Recommendation systems:

Data Collection:

Recommendation systems need user-item interaction data. Examples include:

- **Explicit Feedback**: ratings, reviews (e.g., 5-star ratings)

  Users **deliberately express their opinions** about items.

  - ⭐ **Ratings**: "I give this movie 4 out of 5"
  - 📝 **Reviews**: "This phone has great battery life"
  - 👍 / 👎 Thumbs up/down
  - ❤️ "Favorite" or "Like" buttons

- **Implicit Feedback**: clicks, views, purchases, watch time

  - 👁️ **Views**: Watching a video
  - 🖱️ **Clicks**: Clicking on an item or link
  - ⏳ **Dwell Time**: How long they watch/read/interact
  - 🛒 **Purchases** or Add to Cart
  - 🎵 Replays or skips (in music platforms)

Aspect	Explicit Feedback	Implicit Feedback
User effort	High	Low
Signal quality	Strong	Noisy
Data availability	Sparse	Abundant
Contains negatives?	Yes (e.g., low rating)	No (absence ≠ dislike)
Example systems	Amazon ratings, Yelp reviews	Netflix views, Spotify listens

Real-world Use:

- **Netflix** combines both:
  - Star ratings = explicit
  - Viewing behavior = implicit
- **YouTube** uses implicit (watch time, likes)
- **Amazon** combines reviews with purchase behavior

🔹 Data Storage:

Data is typically stored in:

📊 Relational Databases (SQL)

What It Is:

- Classic **structured** data storage using **tables** (rows and columns).
- You define a **schema** (e.g., Users, Items, Ratings).

Examples:

- MySQL
- PostgreSQL
- SQLite

Features:

- Strong **data integrity** (ACID properties).
- **Easy to query** using SQL (`JOIN`, `GROUP BY`, etc.).
- Great for **moderate-scale** applications.

  - **NoSQL stores** (MongoDB, Cassandra for scalability)

📦 NoSQL Databases

What It Is:

- Non-tabular, **schema-less** systems.

- Supports **unstructured/semi-structured** data like JSON, logs.

📌 *Examples:*

- MongoDB (Document store)
- Cassandra (Wide-column store)
- Redis (Key-value store)

Features:

- **Highly scalable** (horizontal scaling).
- Fast read/write for large data volumes.
- Flexible schema: evolve data structure over time.

Distributed systems like Hadoop or cloud storage for big data

- Designed for **very large-scale** data that doesn't fit on a single machine.

📌 *Examples:*

- **Hadoop HDFS**: Distributed file system
- **Amazon S3**, **Google Cloud Storage**: Object storage in the cloud
- **Apache Spark**: Process and analyze data at scale

🔍 *How It's Used:*

- Store raw event logs (clicks, views, purchases)
- Batch processing for model training
- Store millions of user-item interactions

🟠 *Features*

- Can handle **petabytes** of data
- Works well with **parallel processing**
- Ideal for training models on large datasets

Storage Type	Best For	Examples	Strengths
Relational (SQL)	Structured data, small-medium scale	MySQL, PostgreSQL	Query power, schema integrity
NoSQL	Semi-structured, fast writes	MongoDB, Cassandra	Scalability, flexibility
Distributed/Big Data	Very large-scale data & logs	Hadoop, S3, Spark	Parallel processing, scale

🔹 Data Filtering:

Before building models, data must be cleaned:

Ensure your model trains on **high-quality, meaningful data**

- Remove duplicates, spam, and bots

🔄 *Duplicates:*

Sometimes users rate or interact with the same item multiple times.

- **Problem**: It skews the model by overweighting those interactions.
- ✅ **Fix**: Keep the latest or average the multiple entries.

Some users (or automated scripts) might:

- Give random or extreme ratings
- Click everything to manipulate results (especially in e-commerce, reviews)

- Filter out inactive users or items with very few interactions
- Normalize ratings (e.g., subtract user/item mean)

2. Collaborative Filtering

This is a **core technique** used to recommend items by leveraging patterns in user-item interactions.

🔹 Types:

a) User-Based Collaborative Filtering

- Recommends items that similar users liked.
- Example: "Users like you also liked..."
- "**People like you liked X, so you might like X too.**"

- It works on the assumption that **similar users have similar tastes**.

b) Item-Based Collaborative Filtering

- Recommends items similar to what the user liked before.
- Example: "If you liked *Item A*, you might like *Item B*"

🔹 Core Idea:

Use similarity metrics like:

- Cosine similarity

Use similarity metrics like:

- Cosine similarity
- Pearson correlation

- Pearson correlation

❗ Problem: Doesn't work well with sparse data or new users/items (Cold Start).

Some users rate everything 5 stars, some are stingier. Pearson helps by **centering** ratings around each user's average.

Metric	Adjusts for User Bias?	Handles Missing Data	Best For
Cosine Similarity	❌ No	✅ Yes (common items)	Implicit feedback, fast calc
Pearson Correlation	✅ Yes	✅ Yes (common items)	Explicit feedback (ratings)

Measures **linear correlation** between two users' ratings, **adjusted for their rating biases**.

🔢 3. Factorization Methods

Used to deal with **sparsity** and **scale** in large datasets.

Sparsity - Used to deal with **sparsity** and **scale** in large datasets.

```
        Movie1   Movie2   Movie3   Movie4
User1     5        ?        ?        4
User2     ?        3        ?        ?
User3     4        ?        ?        ?
User4     ?        ?        2        ?
```

Out of 16 possible entries, only **5 are filled** → that's **high sparsity**.

📉 Problem:

- Hard to find enough data to compute similarities
- Many users/items don't have enough history
- Models can't generalize well due to missing info

Scale refers to the **size of the data** — number of users, items, and interactions.

For large platforms like Netflix, Amazon, Spotify:

- **Millions** of users
- **Millions** of items
- **Billions** of interactions

🧠 Why It's Challenging:

- More memory and processing power required
- Need **efficient algorithms** that can run in real time
- Models must be **distributed** across many machines

🧩 How They're Related:

Challenge	What it means	Why it matters
Sparsity	Too many missing values	Can't compute accurate recommendations
Scale	Too much data	Need efficient, distributed algorithms

🔹 Matrix Factorization:

- Decomposes the user-item matrix into **latent factors**:

  $R \approx U \times V^T$

  - R: user-item rating matrix
  - U: user features
  - V: item features

🔹 Common Techniques:

- **SVD (Singular Value Decomposition)**

1. SVD (Singular Value Decomposition)

📘 What It Does:

SVD factorizes a matrix `R` into **three matrices**:

$$R = U \times \Sigma \times V^T$$

- `U`: user matrix (orthogonal)
- `Σ`: diagonal matrix of **singular values** (importance of features)
- `Vᵀ`: item matrix (orthogonal)
- You keep only the **top-k singular values** (dimensionality reduction)
- Approximate `R` with a smaller matrix:
- **Needs full matrix** (not good for sparse data)
- Better for **small/medium datasets**
- Not optimized for **missing values**

- **ALS (Alternating Least Squares)**

ALS (Alternating Least Squares)

Solve matrix factorization even with **missing data** (i.e., sparsity).

Instead of trying to solve for both user and item matrices at once, **ALS alternates**:

- Fix `V`, solve for `U`
- Fix `U`, solve for `V`
- Repeat until convergence

Optimizes a **regularized loss** (to prevent overfitting):

Funk-SVD (used by Netflix)

- Doesn't compute full SVD.
- Optimizes only for the **observed ratings** using **Stochastic Gradient Descent (SGD)**.

A **latent factor** is an **underlying, hidden feature** that helps explain why a user likes or interacts with certain items.

- **Non-negative Matrix Factorization**

These reduce dimensionality and uncover hidden patterns.

Works well in **text mining**, **bioinformatics**, etc.

Easier to interpret latent features:

- Each latent factor might represent a **genre**, **topic**, or **taste**

📏 4. Evaluation Metrics

1. Precision

Measures relevance among recommended items.

```
Precision = (Relevant items recommended) / (Total items recommended)
```

🎯 What it Measures:

Out of the items your model **recommended**, how many were actually **relevant** to the user?

📦 Use Case:

- **E-commerce** (e.g., Amazon): Are the top 10 recommended products truly what the user might buy?
- **Streaming** (e.g., Spotify): Do the next 5 songs match the user's taste?

✅ Why It's Used:

You want to **maximize relevance**. High precision means you're not recommending junk.

2. Recall

Measures coverage of all relevant items.

```
Recall = (Relevant items recommended) / (Total relevant items)
```

🎯 What it Measures:

Out of all the items the user actually liked, how many did your system manage to **recommend**?

📦 Use Case:

- **News** or **education** apps: Did the system surface most of the important articles/courses a user would find useful?

✅ Why It's Used:

To check whether the system **missed relevant items**. High recall means broader **coverage** of what the user likes.

3. RMSE (Root Mean Square Error)

For rating prediction. Lower RMSE = better.

```
RMSE = sqrt(mean((predicted - actual)^2))
```

🎯 What it Measures:

How **far off** your predicted ratings are from the actual ratings.

📦 Use Case:

- **Movie rating systems** like Netflix: Predict how much a user would rate a movie (e.g., 4.5 stars vs 2 stars)

✅ Why It's Used:

- Good for **rating prediction tasks**
- Lower RMSE = more accurate predictions

4. MRR (Mean Reciprocal Rank)

Used for ranked lists. Focuses on the **rank** of the first relevant item.

```
MRR = average(1 / rank of first relevant item)
```

🎯 What it Measures:

How **early** the **first relevant item** appears in your ranked recommendation list.

📦 Use Case:

- **Search engines**, **Q&A systems**, **Spotify/YouTube**: If the top-ranked result is relevant, the system gets rewarded.
- **Chatbots**: Ranking best responses.

✅ Why It's Used:

Measures **how fast** your model gets to the good stuff.

5. MAP@K (Mean Average Precision at K)

Averages precision at each relevant position up to top K items.

- Measures both order and relevance.
- Higher is better.

🎯 What it Measures:

How well your system ranks relevant items **within the top-K recommendations**. It takes into account **both relevance and position**.

📦 Use Case:

- **Shopping**, **content platforms**, or **job recommenders**: Are you surfacing good results **early** in the list?

✅ Why It's Used:

Combines **precision and ranking order**. A more nuanced metric than just precision.

6. NDCG (Normalized Discounted Cumulative Gain)

Evaluates the quality of ranking. Rewards placing relevant items higher.

```
NDCG = DCG / IDCG
```

- DCG (Discounted Cumulative Gain)
- IDCG = ideal DCG (best possible ranking)

Great for **ranking-based** recommenders like YouTube, Amazon

🎯 What it Measures:

How well your system ranks **highly relevant items toward the top**. It **rewards better ordering** of results.

📦 Use Case:

- **YouTube, Amazon, Netflix**: Prioritize highly relevant items at top positions.
- Great for **ranking-based systems**

✅ Why It's Used:

- More realistic for **ranking tasks**
- Emphasizes **early relevance**
- Normalized → values between 0 and 1, so you can compare across users or queries

Metric	Measures	Use Case	Goal
Precision	Relevance of recommended items	Shopping, video/music platforms	Recommend only relevant items
Recall	Coverage of all relevant items	News, learning, retail	Don't miss anything important
RMSE	Accuracy of rating predictions	Netflix, IMDB	Predict exact user ratings
MRR	Position of 1st relevant item	Search engines, Q&A bots	Get to relevant result quickly
MAP@K	Ranked precision up to K	Top-K recommendation tasks	Reward early relevance
NDCG	Overall ranking quality	YouTube, Amazon, Spotify	Prioritize best items at top

---

Summary

Component	What It Does
Data Collection & Storage	Gathers and organizes user behavior
Data Filtering	Cleans and preps the data
Collaborative Filtering	Uses user-item similarities to recommend
Factorization	Finds latent features and handles sparsity
Evaluation Metrics	Measures how good the recommendations are
