matrix

# PureUserKNN Class

- ▶ A pure Python implementation of user-based collaborative filtering
  - ▶ Uses K-Nearest Neighbors approach to predict user ratings for items
  - ▶ **Core assumption:** Users with similar taste in the past will have similar preferences in the future
  - ▶ **Goal:** Predict unknown ratings based on patterns in existing ratings
  - ▶ Does not rely on item content features - only uses the rating matrix
- 
- ▶ *PureUserKNN examines user rating patterns to identify similar users, then leverages these similarities to make personalized predictions*

# Class Structure: Key Components

## ▶ Initialization Parameters

- ▶ **k**: Number of nearest neighbors to consider
- ▶ **similarity metric**: 'pearson' (default) or 'cosine'

## ▶ Key Data Structures

- ▶ User-user similarity matrix
- ▶ User rating dictionaries
- ▶ User mean ratings
- ▶ Inverted index (item  $\rightarrow$  users who rated it)

## ▶ Primary Methods

- ▶ **fit**: Train the model
- ▶ **predict**: Generate rating predictions
- ▶ **similarity computations**: Calculate user similarity

# Method: fit() - Training the Model

## 1. Data Organization

- ▶ Process input (user, item, rating) tuples
- ▶ Organize ratings by user
- ▶ Create inverted index for efficient lookups
- ▶ Calculate global mean rating
- ▶ Calculate per-user mean ratings

## 2. User-User Similarity Matrix Computation

- ▶ Create an  $n \times n$  matrix where  $n$  is the number of users
- ▶ For each pair of users ( $u, v$ ):
  - ▶ Find common items they both rated
  - ▶ Compute similarity based on these co-rated items
  - ▶ Store similarity values symmetrically
- ▶ Uses either Pearson correlation or cosine similarity

## 3. Optimization

- ▶ Only compute each similarity once:  $\text{sim}(u, v) = \text{sim}(v, u)$
- ▶ Track progress for large datasets

# Methods: Similarity Computation

- ▶ **\_compute\_similarity**: Gateway method that selects the appropriate metric
  - ▶ Identifies common items rated by both users
  - ▶ Ensures at least 2 common items for meaningful similarity
  - ▶ Delegates to specific similarity implementations
- ▶ **\_compute\_pearson\_similarity**:
  - ▶ Measures correlation between user rating patterns
  - ▶ Accounts for different rating scales via mean-centering
  - ▶ Formula:

$$\text{sim}(u, v) = \frac{\sum_{i \in I_{uv}} (r_{ui} - \bar{r}_u)(r_{vi} - \bar{r}_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \bar{r}_u)^2 \sum_{i \in I_{uv}} (r_{vi} - \bar{r}_v)^2}}$$

- ▶ **\_compute\_cosine\_similarity**:
  - ▶ Measures angle between user rating vectors
  - ▶ Simpler but doesn't adjust for rating scale differences
  - ▶ Formula:

$$\text{sim}(u, v) = \frac{\sum_{i \in I_{uv}} r_{ui} r_{vi}}{\sqrt{\sum_{i \in I_{uv}} r_{ui}^2 \sum_{i \in I_{uv}} r_{vi}^2}}$$

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify user exists in training data
- ▶ Check if user has already rated this item

## 2. Neighbor Selection

- ▶ Find users who have rated the target item
- ▶ Calculate similarity between target user and these users
- ▶ Filter to include only positively correlated users
- ▶ Select top-k most similar users (neighbors)

## 3. Rating Calculation

- ▶ Use weighted average based on similarity scores
- ▶ Apply mean-centering to normalize different user rating scales
- ▶ Formula:  $\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N} \text{sim}(u, v) \cdot (r_{vi} - \bar{r}_v)}{\sum_{v \in N} \text{sim}(u, v)}$
- ▶ Clip final prediction to valid rating range (1-5)

## 4. Fallback Strategies

- ▶ Return user's mean rating if no neighbors found
- ▶ Return global mean for unknown users

# Item-to-Item Collaborative Filtering

- ▶ **Core Principle:** Similar items tend to receive similar ratings from the same user
- ▶ **Implementation Process:**
  1. Create a user-item rating matrix
  2. Calculate similarity between items based on co-ratings
  3. Identify the most similar items to those the user has rated (neighbors)
  4. Predict ratings using weighted average of ratings given to similar items
- ▶ **Advantages:**
  - ▶ More scalable than user-based approach
  - ▶ More stable (item relationships change less frequently)
  - ▶ Better performance with sparse data
  - ▶ Can be pre-computed offline
- ▶ **Challenges:**
  - ▶ Cold-start problem for new items
  - ▶ Limited to recommending similar items (less diversity)
  - ▶ Requires sufficient co-ratings for accurate similarity
  - ▶ May struggle with niche user preferences

# What is PureItemKNN?

- ▶ A pure Python implementation of item-based collaborative filtering
- ▶ Uses K-Nearest Neighbors approach to predict ratings based on item similarities
- ▶ Core assumption: Similar items will receive similar ratings from the same user
- ▶ Goal: Predict how a user would rate an item based on their ratings of similar items
- ▶ Does not require user demographic information - only analyzes the rating matrix
- ▶ *PureItemKNN examines patterns in how items are rated across users, then leverages item similarities to make personalized predictions*



# Class Structure: Key Components

## ▶ Initialization Parameters

- ▶ **k**: Number of nearest neighbor items to consider
- ▶ **similarity metric**: 'pearson' (default) or 'cosine'

## ▶ Key Data Structures

- ▶ Item-item similarity matrix
- ▶ Item rating dictionaries
- ▶ Item mean ratings
- ▶ ID mapping dictionaries

## ▶ Primary Methods

- ▶ **fit**: Train the model
- ▶ **predict**: Generate rating predictions
- ▶ **similarity computations**: Calculate item similarity

# Method: fit() - Training the Model

## 1. Data Organization

- ▶ Process input (user, item, rating) tuples
- ▶ Organize ratings by item rather than by user
- ▶ Create item ID mappings for matrix operations
- ▶ Calculate global mean rating
- ▶ Calculate per-item mean ratings

## 2. Item-Item Similarity Matrix Computation

- ▶ Create an  $m \times m$  matrix where  $m$  is the number of items
- ▶ For each pair of items  $(i, j)$ :
  - ▶ Find common users who rated both items
  - ▶ Compute similarity based on these co-ratings
  - ▶ Store similarity values symmetrically
- ▶ Uses either Pearson correlation or cosine similarity

## 3. Optimization

- ▶ Only compute each similarity once:  $\text{sim}(i, j) = \text{sim}(j, i)$
- ▶ Track progress during computation (especially important for large item catalogs)

# Methods: Similarity Computation

- ▶ **\_compute\_similarity:** Gateway method that selects the appropriate metric
  - ▶ Identifies common users who rated both items
  - ▶ Ensures at least 2 common users for meaningful similarity
  - ▶ Delegates to specific similarity implementations
- ▶ **\_compute\_pearson\_similarity:**
  - ▶ Measures correlation between item rating patterns
  - ▶ Accounts for different item popularity via mean-centering
  - ▶ Formula:

$$\text{sim}(i, j) = \frac{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)^2 \sum_{u \in U_{ij}} (r_{uj} - \bar{r}_j)^2}}$$

- ▶ **\_compute\_cosine\_similarity:**
  - ▶ Measures angle between item rating vectors
  - ▶ Simpler but doesn't adjust for item popularity differences
  - ▶ Formula:

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

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify target item exists in training data
- ▶ Retrieve item index for similarity lookup

## 2. Neighbor Selection

- ▶ Find items the target user has already rated
- ▶ Calculate similarity between target item and these items
- ▶ Filter to include only positively correlated items
- ▶ Select top-k most similar items (neighbors)

## 3. Rating Calculation

- ▶ Use weighted average based on similarity scores
- ▶ Apply mean-centering to normalize different item popularity
- ▶ Formula:  $\hat{r}_{ui} = \bar{r}_i + \frac{\sum_{j \in N} \text{sim}(i,j) \cdot (r_{uj} - \bar{r}_j)}{\sum_{j \in N} \text{sim}(i,j)}$
- ▶ Clip final prediction to valid rating range (1-5)

## 4. Fallback Strategies

- ▶ Return item's mean rating if no similar items found
- ▶ Return null for unknown items (handled by adapter class)

# Matrix Factorization: Introduction

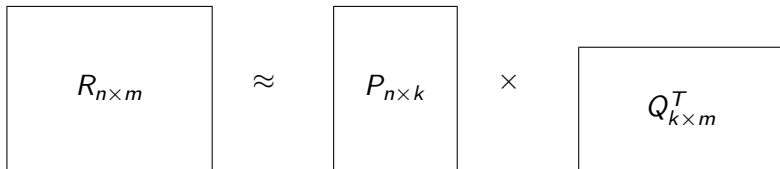
- ▶ A technique to break down a large user-item rating matrix into smaller matrices
  - ▶ Core idea: Represent both users and items in a lower-dimensional "latent factor" space
  - ▶ These latent factors capture underlying patterns in user preferences
  - ▶ Popular in recommender systems for its accuracy and scalability
- 
- ▶ *"Matrix factorization models map both users and items to a joint latent factor space, such that user-item interactions are modeled as inner products in that space."*

# Matrix Factorization: Formula

$$R \approx P \times Q^T$$

Where:

- ▶  $R$  is the original user-item rating matrix (users  $\times$  items)
- ▶  $P$  is the user factors matrix (users  $\times$  factors)
- ▶  $Q$  is the item factors matrix (items  $\times$  factors)
- ▶  $k$  is the number of latent factors (typically 10-100)


$$R_{n \times m} \approx P_{n \times k} \times Q_{k \times m}^T$$

# Matrix Factorization: Making Predictions

## Basic Prediction Formula:

$$\hat{r}_{ui} = p_u^T q_i = \sum_{f=1}^k p_{uf} \cdot q_{if}$$

Where:

- ▶  $\hat{r}_{ui}$  is the predicted rating of user  $u$  for item  $i$
- ▶  $p_u$  is the latent factor vector for user  $u$  (row of  $P$ )
- ▶  $q_i$  is the latent factor vector for item  $i$  (row of  $Q$ )

## Enhanced Model with Biases:

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$$

Adding:

- ▶  $\mu$ : Global average rating
- ▶  $b_u$ : User bias (how much user  $u$  tends to give higher/lower ratings)
- ▶  $b_i$ : Item bias (how much item  $i$  tends to receive higher/lower ratings)

# Matrix Factorization: Learning Process

## Optimization Objective:

$$\min_{P,Q,b} \sum_{(u,i) \in K} (r_{ui} - \hat{r}_{ui})^2 + \lambda(||P||^2 + ||Q||^2 + ||b||^2)$$

## Common Solution Approaches:

- ▶ **Stochastic Gradient Descent (SGD):**
  - ▶ Iteratively update factors based on prediction errors
  - ▶ Simple to implement, works well with large datasets
- ▶ **Alternating Least Squares (ALS):**
  - ▶ Fix one factor matrix and solve for the other, then alternate
  - ▶ Easily parallelizable
- ▶ **Probabilistic Methods:**
  - ▶ Treat factorization as a probabilistic model
  - ▶ Better handles uncertainty in sparse data

The term  $\lambda(||P||^2 + ||Q||^2 + ||b||^2)$  is regularization to prevent overfitting



# What is PureSVD?

- ▶ A pure Python implementation of Singular Value Decomposition for matrix factorization
  - ▶ Uses stochastic gradient descent (SGD) to learn latent factors
  - ▶ Core assumption: User preferences and item attributes can be represented in a shared low-dimensional latent space
  - ▶ Goal: Decompose the user-item rating matrix into user and item factor matrices
  - ▶ Includes biases to account for user and item rating tendencies
- 
- ▶ *PureSVD learns compact representations of users and items in a latent factor space, allowing it to capture complex preference patterns and make accurate predictions*

# Class Structure: Key Components

## ▶ Initialization Parameters

- ▶ **n\_factors**: Number of latent factors (default=100)
- ▶ **n\_epochs**: Number of iterations for SGD (default=20)
- ▶ **lr**: Learning rate for gradient descent (default=0.005)
- ▶ **reg**: Regularization term to prevent overfitting (default=0.02)

## ▶ Key Data Structures

- ▶ User factors matrix ( $\text{users} \times \text{factors}$ )
- ▶ Item factors matrix ( $\text{items} \times \text{factors}$ )
- ▶ User biases dictionary
- ▶ Item biases dictionary
- ▶ Global mean rating
- ▶ ID mapping dictionaries

## ▶ Primary Methods

- ▶ **fit**: Train the model using SGD
- ▶ **predict**: Generate rating predictions
- ▶ **get\_user\_factors/get\_item\_factors**: Access latent factors

# Method: fit() - Training the Model

## 1. Data Organization

- ▶ Process input (user, item, rating) tuples
- ▶ Create ID mappings for users and items
- ▶ Calculate global mean rating

## 2. Parameter Initialization

- ▶ Initialize user and item factor matrices with small random values
- ▶ Initialize user and item biases to zero

## 3. SGD Training Process

- ▶ For each epoch:
  - ▶ Shuffle the data for better convergence
  - ▶ For each rating:
    - ▶ Compute prediction error:  $error = r_{ui} - \hat{r}_{ui}$
    - ▶ Update user and item biases
    - ▶ Update user and item factors
    - ▶ Track RMSE progress
    - ▶ Reduce learning rate over time (adaptive learning)

## 4. Regularization

- ▶ Apply regularization to biases and factors to prevent overfitting

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify user and item exist in training data
- ▶ Return global mean for cold-start cases (new users or items)

## 2. Rating Calculation

- ▶ Retrieve user and item indices
- ▶ Compute prediction using the learned model:

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$$

where:

- ▶  $\mu$  is the global mean rating
- ▶  $b_u$  is the user bias
- ▶  $b_i$  is the item bias
- ▶  $p_u$  is the user factor vector
- ▶  $q_i$  is the item factor vector
- ▶ Clip prediction to valid rating range (1-5)

## 3. Additional Methods

- ▶ **get\_user\_factors**: Retrieve latent factors for a specific user
- ▶ **get\_item\_factors**: Retrieve latent factors for a specific item

# The Machine Learning Algorithm: SVD with SGD

## SVD Principles

- ▶ Matrix factorization technique
- ▶ Reduces dimensionality while preserving structure
- ▶ Learns latent factors representing underlying patterns
- ▶ Models both user preferences and item characteristics

## Model Formulation

- ▶ Predicts rating as:

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$$

- ▶ Minimizes squared error:

$$\min_{p,q,b} \sum_{(u,i)} (r_{ui} - \hat{r}_{ui})^2 + \lambda(\|p\|^2 + \|q\|^2 + \|b\|^2)$$

# The Machine Learning Algorithm: SVD with SGD

## Training Process (SGD)

1. Initialize model parameters
2. For each epoch:
  - ▶ Shuffle training data
  - ▶ For each rating  $(u, i, r_{ui})$ :
    - ▶ Compute error:  $e_{ui} = r_{ui} - \hat{r}_{ui}$
    - ▶ Update biases:

$$b_u \leftarrow b_u + \gamma(e_{ui} - \lambda b_u)$$

$$b_i \leftarrow b_i + \gamma(e_{ui} - \lambda b_i)$$

- ▶ Update factors:

$$p_u \leftarrow p_u + \gamma(e_{ui}q_i - \lambda p_u)$$

$$q_i \leftarrow q_i + \gamma(e_{ui}p_u - \lambda q_i)$$

- ▶ Reduce learning rate  $\gamma$

# What is PureSVDpp?

- ▶ A pure Python implementation of SVD++ (SVD Plus Plus), an enhanced matrix factorization algorithm
- ▶ Extends standard SVD by incorporating implicit feedback (which items users have rated)
- ▶ Core innovation: Creates a richer user representation using both explicit and implicit feedback
- ▶ Goal: Improve prediction accuracy by considering not just rating values but also rating patterns
- ▶ Includes both explicit preferences (ratings) and implicit preferences (interactions)
- ▶ *PureSVDpp enhances user representation by combining explicit ratings with implicit feedback signals, leading to more nuanced and accurate predictions*

# Class Structure: Key Components

## ► Initialization Parameters

- **n\_factors**: Number of latent factors (default=100)
- **n\_epochs**: Number of iterations for SGD (default=20)
- **lr**: Learning rate for gradient descent (default=0.005)
- **reg**: Regularization term to prevent overfitting (default=0.02)
- **implicit\_weight**: Weight for implicit feedback influence (default=0.1)

## ► Key Data Structures

- User factors matrix ( $\text{users} \times \text{factors}$ )
- Item factors matrix ( $\text{items} \times \text{factors}$ )
- Item implicit factors matrix ( $\text{items} \times \text{factors}$ ) - unique to SVD++
- User and item biases dictionaries
- User-rated items mapping (for implicit feedback)
- Normalized user rating counts (for proper weighting)

## ► Primary Methods

- **fit**: Train the model using enhanced SGD
- **predict**: Generate rating predictions with implicit components
- **get\_user\_factors**: Access combined explicit and implicit factors



# Method: fit() - Training the Enhanced Model

## 1. Standard Data Processing

- ▶ Process input (user, item, rating) tuples
- ▶ Create ID mappings for users and items
- ▶ Calculate global mean rating

## 2. Implicit Feedback Processing

- ▶ Build user-item interaction data (which items each user has rated)
- ▶ Pre-calculate normalization factors:  $\frac{1}{\sqrt{|I_u|}}$  for each user

# Method: fit() - Training the Enhanced Model

## 3. Parameter Initialization

- ▶ Initialize user factors, item factors with small random values
- ▶ Initialize item implicit factors (new in SVD++)
- ▶ Initialize user and item biases to zero

## 4. Enhanced SGD Training Process

- ▶ For each epoch:
  - ▶ Shuffle the data for better convergence
  - ▶ For each rating:
    - ▶ Compute implicit feedback sum for the user
    - ▶ Make prediction with implicit component
    - ▶ Update biases and explicit factors
    - ▶ Update implicit item factors for all items user has rated
  - ▶ Track RMSE progress
  - ▶ Reduce learning rate over time

# Method: predict() - Enhanced Rating Prediction

## 1. Initial Checks

- ▶ Verify user and item exist in training data
- ▶ Return global mean for cold-start cases

## 2. Implicit Feedback Calculation

- ▶ Calculate sum of implicit item factors for all items the user has rated
- ▶ Normalize by  $\frac{1}{\sqrt{|I_u|}}$  (where  $|I_u|$  is the number of items rated by user  $u$ )

# Method: predict() - Enhanced Rating Prediction

## 3. Enhanced Rating Calculation

- ▶ Compute prediction using the enhanced model:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left( p_u + w \cdot \frac{1}{\sqrt{|I_u|}} \sum_{j \in I_u} y_j \right)$$

where:

- ▶  $\mu$  is the global mean rating
  - ▶  $b_u$  is the user bias
  - ▶  $b_i$  is the item bias
  - ▶  $p_u$  is the explicit user factor vector
  - ▶  $q_i$  is the item factor vector
  - ▶  $y_j$  are the implicit item factors
  - ▶  $w$  is the implicit weight parameter
  - ▶  $I_u$  is the set of items rated by user  $u$
- ▶ Clip prediction to valid rating range (1-5)

# The Machine Learning Algorithm: SVD++ with SGD

- ▶ Extension of standard matrix factorization
- ▶ Enhances user representation with implicit feedback
- ▶ Models both explicit preferences (what users rate) and implicit preferences (which items users rate)
- ▶ Captures more nuanced patterns in user behavior
- ▶ Predicts rating with implicit feedback component:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left( p_u + w \cdot \frac{1}{\sqrt{|I_u|}} \sum_{j \in I_u} y_j \right)$$

- ▶ Minimizes squared error with additional regularization:

$$\min_{p, q, y, b} \sum_{(u, i)} (r_{ui} - \hat{r}_{ui})^2 + \lambda (\|p\|^2 + \|q\|^2 + \|y\|^2 + \|b\|^2)$$

# Training Process: SVD++ with SGD

1. Initialize model parameters including implicit item factors
2. For each epoch:

- ▶ Shuffle training data

- ▶ For each rating  $(u, i, r_{ui})$ :

- ▶ Compute implicit factor sum:  $impl_u = \frac{w}{\sqrt{|I_u|}} \sum_{j \in I_u} y_j$

- ▶ Predict:  $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T (p_u + impl_u)$

- ▶ Compute error:  $e_{ui} = r_{ui} - \hat{r}_{ui}$

- ▶ Update biases:

$$b_u \leftarrow b_u + \gamma(e_{ui} - \lambda b_u)$$

$$b_i \leftarrow b_i + \gamma(e_{ui} - \lambda b_i)$$

- ▶ Update explicit factors:

$$p_u \leftarrow p_u + \gamma(e_{ui} q_i - \lambda p_u)$$

$$q_i \leftarrow q_i + \gamma(e_{ui} (p_u + impl_u) - \lambda q_i)$$

- ▶ Update implicit factors for all  $j \in I_u$ :

$$y_j \leftarrow y_j + \gamma(e_{ui} \cdot \frac{w}{\sqrt{|I_u|}} \cdot q_i - \lambda y_j)$$

# What is PureNMF?

- ▶ A pure Python implementation of Non-negative Matrix Factorization (NMF)
- ▶ Matrix factorization technique with the constraint that all factors must be non-negative
- ▶ Core principle: Decompose rating matrix  $R$  into non-negative matrices  $P$  and  $Q$  where  $R \approx P \cdot Q^T$
- ▶ Goal: Learn interpretable, parts-based representations of users and items
- ▶ Enforces non-negativity to create more meaningful latent factors
- ▶ *PureNMF generates additive, parts-based representations where latent factors can be interpreted as features or topics, leading to more intuitive recommendation explanations*

# Class Structure: Key Components

## ► Initialization Parameters

- **n\_factors**: Number of latent factors (default=15)
- **n\_epochs**: Number of iterations for SGD (default=50)
- **lr**: Learning rate for gradient descent (default=0.01)
- **reg**: Regularization term to prevent overfitting (default=0.02)
- **beta**: Special parameter for non-negativity constraint (default=0.02)

## ► Key Data Structures

- User factors matrix - non-negative ( $\text{users} \times \text{factors}$ )
- Item factors matrix - non-negative ( $\text{items} \times \text{factors}$ )
- Normalization parameters (min rating and rating range)
- ID mapping dictionaries
- Global mean rating

## ► Primary Methods

- **fit**: Train the model using constrained SGD
- **predict**: Generate rating predictions
- **get\_user\_factors/get\_item\_factors**: Access learned factors



# Method: fit() - Training the NMF Model

## 1. Data Processing

- ▶ Process input (user, item, rating) tuples
- ▶ Create ID mappings for users and items
- ▶ Calculate global mean rating

## 2. Rating Normalization (NMF-specific)

- ▶ Scale ratings to  $[0,1]$  range for better NMF performance
- ▶ Store scaling parameters for later denormalization

# Method: `fit()` - Training the NMF Model

## 3. Parameter Initialization

- ▶ Initialize user and item factors with small positive random values (0.01-0.1)
- ▶ Starting with positive values ensures non-negativity constraints are satisfied

## 4. Constrained SGD Training Process

- ▶ For each epoch:
  - ▶ Shuffle the data for better convergence
  - ▶ For each rating:
    - ▶ Make dot product prediction
    - ▶ Calculate error on normalized scale
    - ▶ Update user and item factors with special non-negativity constraints
  - ▶ Track RMSE progress
  - ▶ Reduce learning rate over time (adaptive learning)

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify user and item exist in training data
- ▶ Return global mean for cold-start cases

## 2. Normalized Prediction

- ▶ Retrieve user and item latent factors
- ▶ Compute dot product between factors:  $\text{norm\_pred} = p_u^T q_i$
- ▶ Clip normalized prediction to  $[0,1]$  range

## 3. Denormalization

- ▶ Convert from normalized  $[0,1]$  scale back to original rating scale:

$$\text{prediction} = \text{norm\_pred} \times \text{rating\_range} + \text{min\_rating}$$

- ▶ Clip final prediction to valid rating range (1-5)

## 4. Additional Methods

- ▶ **get\_user\_factors**: Access latent factors for a specific user
- ▶ **get\_item\_factors**: Access latent factors for a specific item

# The Machine Learning Algorithm: Non-negative Matrix Factorization

## NMF Principles

- ▶ Matrix factorization technique where all elements must be non-negative
- ▶ Learns additive components rather than arbitrary features
- ▶ Results in more interpretable latent factors
- ▶ Each factor can be viewed as a feature or topic
- ▶ Creates parts-based representations (vs. holistic representations in SVD)

# The Machine Learning Algorithm: Non-negative Matrix Factorization

## Model Formulation

- Factors user-item matrix into non-negative components:

$$R \approx P \times Q^T \quad \text{subject to } P, Q \geq 0$$

- Optimization objective:

$$\min_{P, Q \geq 0} \sum_{(u,i)} (r_{ui} - p_u^T q_i)^2 + \lambda(\|P\|^2 + \|Q\|^2)$$

- Non-negativity constraint adds interpretability but complicates optimization

# Training Process: NMF with Constrained SGD

## Training Process Steps

1. Initialize factors with small positive random values (0.01-0.1)
2. For each epoch:

- ▶ Shuffle training data

- ▶ For each rating  $(u, i, r_{ui})$ :

- ▶ Calculate normalized prediction:  $\hat{r}_{ui} = p_u^T q_i$

- ▶ Compute error:  $e_{ui} = r_{ui} - \hat{r}_{ui}$

- ▶ Update user factors with non-negativity constraint:

$$p_u^{(f)} \leftarrow p_u^{(f)} + \gamma(e_{ui}q_i^{(f)} - \lambda p_u^{(f)})$$

If  $p_u^{(f)} < 0$ , then apply soft constraint with  $\beta$

$$p_u^{(f)} \leftarrow \max(0, p_u^{(f)}) \text{ (hard constraint)}$$

- ▶ Update item factors with non-negativity constraint:

$$q_i^{(f)} \leftarrow q_i^{(f)} + \gamma(e_{ui}p_u^{(f)} - \lambda q_i^{(f)})$$

If  $q_i^{(f)} < 0$ , then apply soft constraint with  $\beta$

$$q_i^{(f)} \leftarrow \max(0, q_i^{(f)}) \text{ (hard constraint)}$$

- ▶ Reduce learning rate:  $\gamma \leftarrow 0.95 \times \gamma$

# What is PureALS?

- ▶ A pure Python implementation of Alternating Least Squares (ALS) for matrix factorization
- ▶ Matrix factorization technique that alternates between solving for user factors and item factors
- ▶ Core principle: Decompose rating matrix  $R$  into matrices  $P$  and  $Q$  where  $R \approx P \cdot Q^T$
- ▶ Goal: Learn latent factors that best explain observed ratings while handling implicit feedback
- ▶ Particularly effective for implicit feedback datasets and systems with sparse ratings
  
- ▶ *PureALS provides an efficient approach to matrix factorization by alternately solving for one set of parameters while keeping the other fixed, enabling easier parallelization and faster convergence*

# Class Structure: Key Components

## ► Initialization Parameters

- **n\_factors**: Number of latent factors (default=20)
- **n\_epochs**: Number of iterations for ALS (default=15)
- **reg**: Regularization term to prevent overfitting (default=0.1)
- **confidence\_scaling**: Scaling factor for implicit confidence values (default=40)

## ► Key Data Structures

- User factors matrix (users  $\times$  factors)
- Item factors matrix (items  $\times$  factors)
- User-item mappings with confidence values
- Item-user mappings with confidence values
- Rating normalization parameters
- ID mapping dictionaries

## ► Primary Methods

- **fit**: Train the model using alternating least squares
- **predict**: Generate rating predictions
- **\_compute\_rmse**: Evaluate model performance during training
- **get\_user\_factors/get\_item\_factors**: Access learned factors



# Method: fit() - Training the ALS Model

## 1. Data Processing

- ▶ Process input (user, item, rating) tuples
- ▶ Create ID mappings for users and items
- ▶ Calculate global mean rating

## 2. Parameter Initialization

- ▶ Initialize user and item factors with small random values
- ▶ Build optimized user-item and item-user mappings for efficient updates

## 3. Confidence Calculation

- ▶ Normalize ratings to  $[0, 1]$  range
- ▶ Convert ratings to confidence values:  $c_{ui} = 1 + \alpha \cdot r_{ui}$
- ▶ Higher ratings result in higher confidence values

# Method: `fit()` - The Alternating Least Squares Process

## ALS Training Process

For each epoch:

- ▶ **Step 1:** Fix item factors, solve for each user factor
  - ▶ For each user  $u$ :
    - ▶ Build the system  $(Y^T C_u Y + \lambda I) \cdot x_u = Y^T C_u p(u)$
    - ▶ Solve for user factor vector  $x_u$  using linear algebra
- ▶ **Step 2:** Fix user factors, solve for each item factor
  - ▶ For each item  $i$ :
    - ▶ Build the system  $(X^T C_i X + \lambda I) \cdot y_i = X^T C_i p(i)$
    - ▶ Solve for item factor vector  $y_i$  using linear algebra
- ▶ Calculate and report RMSE on training data periodically

**Key insight:** Each step has a closed-form solution that can be computed directly, avoiding the need for gradient-based optimization

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify user and item exist in training data
- ▶ Return global mean for cold-start cases

## 2. Raw Prediction Calculation

- ▶ Retrieve user and item latent factors
- ▶ Compute dot product between factors:  $\text{prediction} = p_u^T q_i$

## 3. Confidence to Rating Conversion

- ▶ Convert from confidence space back to normalized rating:

$$\text{scaled\_prediction} = \frac{\text{prediction} - 1.0}{\text{confidence\_scaling}}$$

- ▶ Then convert to original rating scale:

$$\text{final\_prediction} = \text{scaled\_prediction} \times \text{rating\_range} + \text{min\_rating}$$

- ▶ Clip prediction to valid rating range (typically 1-5)

## 4. Helper Methods

- ▶ **get\_user\_factors**: Access latent factors for a specific user
- ▶ **get\_item\_factors**: Access latent factors for a specific item

# The Machine Learning Algorithm: Alternating Least Squares

## ALS Core Principles

- ▶ Matrix factorization technique that alternates between optimizing two sets of parameters
- ▶ Well-suited for implicit feedback data (views, clicks, purchases)
- ▶ Can be highly parallelized (each user/item update is independent)
- ▶ Directly solves for optimal factors given current estimates of the other factors
- ▶ Converges more quickly than SGD for certain problems

# The Machine Learning Algorithm: Alternating Least Squares

## Model Formulation

- ▶ With confidence values for implicit feedback:

$$\min_{X,Y} \sum_{(u,i)} c_{ui} (r_{ui} - x_u^T y_i)^2 + \lambda (\|X\|^2 + \|Y\|^2)$$

where:

- ▶  $c_{ui}$  is the confidence for user  $u$ 's preference for item  $i$
- ▶  $r_{ui}$  is the observed rating (or binary interaction)
- ▶  $x_u$  and  $y_i$  are user and item factors
- ▶  $\lambda$  is the regularization parameter

# Training Process: The ALS Algorithm

## Alternating Optimization Steps

1. Initialize factor matrices with small random values
2. For each epoch:
  - ▶ **Fix  $Y$ , solve for each row of  $X$ :**

$$x_u = (Y^T C_u Y + \lambda I)^{-1} Y^T C_u p(u)$$

where:

- ▶  $Y$  is the item factor matrix
  - ▶  $C_u$  is a diagonal matrix with confidence values for user  $u$
  - ▶  $p(u)$  is the vector of user  $u$ 's preferences
  - ▶  $I$  is the identity matrix
- ▶ **Fix  $X$ , solve for each row of  $Y$ :**

$$y_i = (X^T C_i X + \lambda I)^{-1} X^T C_i p(i)$$

where:

- ▶  $X$  is the user factor matrix
- ▶  $C_i$  is a diagonal matrix with confidence values for item  $i$
- ▶  $p(i)$  is the vector of preferences for item  $i$

# Advantages and Limitations of ALS

## Advantages

### ▶ **Mathematical Properties**

- ▶ Guaranteed to converge to at least a local optimum
- ▶ Each update has a closed-form solution
- ▶ Easier to parallelize than SGD methods

### ▶ **Performance Characteristics**

- ▶ Works well with implicit feedback (e.g., views, clicks)
- ▶ Handles large, sparse datasets efficiently
- ▶ Often converges in fewer iterations than gradient methods

## Limitations

- ▶ Computationally expensive for very large datasets
- ▶ Requires matrix inversions or solving linear systems
- ▶ Less effective when factors should be constrained (e.g., non-negative)
- ▶ Cold-start problems for new users and items

# What is PurePMF?

- ▶ A pure Python implementation of Probabilistic Matrix Factorization (PMF)
- ▶ Matrix factorization technique with a principled probabilistic interpretation
- ▶ Core principle: Models ratings as  $R = P \cdot Q^T + \epsilon$  where  $\epsilon$  is Gaussian noise
- ▶ Goal: Learn latent factors that maximize the posterior probability of observed ratings
- ▶ Places Gaussian priors on user and item latent factors for regularization
  
- ▶ *PurePMF extends standard matrix factorization with a probabilistic framework, offering better theoretical justification and uncertainty quantification while maintaining computational efficiency*



# Class Structure: Key Components

## ► Initialization Parameters

- **n\_factors**: Number of latent factors (default=10)
- **n\_epochs**: Number of iterations for SGD (default=50)
- **lr**: Learning rate for gradient descent (default=0.005)
- **user\_reg/item\_reg**: Separate regularization terms (default=0.1)
- **lr\_decay**: Learning rate decay factor (default=0.95)
- **min\_lr**: Minimum learning rate (default=0.001)
- **init\_std**: Standard deviation for initial factor values (default=0.1)

# Class Structure: Key Components

## ▶ Key Data Structures

- ▶ User factors matrix ( $\text{users} \times \text{factors}$ )
- ▶ Item factors matrix ( $\text{items} \times \text{factors}$ )
- ▶ Rating normalization parameters
- ▶ ID mapping dictionaries
- ▶ Global mean rating

## ▶ Primary Methods

- ▶ **fit**: Train the model using stochastic gradient descent
- ▶ **predict**: Generate rating predictions
- ▶ **get\_user/item\_factors**: Access learned factors
- ▶ **calculate\_posterior\_variance**: Compute uncertainty measures

# Method: `fit()` - Training the PMF Model

## 1. Data Processing

- ▶ Process input (user, item, rating) tuples
- ▶ Create ID mappings for users and items
- ▶ Calculate global mean rating and rating bounds

## 2. Parameter Initialization

- ▶ Initialize user factors from Gaussian distribution:  $\mathcal{N}(0, \sigma^2)$
- ▶ Initialize item factors from Gaussian distribution:  $\mathcal{N}(0, \sigma^2)$
- ▶ This aligns with PMF's probabilistic interpretation

## 3. Rating Normalization

- ▶ Scale ratings to  $[0, 1]$  range for numerical stability
- ▶ Store scaling parameters for later denormalization

# Method: fit() - The SGD Process with Momentum

## SGD Training Process

- ▶ Initialize momentum vectors for user and item factors
- ▶ For each epoch:
  - ▶ Shuffle training data for better convergence
  - ▶ For each rating  $(u, i, r_{ui})$ :
    - ▶ Compute prediction:  $\hat{r}_{ui} = \mathbf{p}_u^T \mathbf{q}_i$
    - ▶ Calculate error:  $e_{ui} = r_{ui} - \hat{r}_{ui}$
    - ▶ Compute gradients with separate regularization terms:

$$\nabla_{\mathbf{p}_u} = -e_{ui} \cdot \mathbf{q}_i + \lambda_u \cdot \mathbf{p}_u$$

$$\nabla_{\mathbf{q}_i} = -e_{ui} \cdot \mathbf{p}_u + \lambda_i \cdot \mathbf{q}_i$$

- ▶ Update factors using momentum:

$$\mathbf{v}_{\mathbf{p}_u} = \alpha \cdot \mathbf{v}_{\mathbf{p}_u} - \eta \cdot \nabla_{\mathbf{p}_u}$$

$$\mathbf{p}_u = \mathbf{p}_u + \mathbf{v}_{\mathbf{p}_u}$$

$$\mathbf{v}_{\mathbf{q}_i} = \alpha \cdot \mathbf{v}_{\mathbf{q}_i} - \eta \cdot \nabla_{\mathbf{q}_i}$$

$$\mathbf{q}_i = \mathbf{q}_i + \mathbf{v}_{\mathbf{q}_i}$$

- ▶ Track and report RMSE progress
- ▶ Decay learning rate:  $\eta = \max(\eta_{\min}, \eta \cdot \gamma)$

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify user and item exist in training data
- ▶ Return global mean for cold-start cases

## 2. Normalized Prediction

- ▶ Retrieve user and item latent factors
- ▶ Compute dot product between factors:  $\text{norm\_pred} = p_u^T q_i$
- ▶ Clip normalized prediction to  $[0,1]$  range

## 3. Denormalization

- ▶ Convert from normalized  $[0,1]$  scale back to original rating scale:

$$\text{prediction} = \text{norm\_pred} \times \text{rating\_range} + \text{min\_rating}$$

- ▶ Clip final prediction to valid rating range (typically 1-5)

# Additional Methods

- ▶ **get\_user\_factors / get\_item\_factors**
  - ▶ Return learned latent factors for specific users or items
  - ▶ Useful for similarity calculations and feature analysis
- ▶ **calculate\_posterior\_variance**
  - ▶ Unique to probabilistic models like PMF
  - ▶ Provides uncertainty estimates for learned factors
  - ▶ In PMF, posterior variance is related to regularization parameters:

$$\sigma_{user}^2 = \frac{1}{\lambda_{user}}$$
$$\sigma_{item}^2 = \frac{1}{\lambda_{item}}$$

- ▶ Lower regularization implies higher variance (more uncertainty)
- ▶ Can be used for confidence intervals on predictions

# The Machine Learning Algorithm: Probabilistic Matrix Factorization

- ▶ Extends matrix factorization with a probabilistic interpretation
- ▶ Models observed ratings as samples from a probability distribution
- ▶ Assumes ratings follow Gaussian distribution centered on the dot product of factors:

$$p(R|P, Q, \sigma^2) = \prod_{(u,i) \in K} \mathcal{N}(r_{ui} | p_u^T q_i, \sigma^2)$$

- ▶ Places Gaussian priors on user and item factors:

$$p(P | \sigma_P^2) = \prod_u \mathcal{N}(p_u | 0, \sigma_P^2 I)$$

$$p(Q | \sigma_Q^2) = \prod_i \mathcal{N}(q_i | 0, \sigma_Q^2 I)$$

- ▶ Seeks maximum a posteriori (MAP) estimation of factors
- ▶ Provides theoretical basis for the squared error loss and regularization terms

# Training Process: The PMF Algorithm

## MAP Estimation through SGD

1. Initialize factor matrices with small random values from a Gaussian distribution
2. Maximize log posterior probability, which is equivalent to minimizing:

$$\min_{P,Q} \sum_{(u,i) \in K} (r_{ui} - p_u^T q_i)^2 + \lambda_P \sum_u \|p_u\|^2 + \lambda_Q \sum_i \|q_i\|^2$$

where:

- ▶  $\lambda_P = \frac{\sigma^2}{\sigma_P^2}$  is the user regularization parameter
  - ▶  $\lambda_Q = \frac{\sigma^2}{\sigma_Q^2}$  is the item regularization parameter
3. For each iteration of SGD:
    - ▶ Sample a rating  $(u, i, r_{ui})$
    - ▶ Update factors in the direction of steepest gradient descent
    - ▶ Apply momentum to accelerate convergence and avoid local minima
    - ▶ Adaptively reduce learning rate to ensure convergence



# Advantages and Limitations of PMF

## Advantages

- ▶ **Theoretical Foundation**
  - ▶ Provides a principled probabilistic interpretation
  - ▶ Naturally justifies the use of L2 regularization
  - ▶ Enables uncertainty quantification in predictions
- ▶ **Performance Characteristics**
  - ▶ Good performance on sparse datasets
  - ▶ Scalable through stochastic gradient methods
  - ▶ Can incorporate additional priors for domain knowledge

## Limitations

- ▶ Assumes Gaussian distributions which may not always be appropriate
- ▶ SGD training can be sensitive to hyperparameters
- ▶ Still suffers from cold-start problems for new users/items
- ▶ Full Bayesian treatment would require more complex inference methods

# What is Content-Based Filtering?

## Core Concept:

- ▶ Recommends items similar to what users liked in the past
- ▶ Uses item features/attributes rather than user ratings
- ▶ Each item represented as a feature vector
- ▶ Each user has a preference profile based on liked items
- ▶ "If you liked this movie with these actors and genre, you'll like similar ones"

## Examples of Item Features:

- ▶ **Movies:** genre, director, actors, keywords
- ▶ **News:** topics, authors, entities mentioned
- ▶ **Products:** category, brand, price range, specifications

*Unlike collaborative filtering, content-based approaches can recommend items with no ratings and don't suffer from the cold-start problem for new items*

# What is PureDecisionTree?

- ▶ A pure Python implementation of Decision Trees for movie rating prediction
- ▶ Content-based filtering approach using movie features (genres and release year)
- ▶ Core principle: Creates regression trees to predict ratings based on movie attributes
- ▶ Goal: Build personalized decision trees for each user to predict ratings for unseen movies
- ▶ Uses movie content features rather than collaborative patterns between users
- ▶ *PureDecisionTree bridges content-based filtering with supervised learning, creating a separate decision model for each user based on their historical preferences for movies with specific features*

# Class Structure: Key Components

## ► Initialization Parameters

- **max\_depth**: Maximum depth of the decision tree (default=8)
- **min\_samples\_split**: Minimum samples required to split a node (default=5)

## ► Key Data Structures

- **user\_trees**: Dictionary mapping user IDs to their personalized decision trees
- **movie\_features**: Dictionary mapping movie IDs to feature vectors
- **user\_avg\_ratings**: Average rating for each user (for fallback)
- **global\_mean**: Global mean rating (for ultimate fallback)
- **Node**: Helper class representing nodes in the decision tree

## ► Primary Methods

- **fit**: Train personalized trees for users with sufficient ratings
- **predict**: Generate rating predictions using appropriate tree
- **\_build\_tree**: Recursively create the tree structure
- **\_best\_split**: Find optimal feature and threshold for splitting

# Method: fit() - Training the Decision Tree Models

## 1. Feature Extraction

- ▶ Load movie content features (genres and release year)
- ▶ Create feature vectors for each movie:
  - ▶ Binary encoding for each genre (1 if present, 0 if not)
  - ▶ Normalized release year (divided by a constant)
- ▶ Example: [0, 1, 1, 0, ..., 0, 0.995] for an Action/Adventure movie from 2019

## 2. Rating Organization

- ▶ Group ratings by user
- ▶ Calculate global mean rating and per-user average ratings
- ▶ Filter to include only users with sufficient ratings (e.g., 20+)

## 3. User-Specific Tree Creation

- ▶ For each eligible user:
  - ▶ Extract feature vectors and ratings for movies they've rated
  - ▶ Build a personalized decision tree using these examples

# Method: `_build_tree()` - The Tree Construction Process

## Recursive Tree Building

- ▶ Create a node for the current partition of data
- ▶ Check stopping criteria:
  - ▶ Maximum depth reached
  - ▶ Minimum samples for splitting not met
  - ▶ All ratings in the node are identical
- ▶ If stopping criteria met: create a leaf node with average rating
- ▶ Otherwise:
  - ▶ Find best feature and threshold for splitting using variance reduction
  - ▶ Split data into left and right partitions
  - ▶ Recursively build subtrees for each partition
  - ▶ Store split information (feature index and threshold) in the node

**Key insight:** Each split aims to maximize the reduction in variance of ratings, creating more homogeneous groups that lead to more accurate predictions

## Method: `_best_split()` - Finding Optimal Splits

### ► **Splitting Criterion:** Variance Reduction

- Goal: Find feature and threshold that maximizes variance reduction
- Variance reduction = parent variance - weighted average of child variances
- Mathematical formulation:

$$\Delta Var = Var(S) - \left( \frac{|S_L|}{|S|} Var(S_L) + \frac{|S_R|}{|S|} Var(S_R) \right)$$

where:

- $S$  is the set of examples at the current node
- $S_L$  and  $S_R$  are the left and right subsets after splitting
- $Var(S)$  is the variance of ratings in set  $S$

### ► **Split Selection Process**

- For each feature:
  - For genre features (binary): use threshold of 0.5
  - For year feature (continuous): try multiple thresholds
  - Evaluate variance reduction for each potential split
  - Keep track of best split found so far
- Return feature index and threshold that yield maximum variance reduction

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify movie exists in feature database
- ▶ If not, prediction is impossible

## 2. Tree-Based Prediction

- ▶ Check if user has a personalized decision tree
- ▶ If yes:
  - ▶ Retrieve movie feature vector
  - ▶ Traverse tree from root to leaf by comparing feature values to node thresholds
  - ▶ Return the value stored at the reached leaf node
- ▶ Clip prediction to valid rating range (1-5)

## 3. Fallback Strategies

- ▶ If user has no tree but has ratings: return user's average rating
- ▶ If user is completely new: return global mean rating



# The Machine Learning Algorithm: Decision Trees for Regression

- ▶ Tree-based supervised learning method for predicting continuous values
- ▶ Creates a hierarchical structure of if-then rules based on features
- ▶ Divides feature space into regions with similar target values
- ▶ Predictions made by navigating from root to leaf based on feature values
- ▶ Each leaf contains the average rating of training examples in that region
- ▶ Naturally handles:
  - ▶ Both categorical features (genres) and numerical features (year)
  - ▶ Non-linear relationships between features and ratings
  - ▶ Feature interactions (e.g., preference for action movies, but only from recent years)

# Training Process: The Decision Tree Algorithm

## Tree Construction Steps

1. Start with all training data at the root node
2. For each node:
  - ▶ Calculate current variance of ratings
  - ▶ For each feature and potential threshold:
    - ▶ Split data according to the threshold
    - ▶ Calculate variance of each resulting subset
    - ▶ Compute variance reduction:

$$\Delta Var = Var(S) - \left( \frac{n_L}{n} Var(S_L) + \frac{n_R}{n} Var(S_R) \right)$$

- ▶ Choose split that maximizes variance reduction
  - ▶ If meaningful split found, create internal node and continue recursively
  - ▶ Otherwise, create leaf node with the mean rating
3. Stopping criteria prevent overfitting:
    - ▶ Maximum depth reached
    - ▶ Minimum samples for a split not met
    - ▶ No significant variance reduction possible

# Advantages and Limitations of Decision Trees

## Advantages

### ► Interpretability

- Clear decision rules that can explain predictions
- Easy to understand which features influence recommendations
- Provides insights into user preferences

### ► Content-Based Benefits

- No cold-start problem for new items with known features
- Can recommend niche items that have few or no ratings
- Privacy-preserving (doesn't require data from other users)

## Limitations

- Limited by available content features
- Can't capture collaborative patterns across users
- May create over-specialized recommendations
- Requires separate tree training for each user
- Cold-start problem for new users still exists

# What is PureSVM?

- ▶ A pure Python implementation of Support Vector Regression (SVR) for movie rating prediction
  - ▶ Content-based filtering approach using movie features (genres and release year)
  - ▶ Core principle: Creates a hyperplane in feature space that best predicts ratings with minimal error
  - ▶ Goal: Build personalized SVR models for each user to predict ratings for unseen movies
  - ▶ Uses epsilon-insensitive loss function that tolerates small errors
- 
- ▶ *PureSVM applies Support Vector Regression to recommendation, creating a personalized regression model for each user based on their historical preferences and movie features, balancing accuracy with generalization*

# Class Structure: Key Components

## ► Initialization Parameters

- **C**: Regularization parameter (default=1.0)
- **epsilon**: Width of the insensitive zone (default=0.1)
- **learning\_rate**: Step size for gradient descent (default=0.001)
- **max\_iterations**: Maximum training iterations (default=1000)

## ► Key Data Structures

- **user\_models**: Dictionary mapping user IDs to their personalized SVR models
- **movie\_features**: Dictionary mapping movie IDs to feature vectors
- Each model contains: weights, bias, and feature normalization parameters

## ► Primary Methods

- **fit**: Train personalized SVR models for users with sufficient ratings
- **predict**: Generate rating predictions using appropriate model
- **\_normalize\_features**: Scale features to  $[0,1]$  range
- **\_compute\_gradient**: Calculate gradients for model optimization
- **\_stochastic\_gradient\_descent**: Core optimization algorithm

# Method: fit() - Training the SVR Models

## 1. Feature Extraction

- ▶ Load movie content features (genres and release year)
- ▶ Create feature vectors for each movie:
  - ▶ Binary encoding for each genre (1 if present, 0 if not)
  - ▶ Release year as a numerical feature
- ▶ Example: [0, 1, 1, 0, ..., 0, 2019] for an Action/Adventure movie from 2019

## 2. Rating Organization

- ▶ Group ratings by user
- ▶ Filter to include only users with sufficient ratings (minimum 10)

# Method: `fit()` - Training the SVR Models

## 3. User-Specific Model Creation

- ▶ For each eligible user:
  - ▶ Extract feature vectors and ratings for movies they've rated
  - ▶ Train a personalized SVR model using stochastic gradient descent
  - ▶ Store model parameters (weights, bias, normalization constants)

# Method: `_stochastic_gradient_descent()` - The Optimization Process

## SVR Parameter Optimization

- ▶ Initialize weights to zeros, bias to zero
- ▶ Normalize features to  $[0,1]$  range for stable training
- ▶ For each iteration:
  - ▶ Shuffle the training data for better convergence
  - ▶ Process data in mini-batches for efficiency
  - ▶ For each mini-batch:
    - ▶ Compute gradients using epsilon-insensitive loss
    - ▶ Update weights and bias using calculated gradients
  - ▶ Periodically check convergence using loss function
  - ▶ Stop if loss improvement falls below threshold

**Key insight:** The epsilon-insensitive loss means that errors smaller than epsilon contribute nothing to the gradient, creating a "tube" around the regression line where small errors are tolerated



## Method: `_compute_gradient()` - Gradient Calculation

### ▶ **Epsilon-insensitive Loss Concept**

- ▶ Standard SVR uses a loss function that ignores errors smaller than epsilon
- ▶ Only errors outside the epsilon tube contribute to model updates
- ▶ Mathematical formulation of epsilon-insensitive loss:

$$L_{\epsilon}(y, f(x)) = \max(0, |y - f(x)| - \epsilon)$$

where:

- ▶  $y$  is the actual rating
- ▶  $f(x)$  is the predicted rating
- ▶  $\epsilon$  is the width of the insensitive zone

### ▶ **Gradient Computation Process**

- ▶ For each sample in the batch:
  - ▶ Calculate prediction:  $\hat{y} = \mathbf{w}^T \mathbf{x} + b$
  - ▶ Compute error:  $e = \hat{y} - y$
  - ▶ If  $|e| \leq \epsilon$ : No gradient contribution (in the tube)
  - ▶ If  $|e| > \epsilon$ : Gradient direction depends on error sign
  - ▶ Add regularization term to weight gradient:  $2C\mathbf{w}$
- ▶ Average gradients over the batch
- ▶ Return weight and bias gradients for parameter updates

# Method: predict() - Rating Prediction

## 1. Initial Checks

- ▶ Verify user has a trained model
- ▶ Verify movie exists in feature database
- ▶ Return None if either check fails

## 2. Model-Based Prediction

- ▶ Retrieve user's trained model (weights, bias, normalization parameters)
- ▶ Get movie feature vector
- ▶ Normalize features using user's specific normalization parameters:

$$\mathbf{x}_{\text{normalized}} = \frac{\mathbf{x} - \text{min\_vals}}{\text{range\_vals}}$$

- ▶ Compute prediction as linear function:

$$\hat{r} = \mathbf{w}^T \mathbf{x}_{\text{normalized}} + b$$

- ▶ Clip prediction to valid rating range (1-5)

## 3. Fallback Strategy

- ▶ If prediction is impossible, return None (handled by adapter class)

# The Machine Learning Algorithm: Support Vector Regression (SVR)

- ▶ Regression variant of Support Vector Machines (SVMs)
- ▶ Aims to find a function that has at most  $\epsilon$  deviation from target ratings
- ▶ Creates a "tube" around the regression hyperplane where errors are ignored
- ▶ Balances two objectives:
  - ▶ Minimize the complexity of the model (flat hyperplane)
  - ▶ Minimize prediction errors outside the epsilon-tube
- ▶ Formally tries to solve:

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n L_{\epsilon}(y_i, f(\mathbf{x}_i))$$

where:

- ▶ First term controls model complexity (regularization)
- ▶ Second term penalizes prediction errors beyond epsilon
- ▶ C controls the trade-off between these objectives

# Training Process: The Support Vector Regression Algorithm

## SVR Optimization Steps

1. Prepare and normalize feature data for each user
2. Initialize model parameters (weights to 0, bias to 0)
3. For each iteration of SGD:
  - ▶ Shuffle training data to avoid learning bias
  - ▶ Process data in mini-batches for computational efficiency
  - ▶ For each mini-batch:
    - ▶ Make predictions:  $\hat{y}_i = \mathbf{w}^T \mathbf{x}_i + b$
    - ▶ Calculate errors:  $e_i = \hat{y}_i - y_i$
    - ▶ Apply epsilon-insensitive loss:
    - ▶ If  $|e_i| \leq \epsilon$ : Error contributes nothing to gradient
    - ▶ If  $|e_i| > \epsilon$ : Error contributes to gradient based on sign
    - ▶ Update weights:  $\mathbf{w} = \mathbf{w} - \eta(\nabla L_{\mathbf{w}} + 2C\mathbf{w})$
    - ▶ Update bias:  $b = b - \eta \nabla L_b$
  - ▶ Periodically evaluate convergence
4. Store final model parameters for each user

# Advantages and Limitations of SVR for Recommendations

## Advantages

- ▶ Epsilon-insensitive loss makes model robust to noisy ratings
- ▶ Regularization prevents overfitting to training examples
- ▶ Works well with high-dimensional feature spaces

## Content-Based Benefits

- ▶ Can recommend items with no rating history (no cold-start for items)
- ▶ Explanations are straightforward ("recommended because it has features X")
- ▶ No dependency on other users' data (privacy friendly)

# Advantages and Limitations of SVR for Recommendations

## Limitations

- ▶ Requires sufficient ratings per user to train effective models
- ▶ Limited by the quality and expressiveness of available features
- ▶ Cannot capture complex non-linear relationships without kernel tricks
- ▶ Cold-start problem for new users still exists
- ▶ Training separate models for each user is computationally intensive

# What is PureNaiveBayes?

- ▶ A pure Python implementation of Naive Bayes for movie rating prediction
- ▶ Content-based filtering approach that applies Bayesian probability to recommendation
- ▶ Core principle: Uses Bayes' theorem to calculate  $P(\text{rating} \mid \text{features})$  for movie attributes
- ▶ Goal: Find the most likely rating given a movie's content features like genres and release year
- ▶ Applies user-specific personalization to adjust predictions based on user's rating history
  
- ▶ *PureNaiveBayes treats rating prediction as a probabilistic classification problem, learning which movie characteristics are associated with different rating levels and making predictions that balance feature evidence with user preferences*

# Class Structure: Key Components

## ▶ Initialization Parameters

- ▶ **rating\_levels**: Number of possible rating values (default=5)

## ▶ Key Data Structures

- ▶ **genre\_priors**: Conditional probabilities  $P(\text{genre}=1 \mid \text{rating})$  for each genre and rating
- ▶ **year\_mean/year\_var**: Mean and variance of release years for each rating level
- ▶ **rating\_priors**: Prior probabilities  $P(\text{rating})$  for each rating level
- ▶ **user\_avg\_ratings**: Average rating for each user (for personalization)
- ▶ **global\_mean**: Global mean rating across all users

## ▶ Primary Methods

- ▶ **fit**: Train the model by calculating probability distributions
- ▶ **predict**: Generate personalized rating predictions using Bayes' theorem
- ▶ **\_gaussian\_probability**: Calculate probabilities for numerical features



# Method: fit() - Training the Naive Bayes Model

## 1. Feature Extraction

- ▶ Load movie content features (genres and release year)
- ▶ Create feature vectors for each movie:
  - ▶ Binary encoding for each genre (1 if present, 0 if not)
  - ▶ Release year as a numerical feature

## 2. Data Preprocessing

- ▶ Filter ratings to include only movies with known features
- ▶ Calculate global mean rating across all users and movies
- ▶ Calculate per-user average ratings for personalization

# Method: `fit()` - Training the Naive Bayes Model

## 3. Probability Calculation

- ▶ Count occurrences of each rating level to compute  $P(\text{rating})$
- ▶ For each genre and rating level, count co-occurrences to compute  $P(\text{genre}=1 \mid \text{rating})$
- ▶ For each rating level, calculate mean and variance of movie release years

# Method: fit() - Probability Estimation and Smoothing

## Probability Estimation Process

### ► Prior Probabilities

- Calculate rating priors:  $P(\text{rating} = r) = \frac{\text{count}(r)}{\text{total ratings}}$

### ► Conditional Probabilities for Genres

- Apply Laplace smoothing for reliable probability estimates:

$$P(\text{genre}_i = 1 | \text{rating} = r) = \frac{\text{count}(\text{genre}_i = 1, \text{rating} = r) + 1}{\text{count}(\text{rating} = r) + 2}$$

- Smoothing prevents zero probabilities for unseen feature-rating combinations

# Method: fit() - Probability Estimation and Smoothing

## Probability Estimation Process

### ► Gaussian Parameters for Release Year

- For each rating level, calculate mean and variance:

$$\mu_r = \frac{1}{n_r} \sum_{i:\text{rating}_i=r} \text{year}_i$$
$$\sigma_r^2 = \frac{1}{n_r} \sum_{i:\text{rating}_i=r} (\text{year}_i - \mu_r)^2 + 1$$

- Add small constant to variance to prevent numerical issues

**Key insight:** Laplace smoothing and variance stabilization ensure robust probability estimates even with sparse data

# Method: predict() - Rating Prediction with Bayes' Theorem

## 1. Initial Checks

- ▶ Verify movie exists in feature database
- ▶ If not, prediction is impossible

## 2. Feature Extraction

- ▶ Retrieve genre indicators and release year for the target movie

# Method: predict() - Rating Prediction with Bayes' Theorem

## 3. Bayesian Inference

- ▶ For each possible rating level  $r$ :
  - ▶ Start with prior probability:  $\log P(r)$
  - ▶ For each genre feature, add:

$$\log P(\text{genre}_i|r) = \begin{cases} \log P(\text{genre}_i = 1|r) & \text{if movie has genre}_i \\ \log(1 - P(\text{genre}_i = 1|r)) & \text{otherwise} \end{cases}$$

- ▶ For release year, add log Gaussian probability:

$$\log P(\text{year}|r) = \log \left( \frac{1}{\sqrt{2\pi\sigma_r^2}} e^{-\frac{(\text{year} - \mu_r)^2}{2\sigma_r^2}} \right)$$

- ▶ Identify rating with maximum log probability

# Method: predict() - Rating Prediction with Bayes' Theorem

## 4. Personalization

- ▶ Adjust prediction using user's rating bias:

$$\text{rating}_{\text{personalized}} = r_{\text{max}} + \alpha \cdot (\text{avg}_{\text{user}} - \text{avg}_{\text{global}})$$

- ▶  $\alpha = 0.25$  controls the strength of personalization
- ▶ Clip final prediction to valid rating range (1-5)

# The Machine Learning Algorithm: Naive Bayes for Recommendations

- ▶ Applies Bayesian probability to the recommendation problem
- ▶ Core idea: Model  $P(\text{rating} \mid \text{features})$  using Bayes' theorem:

$$P(\text{rating} \mid \text{features}) \propto P(\text{features} \mid \text{rating}) \cdot P(\text{rating})$$

- ▶ "Naive" assumption: Features are conditionally independent given the rating:

$$P(\text{features} \mid \text{rating}) = \prod_i P(\text{feature}_i \mid \text{rating})$$

- ▶ Models different features differently:
  - ▶ Binary genre features: Bernoulli distribution
  - ▶ Continuous year feature: Gaussian distribution
- ▶ Works with log probabilities for numerical stability:

$$\log P(\text{rating} \mid \text{features}) \propto \log P(\text{rating}) + \sum_i \log P(\text{feature}_i \mid \text{rating})$$

- ▶ Makes prediction by finding the rating with maximum posterior probability (MAP estimate)



# Training Process: The Naive Bayes Algorithm

## Learning Steps

### 1. Prior Probability Estimation

- ▶ Count frequency of each rating level
- ▶ Calculate  $P(\text{rating}) = \frac{\text{count}(\text{rating})}{\text{total ratings}}$

### 2. Conditional Probability Estimation

- ▶ For categorical features (genres):
  - ▶ Count co-occurrences of each genre with each rating
  - ▶ Apply Laplace smoothing to avoid zero probabilities:

$$P(\text{genre} = 1 | \text{rating}) = \frac{\text{count}(\text{genre}=1, \text{rating}) + 1}{\text{count}(\text{rating}) + 2}$$

- ▶ For numerical features (release year):
  - ▶ Fit Gaussian distribution to years for each rating level
  - ▶ Calculate  $\mu_r$  (mean) and  $\sigma_r^2$  (variance)

### 3. Personalization Parameters

- ▶ Calculate average rating for each user
- ▶ Store user rating deviations from global mean

# Advantages and Limitations of Naive Bayes for Recommendations

## Advantages

### ► Probabilistic Foundation

- Provides principled way to combine multiple feature types
- Naturally handles uncertainty and missing data
- Can incorporate prior knowledge about rating distributions

### ► Practical Benefits

- Computationally efficient (no complex optimization)
- Works well with small training sets
- Fast prediction time
- Simple yet often surprisingly effective

# Advantages and Limitations of Naive Bayes for Recommendations

## Limitations

- ▶ "Naive" independence assumption often violated in real data
- ▶ Less expressive than more complex models for capturing feature interactions
- ▶ Sensitive to feature representation choices
- ▶ May require careful smoothing for reliable probability estimates
- ▶ Limited by available content features quality

# What is PureTFIDF?

- ▶ A pure Python implementation of TF-IDF (Term Frequency-Inverse Document Frequency) for movie recommendations
- ▶ Content-based filtering approach that analyzes movie titles, genres, and user-generated tags
- ▶ Core principle: Textual similarity between items indicates similar user preferences
- ▶ Goal: Use text representations to find similar movies and predict user ratings
- ▶ Combines text mining with nearest-neighbor collaborative techniques
- ▶ *PureTFIDF transforms movie metadata into a rich feature space where semantic relationships are captured through word importance weighting, enabling recommendations based on content similarity rather than user overlap*

# Class Structure: Key Components

## ▶ Initialization Parameters

- ▶ **k**: Number of nearest neighbors to consider (default=40)

## ▶ Key Data Structures

- ▶ **movie\_documents**: Preprocessed text for each movie
- ▶ **vocabulary**: List of significant terms across all documents
- ▶ **tfidf\_vectors**: TF-IDF weight vectors for each movie
- ▶ **similarities**: Precomputed similarity matrix between all movie pairs
- ▶ **user\_ratings**: User rating history for prediction

## ▶ Primary Methods

- ▶ **fit**: Build vocabulary, compute TF-IDF vectors and similarities
- ▶ **predict**: Generate rating predictions using similar movies
- ▶ **\_preprocess\_text**: Clean and tokenize text data
- ▶ **\_compute\_cosine\_similarity**: Calculate similarity between vectors
- ▶ **get\_similar\_movies**: Find most similar movies to a given movie

# Method: fit() - Text Processing and Feature Extraction

## 1. Document Creation

- ▶ Load movie metadata (titles, genres) and user tags
- ▶ For each movie, create a text document by combining:
  - ▶ Movie title (with year extracted)
  - ▶ Genre information (converted to text)
  - ▶ User-generated tags (aggregated)
- ▶ Example: "Toy Story animation adventure fantasy fun pixar childhood 1995"

## 2. Text Preprocessing

- ▶ Convert all text to lowercase
- ▶ Remove punctuation and numbers
- ▶ Tokenize into individual words
- ▶ Remove stopwords ("the", "and", "of", etc.)
- ▶ Filter out very short tokens (length < 3)

# Method: fit() - Vocabulary Building and TF-IDF Calculation

## 3. Vocabulary Construction

- ▶ Collect all unique tokens across all movie documents
- ▶ Filter to include only terms that appear in at least 2 documents
- ▶ This reduces noise and computational complexity

# Method: fit() - Vocabulary Building and TF-IDF Calculation

## 4. TF-IDF Vector Computation

- ▶ Calculate Term Frequency (TF) for each term in each document:

$$\text{TF}(t, d) = \frac{\text{count of term } t \text{ in document } d}{\text{total terms in document } d}$$

- ▶ Calculate Inverse Document Frequency (IDF) for each term:

$$\text{IDF}(t) = \log \left( \frac{\text{total number of documents}}{\text{number of documents containing term } t} \right)$$

- ▶ Compute TF-IDF weight for each term in each document:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$



# Method: fit() - Similarity Matrix Construction

## 5. Similarity Computation

- ▶ For each pair of movies, compute cosine similarity between TF-IDF vectors:

$$\text{similarity}(i, j) = \frac{\vec{v}_i \cdot \vec{v}_j}{\|\vec{v}_i\| \times \|\vec{v}_j\|}$$

- ▶ Where  $\vec{v}_i$  and  $\vec{v}_j$  are the TF-IDF vectors for movies  $i$  and  $j$
- ▶ For sparse vectors, efficient calculation focuses only on common terms:

$$\text{similarity}(i, j) = \frac{\sum_{t \in (T_i \cap T_j)} \text{TF-IDF}(t, i) \times \text{TF-IDF}(t, j)}{\sqrt{\sum_{t \in T_i} \text{TF-IDF}(t, i)^2} \times \sqrt{\sum_{t \in T_j} \text{TF-IDF}(t, j)^2}}$$

- ▶ Store all similarities in a matrix for efficient lookup during prediction

**Key insight:** Higher similarity values indicate movies with similar content features, which often correlates with similar user preferences

# Method: predict() - Rating Prediction

## ► Process Overview

- Use item-based collaborative filtering approach
- Identify movies similar to target movie that user has already rated
- Weight those ratings by similarity to generate prediction

## ► Algorithm Steps

1. Verify user has ratings and movie exists in similarity matrix
2. For each movie rated by the user:
  - Retrieve similarity between this movie and target movie
  - Store pairs of (similarity, rating) for movies with positive similarity
3. Select top- $k$  movies with highest similarity to target
4. Compute weighted average of ratings:

$$\hat{r}_{ui} = \frac{\sum_{j \in N_i(u)} \text{sim}(i, j) \times r_{uj}}{\sum_{j \in N_i(u)} \text{sim}(i, j)}$$

where  $N_i(u)$  is the set of top- $k$  similar items rated by user  $u$

# Method: `get_similar_movies()` - Finding Related Content

## ► Purpose

- Find most similar movies to a target movie based on content
- Useful for "more like this" recommendations
- Acts as an explanation mechanism for recommendations

## ► Implementation

1. Retrieve target movie's index in similarity matrix
2. Extract similarities to all other movies
3. Filter out negative similarities
4. Sort by similarity score in descending order
5. Return top- $n$  most similar movies

## ► Example Output

- For "Toy Story": ["Toy Story 2", "Finding Nemo", "Monsters, Inc.", "The Incredibles"]
- Shows movies with similar themes, genres, or described with similar terms

# The Machine Learning Algorithm: TF-IDF and Cosine Similarity

## ▶ **TF-IDF: Core Concept**

- ▶ Statistical measure used to evaluate word importance in a document within a collection
- ▶ Increases proportionally to word frequency in document
- ▶ Decreases as word appears in more documents (penalizes common words)
- ▶ Balances local importance (TF) with global discrimination power (IDF)

## ▶ **Why TF-IDF Works for Recommendations**

- ▶ Creates content fingerprints that emphasize distinguishing terms
- ▶ "Comedy" in many movies gets low weight (common across collection)
- ▶ Specific actor names or unique genre combinations get high weight
- ▶ Captures semantic similarity beyond simple keyword matching
- ▶ Works well with unstructured or semi-structured textual metadata

# Training Process: From Text to Recommendations

## Complete TF-IDF Pipeline

### 1. Corpus Creation

- ▶ Assemble textual representation for each item
- ▶ Clean and preprocess text (stopword removal, tokenization)

### 2. Vocabulary Building

- ▶ Extract unique terms across all documents
- ▶ Filter out rare terms to reduce noise

### 3. TF-IDF Calculation

- ▶ Calculate term frequency in each document
- ▶ Calculate inverse document frequency across corpus
- ▶ Compute TF-IDF weights:  $\text{tfidf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$

### 4. Similarity Computation

- ▶ Convert each document to a vector in term space
- ▶ Compute cosine similarity between all document pairs
- ▶ Store in similarity matrix for efficient lookup

### 5. Rating Prediction

- ▶ Use content similarity to identify related items
- ▶ Apply item-based collaborative filtering using similarity weights

# Advantages and Limitations of TF-IDF for Recommendations

## Advantages

### ▶ **Content Understanding**

- ▶ Captures semantic relationships between items
- ▶ Works well with rich textual data (descriptions, reviews, tags)
- ▶ No cold-start problem for new items with descriptive text

### ▶ **Technical Benefits**

- ▶ Intuitive and interpretable recommendations
- ▶ Computationally efficient after initial preprocessing
- ▶ Can work with minimal user preference data
- ▶ Easily combined with other filtering approaches

## Limitations

- ▶ Cannot capture context or word meaning (just frequency)
- ▶ Limited by quality and quantity of available text
- ▶ May overemphasize rare but unimportant terms
- ▶ Does not capture user-specific preferences beyond ratings
- ▶ "Bag of words" approach loses word order and semantic structure