Recommender Systems: Machine Learning Models for Customer Prediction

Overview of the Project

- Build a recommender system using explicit movie ratings provided by users.
- Recommender systems are a specialized type of machine learning model.
- Recommender systems predict items for users that they have never seen before or interacted with explicitly (e.g., by adding to favorites or making a purchase).
- Train the system on explicit user ratings.
- Test how accurately the algorithm can predict ratings for movies the users have already seen and rated.

Introduction to Recommender Systems

What a Recommender System Is

- A specialized type of machine learning system.
- Predicts **ratings** or **preferences** a user might assign to an item.
- Recommends items based on:
  - User's past behavior.
  - Behavior of other users.
- Often presented as **Top-N Recommendations**:
  - A sorted list of items a user might like.

Examples of Recommender Systems

- Common names:
  - Recommender engines
  - Recommendation systems
  - Recommendation platforms
- Examples in practice:
  - **Amazon**: Product recommendations based on user interactions and global customer data.
  - **Netflix**: Movie recommendations based on user preferences and other users' preferences.
  - **Music platforms**: Song recommendations.
  - **Dating apps**: People recommendations.

Applications of Recommender Systems

- Recommender systems can suggest:
  - Physical products.
  - Digital content (e.g., music, videos).
  - Services or experiences.
- The key is learning user preferences over time and blending them with insights from others.

Understanding You through Implicit and Explicit Ratings

How Do Recommender Systems Work?

- Recommender systems aim to understand you — and everyone else.
- They collect data about your preferences and compare them with others.
- The goal: recommend items you might like based on patterns and behaviors.

Sources of Data

- Recommender systems need data to understand user interests.
- Two main types:
  - **Explicit Feedback**
  - **Implicit Behavior**

Explicit Feedback

- Users rate items directly (e.g., 1–5 stars, thumbs up/down).
- Examples:
  - Rating an online course.
  - Giving a movie 4 out of 5 stars.
- Benefits:
  - Clear indication of preferences.
- Limitations:
  - Requires user effort $\rightarrow$ data is sparse.
  - Ratings vary by user and culture.

Implicit Feedback

- Inferred from user actions — not directly stated.
- Examples:
  - Clicks
  - Purchases
  - Watch time
- Advantage: Much more data available.
- Challenge: Noisy signals, fraud, accidental clicks.

Clicks as Implicit Feedback

- Clicking a link can be interpreted as positive interest.
- Advantages:
  - High volume
- Disadvantages:
  - Not always meaningful (clickbait, accidents).
  - Vulnerable to bot/fraud activity.

Purchases as Implicit Feedback

- Strong indication of interest (requires effort and money).
- Resistant to fraud.
- Amazon uses purchase data to great effect.
- High-quality data often outweighs the need for complex algorithms.

Consumption as Feedback

- Time spent with content signals interest (e.g., minutes watched on YouTube).
- Less prone to fraud than clicks.
- A reliable metric, especially in content platforms.
- YouTubers aim to maximize watch time due to algorithm reliance.

Explicit vs Implicit Feedback

- **Explicit Feedback:**
  - Reliable but sparse.
  - Requires active input from users.
- **Implicit Feedback:**
  - More abundant.
  - May be noisier but covers broader behavior.

Working with MovieLens

- In this project, we use the **MovieLens public dataset**.
- Contains explicit ratings from 1 to 5 stars.

Garbage In, Garbage Out

- Even the best recommender algorithm won't perform well without:
  - **Good data**.
  - **Plenty of it**.
- Always start by asking: *What kind of feedback do I have?*
- Design your system around the strengths and limitations of your data.

Top-N Recommender Architecture

What is a Top-N Recommender System?

- Most real-world recommenders aim to present a finite list of top suggestions to users.
- A **Top-N Recommender** outputs the best N items for each user.
- Example: Amazon music widget shows 20 pages $\times$ 5 items $\rightarrow$ $N = 100$.

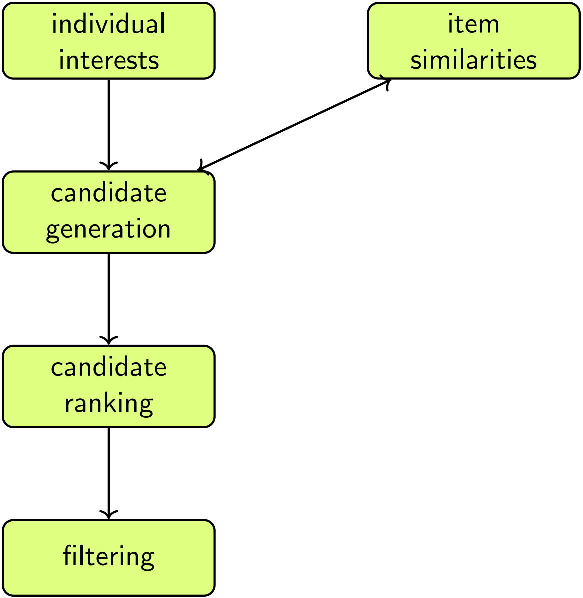
Research vs. Reality

- Academic research often focuses on predicting **ratings** for unseen items.
- In the real world, users don't care about predicted ratings — they care about **what to consume next**.
- Real recommender systems prioritize finding items users will **love**, not predicting items they will **hate**.

Top-N Pipeline Overview

- **Input:** Historical user data (e.g., ratings, purchases).
- **Pipeline Stages:**
  1. Candidate Generation
  2. Candidate Ranking
  3. Filtering
  4. Display
- Each stage plays a key role in producing a useful recommendation list.

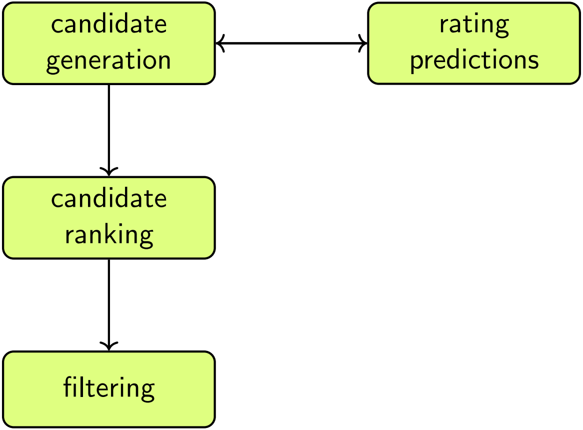
Architecture of a Top-N Recommender



Explaining Architecture of a Top-N Recommender

- **Individual Interests:** User-specific data such as past ratings, purchases, or views. Provides personalized input for recommendations.
- **Item Similarities:** Precomputed relationships between items based on user behavior or content. Learned during preprocessing or training.
- **Candidate Generation:** Uses interests and item similarities to build a shortlist of potentially relevant items for each user.
- **Candidate Ranking:** Scores and ranks the generated candidates using a machine learning model. ← **Training occurs here** — the model learns to predict user preferences based on historical data.
- **Filtering:** Removes items the user has already seen or that don't meet quality thresholds. Keeps only the Top-N results.

Old Architecture of a Top-N Recommender



Explanation of Old Architecture of a Top-N Recommender

- **Rating Predictions:** A precomputed matrix of predicted ratings for all user-item pairs. These predictions are often generated using matrix factorization or other ML models. ← **Training happens here** — the model is trained to estimate how a user might rate unseen items.
- **Candidate Generation:** Retrieves all predicted ratings for a given user from the prediction matrix. All items are considered candidates.
- **Candidate Ranking:** Sorts candidate items based on predicted ratings to prioritize the most relevant ones.
- **Filtering:** Removes already seen or irrelevant items and limits the output to the Top-N results.

1. Candidate Generation

- Based on the user's previous preferences (e.g., liked Star Trek).
- Query a similarity store to find related items (e.g., Star Wars).
- Use a distributed store like Cassandra, MongoDB, or Memcached.
- Optional: normalize user data (e.g., Z-scores), if sparsity allows.

Scoring Candidates

- Score items using:
  - Original item ratings.
  - Similarity strength between original and candidate items.
- Filter out low-scoring items early.
- Assign provisional scores to prepare for ranking.

2. Candidate Ranking

- Combine scores if items appear via multiple paths.
- Boost scores of repeated candidates.
- Sort candidates by final score.
- Optional: use machine learning (**learning to rank**) to optimize ranking.

3. Filtering Candidates

- Remove:
  - Items already rated/viewed.
  - Items on a stop-list (e.g., offensive or low-quality).
- Apply score thresholds.
- Truncate to top N results.

4. Displaying Results

- The final Top-N list is sent to the UI layer.
- Displayed as product/movie/music widgets.
- Should feel relevant, fresh, and personalized.

System Architecture

- Recommendation service is typically distributed and accessed via web APIs.
- Key components:
  - Interest database (ratings, purchases).
  - Similarity store (item-item).
  - Scoring and ranking logic.
  - Filtering logic.

Item-Based Collaborative Filtering

- Used by Amazon (2003 paper).
- Recommends items similar to those a user liked.
- Main challenge: building and maintaining item similarity data.
- Simple concept, scalable in practice.

Alternative: Precomputed Rating Predictions

- Build full matrix of predicted ratings for all users and items.
- For a user, sort all items by predicted rating $\rightarrow$ Top-N.
- Easier to benchmark prediction accuracy using:
  - **RMSE (Root Mean Square Error):**

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2}$$

    Penalizes large errors more heavily.
  - **MAE (Mean Absolute Error):**

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i|$$

    Treats all errors equally.
- **Drawbacks:**
  - Inefficient at runtime (especially for large catalogs).
  - Doesn't prioritize known user interests.

When is This Approach OK?

- Acceptable if:
  - Item catalog is small.
  - You want to evaluate rating prediction accuracy.
- Still doesn't align with actual user engagement goals.
- Use carefully — may optimize for the wrong thing.

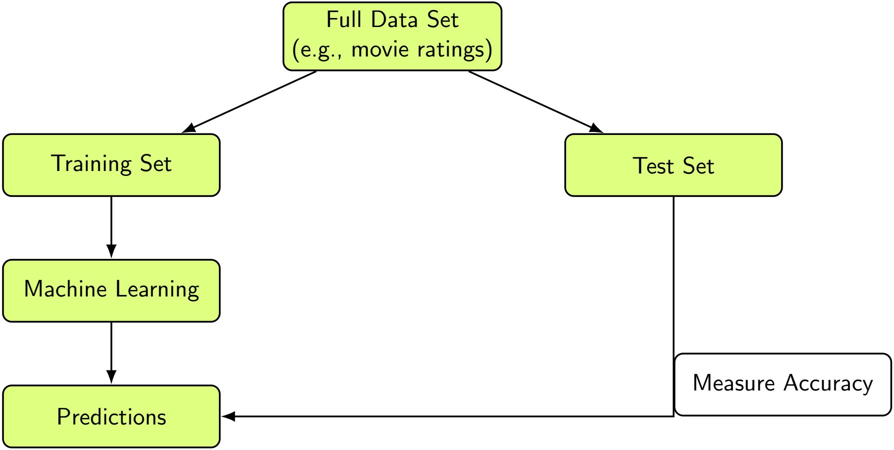
Evaluating Recommendations Systems

Train/Test and Cross Validation

Offline Evaluation Methodology

- Recommender systems are trained on prior user behavior to predict preferences.
- **Train/Test Split:**
  - Split ratings data into:
    - **Training set** (80–90% of data).
    - **Testing set** (10–20% of data).
  - Train the system using only the training data.
  - Test predictions using the reserved testing data.

Train/Test Split Diagram



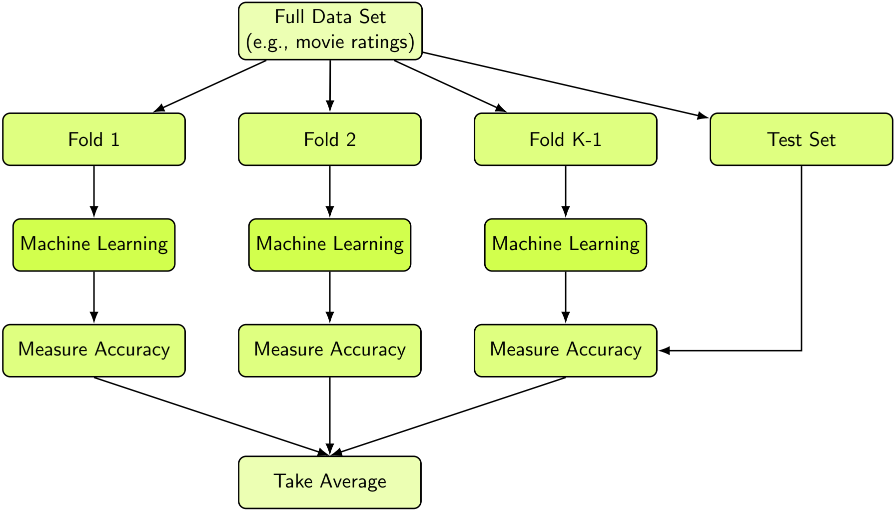
Example of Train/Test Evaluation

- Testing setup:
  - User rated the movie "Up" as 5 stars in the test set.
  - Recommender system predicts the rating without seeing the answer.
- Measure how close the prediction is to the real rating.
- Repeat across all test set ratings to calculate overall accuracy.
- Provides insight into how well the system predicts user ratings.

Improving Evaluation: K-Fold Cross-Validation

- An extension of train/test methodology:
  - Create multiple randomly assigned training/testing sets (**folds**).
  - Train the system on each fold independently.
  - Measure accuracy for each fold and average the results.
- Benefits:
  - Reduces the risk of overfitting to a single training set.
  - Ensures generalizability to different data subsets.
- Drawback: Requires significantly more computation.

K-Fold Cross-Validation Diagram



Limitations of Offline Testing

- Train/test and k-fold cross-validation measure:
  - How accurately the system predicts ratings for items users already saw.
- The goal of recommender systems:
  - Recommend new, unseen items that users find interesting.
- Fundamental problem:
  - Offline methods can't test the novelty or engagement of recommendations.
  - Researchers without access to live systems (e.g., Netflix, Amazon) must rely on offline methods.

Accuracy Metrics (RMSE, MAE)

Mean Absolute Error (MAE)

- MAE is a straightforward metric for evaluating accuracy.
- Measures the average absolute error between predicted and actual ratings.
- Formula:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

- y_i: Predicted rating.
- x_i: Actual rating.
- n: Number of ratings.

Example: Calculating MAE

- Let's evaluate a test set with four ratings.
- The predicted ratings, actual ratings, and absolute errors are:

Predicted Rating	**Actual Rating**	**Error**	**Absolute Error**
5	3	$5-3$	2
4	1	$4-1$	3
5	4	$5-4$	1
1	1	$1-1$	0

- Sum of absolute errors: $2+3+1+0=6$.
- MAE: $\frac{6}{4}=1.5$.

Root Mean Square Error (RMSE)

- RMSE is another common metric for evaluating accuracy.
- Penalizes large errors more than MAE does by squaring the errors.
- Formula:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (y_i - x_i)^2}{n}}$$

- y_i: Predicted rating.
- x_i: Actual rating.
- n: Number of ratings.

Example: Calculating RMSE

- Using the same test set as before, let's calculate the squared errors:

Predicted Rating	Actual Rating	Error	Squared Error
5	3	$5 - 3 = 2$	$2^2 = 4$
4	1	$4 - 1 = 3$	$3^2 = 9$
5	4	$5 - 4 = 1$	$1^2 = 1$
1	1	$1 - 1 = 0$	$0^2 = 0$

- Sum of squared errors: $4 + 9 + 1 + 0 = 14$.
- RMSE: $\sqrt{\frac{14}{4}} \approx 1.87$.

MAE vs. RMSE

- MAE treats all errors equally.
- RMSE penalizes large errors more heavily.
- RMSE is higher than MAE when large errors are present.
- For our example:
  - MAE: 1.5.
  - RMSE: 1.87.

Real-World Metrics

- Users care about the quality of recommendations, not prediction accuracy.
- Top-N recommendation lists are more relevant in practice:
  - How likely are users to interact with recommended items?
  - Do recommendations help users discover new content?
- Offline metrics like MAE and RMSE are useful for development but limited in scope.

Top-N Hit Rate - Ways

Hit Rate (HR)

- Top-N means the system picks the best N recommendations (like top 5 or top 10) that it thinks a user will like the most
- Hit rate measures the success of a recommender system in generating Top-N recommendations.
- If one of the recommendations in a user's Top-N list matches an item they actually rated, it's considered a "hit."
- Formula:

$$\text{Hit Rate} = \frac{\text{Hits}}{\text{Users}}$$

- Hits: Number of successfully recommended items.
- Users: Total number of users.

Example: Hit Rate

- Let's consider a test set of Top-N recommendations.
- If a user rates at least one of the recommended items, it's counted as a "hit."
- Example calculation:

User	**Hit (Yes/No)**
User 1	Yes
User 2	No
User 3	Yes
User 4	Yes

- Total Hits $= 3$, Total Users $= 4$.
- Hit Rate $= \frac{3}{4} = 0.75$ or 75%.

Average Reciprocal Hit Rate (ARHR)

- ARHR builds on hit rate but accounts for the position of hits in the Top-N list.
- Formula:

$$\text{ARHR} = \frac{\sum_{i=1}^{n} \frac{1}{\text{rank}_i}}{\text{Users}}$$

- rank_i: Rank position of the hit.
- Users: Total number of users.
- Hits near the top of the list (e.g., rank 1) are weighted more heavily than hits at the bottom.

Example: ARHR Calculation

- Consider the following example with three hits:

Rank	Reciprocal Rank
1	$\frac{1}{1} = 1.0$
2	$\frac{1}{2} = 0.5$
3	$\frac{1}{3} = 0.33$

- ARHR $= \frac{1.0+0.5+0.33}{3} \approx 0.61$.

Cumulative Hit Rate (cHR)

- A variation of hit rate that filters hits based on a rating threshold.
- Only hits above a certain predicted or actual rating threshold are counted.
- Example:

Hit Rank	Predicted Rating
1	5.0
2	3.0
3	5.0
4	2.0

- If the threshold is ≥ 3.0, we exclude hits with predicted ratings below 3.0.
- Adjusted cHR $= \frac{\text{Filtered Hits}}{\text{Users}}$.

Cumulative Hit Rate (cHR) - Formula

- **Definition:** Measures hit rate while excluding low-confidence or low-rating predictions.
- **Formula:**

$$\text{cHR} = \frac{\text{Number of Hits with Rating} \geq \text{Threshold}}{\text{Total Users}}$$

- Encourages recommending items the model believes the user will rate highly.

Example: Cumulative Hit Rate Calculation

- Assume rating threshold $= 3.0$

Hit Rank	**Predicted Rating**
1	5.0
2	3.0
3	5.0
4	2.0

- Filtered Hits $= 3$ (ratings 5.0, 3.0, 5.0)
- Total Users $= 4$
- **cHR =** $\frac{3}{4} = 0.75$ **or 75%**

Rating Hit Rate (rHR)

- **Definition:** Analyzes hit rate across different predicted rating bins.
- **Formula:**

$$\mathrm{rHR}_r = \frac{\text{Number of Hits with Predicted Rating } r}{\text{Total Users}}$$

- Breaks down hit rates by predicted rating score.
- Helps understand the distribution of predicted ratings for successful recommendations.
- Example:

Rating	Hit Rate
5.0	0.001
4.0	0.004
3.0	0.030
2.0	0.001
1.0	0.0005

Converge, Diversity and Novelty

Coverage

- **Definition:** Coverage measures the percentage of user-item pairs that your system can make predictions for.
- Formula:

$$\text{Coverage} = \frac{\text{Predicted User-Item Pairs}}{\text{Total Possible User-Item Pairs}}$$

- High coverage ensures that more items and users are included in the recommendation process.
- Tradeoff: Higher coverage might reduce accuracy.

Example: Calculating Coverage

- Suppose we have:
  - 5 users.
  - 6 items.
  - Total possible user-item pairs: $5 \times 6 = 30$.
  - Model predicts ratings for 18 of those pairs.

Coverage Formula

$$\text{Coverage} = \frac{18}{30} = 0.6 \text{ or } 60\%$$

- The recommender covers 60% of the user-item space.
- Higher coverage ensures broader personalization, but may impact accuracy.

Item Similarity

- Similarity quantifies how closely two items are related based on user behavior.
- Commonly used to recommend items similar to those a user liked.
- Types of similarity:
  - **Cosine similarity**: Angle between rating vectors.
  - **Pearson correlation**: Measures linear correlation.
- Similarity is usually computed using rating vectors for items across users.

Example: Calculating Cosine Similarity

- Ratings from 3 users for two items:

User	Item A	Item B
User 1	5	3
User 2	4	2
User 3	0	2

- Cosine similarity formula:

$$\text{sim}(A, B) = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \times ||\vec{B}||}$$

- Vectors:

$$\vec{A} = [5, 4, 0], \quad \vec{B} = [3, 2, 2]$$

- Result:

$$\text{sim}(A, B) \approx 0.76$$

Diversity

- **Definition:** Diversity measures how varied the recommendations are within the Top-N list.
- Calculated using average similarity S between recommended items:

$$\text{Diversity} = 1 - S$$

- S: Average similarity between item pairs in the recommendation list.
- High diversity may lead to novel recommendations but risks reducing relevance.

Example: Calculating Diversity

- Example similarity scores between items in a Top-N list:

Item Pair	**Similarity**
Item 1, Item 2	0.7
Item 2, Item 3	0.8
Item 3, Item 1	0.6

- Average similarity $S = \frac{0.7+0.8+0.6}{3} = 0.7$.
- Diversity $= 1 - 0.7 = 0.3$.

Novelty

- **Definition:** Measures how "unpopular" the recommended items are.
- Computed as the mean popularity rank of the recommended items.

$$\text{Novelty} = \frac{1}{N}\sum_{i=1}^{N} \text{rank}(i)$$

- **N**: Total number of recommended items in the Top-N list.
- **rank(i)**: Popularity rank of item i (higher = less popular).
- Higher novelty $\rightarrow$ more obscure items are being recommended.
- Tradeoff: Too much novelty can harm relevance and trust.

Example: Calculating Novelty

- Assume a Top-5 recommendation list for a user:

Item	Popularity Rank
Item A	45
Item B	120
Item C	230
Item D	310
Item E	400

Novelty Formula

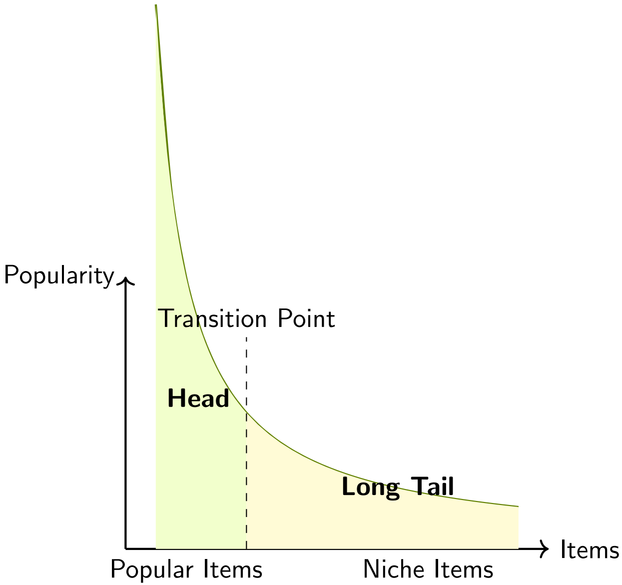
$$\text{Novelty} = \frac{45 + 120 + 230 + 310 + 400}{5} = 221$$

- Higher mean rank $\rightarrow$ less popular items $\rightarrow$ more novelty.
- Promotes long-tail discovery, but too much may harm user trust.

The Long Tail Effect

- Popularity of items often follows a "long tail" distribution:
  - Few items are very popular (head).
  - Most items have low demand (long tail).
- Recommender systems surface niche, less popular items in the long tail.
- Benefits:
  - Users discover new interests.
  - Lesser-known items get more exposure.

The Long Tail Effect



Balancing Metrics in Recommender Systems

- Recommender systems require a balance between:
  - Familiar, popular items (trust-building).
  - Novel, diverse items (serendipitous discovery).
- Metrics such as coverage, diversity, and novelty are interdependent:
  - Increasing one might reduce another.
  - The right balance depends on your use case and audience.

Churn, Responsiveness and A/B Tests

Churn

- **Definition:** Measures how often recommendations for a user change.
- **Purpose:** Indicates how sensitive a system is to new user behavior.
- High churn can indicate:
  - System responsiveness to user updates.
  - Over-sensitivity to minor user actions.
- Tradeoff: High churn risks showing irrelevant or overly dynamic recommendations.

Example: Churn

- Suppose a user rates a new movie:
  - Does this substantially change their recommendations?
  - High churn indicates recommendations update drastically.
- Balance required: Randomization can keep recommendations fresh but may reduce relevance.

Responsiveness

- **Definition:** Measures how quickly new user behavior influences recommendations.
- **Purpose:** Ensures that recommendations adapt to user updates efficiently.
- Levels of responsiveness:
  - **Immediate:** Recommendations change instantly after user actions.
  - **Delayed:** Recommendations update after periodic data processing.
- Tradeoff: Higher responsiveness increases system complexity and maintenance costs.

Balancing Metrics

- Offline metrics include:
  - MAE, RMSE.
  - Hit Rate, Diversity, Novelty, Coverage.
  - Churn, Responsiveness.
- Tradeoffs between metrics:
  - High diversity may reduce relevance.
  - High churn may reduce user trust.
  - High novelty may overwhelm users with unfamiliar options.
- The balance depends on business goals and cultural context.

Online A/B Testing

- **Definition:** Tests recommendations on real users to measure their actual reactions.
- **Process:**
  - Split users into groups.
  - Show different recommendation algorithms to each group.
  - Measure engagement, purchases, or clicks.
- **Benefits:**
  - Ensures recommendations work in real-world scenarios.
  - Avoids introducing unnecessary complexity.

Perceived Quality of Recommendations

- Users can explicitly rate the quality of recommendations.
- Challenges:
  - Users may confuse rating the item with rating the recommendation.
  - Requires additional user effort with little perceived benefit.
  - Data collected may be sparse and unclear.
- Best practice: Focus on user engagement and A/B tests instead.

Evaluation with python code

Evaluation

- The **MovieLens.py** file defines a **MovieLens** class that loads and parses the dataset.
- The **RecommenderMetrics.py** file defines a **RecommenderMetrics** class that calculates the following metrics:
  - MAE (Mean Absolute Error)
  - RMSE (Root Mean Squared Error)
  - GetTopN
  - HitRate
  - CumulativeHitRate (cHR)
  - RatingHitRate (rHR)
  - AverageReciprocalHitRank (ARHR)
  - UserCoverage
  - Diversity
  - Novelty
- The **TestMetrics.py** file uses these two classes to load the data, compute all metrics, and print the evaluation results.

Content-Based Filtering

Introduction to Content-Based Filtering

- Recommend items based solely on their own attributes.
- Useful when collaborative data is scarce or as a hybrid enhancement.
- Example: Suggest movies in the same genre and release period as ones a user enjoyed.

MovieLens Genre Data

- Each movie record lists its genres, e.g., Adventure—Comedy—Fantasy.
- There are 18 possible genres per movie.

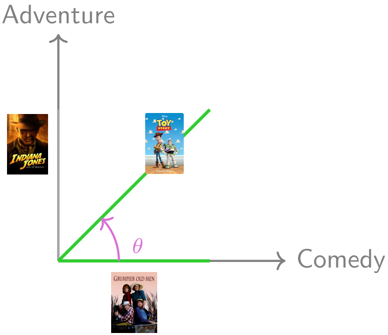
ID	Title	Genres
1	Toy Story (1995)	Adventure—Animation—Children—Comedy—Fantasy
2	Jumanji (1995)	Adventure—Children—Fantasy
3	Grumpier Old Men (1995)	Comedy—Romance
4	Waiting to Exhale (1995)	Comedy—Drama—Romance
5	Father of the Bride Part II (95)	Comedy

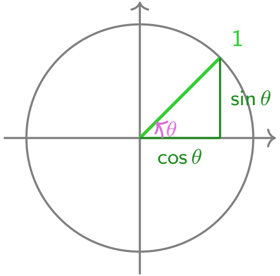
Genres as Binary Vectors

- Convert each genre list into an 18-dimensional $\{0,1\}$ vector.
- 1 indicates presence of a genre, 0 absence.

Movie	Action	Adventure	Animation	Children	Comedy	...	Romance	Sci-Fi	Thriller	Western
Toy Story	0	1	1	1	1	...	0	0	0	0
Jumanji	0	1	0	1	0	...	0	0	0	0
Grumpier Old Men	0	0	0	0	1	...	1	0	0	0

Cosine Similarity (2D Illustration)





Cosine Similarity: Measuring Genre-Based Movie Similarity

- This diagram explains how we compute similarity between movies using genre attributes.
- Each axis represents a binary genre feature — here simplified to just two: **Comedy** and **Adventure**.
- Movies are plotted as vectors in this genre space:
  - **Toy Story** and **Monty Python** are both comedy & adventure $\rightarrow$ vector (1,1).
  - **Grumpier Old Men** is only comedy $\rightarrow$ vector (1,0).
  - **Indiana Jones** is only adventure $\rightarrow$ vector (0,1).
- The angle θ between vectors reflects their similarity. The **cosine of that angle** gives us:

$$\text{cosine similarity} = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \cdot \|\vec{y}\|}$$

- Interpretation:
  - $\cos(0^\circ) = 1.0 \rightarrow$ Identical genre profile.
  - $\cos(90^\circ) = 0 \rightarrow$ No genres in common.
  - $\cos(45^\circ) \approx 0.71 \rightarrow$ Partial similarity.

Cosine Similarity Formula

$$\mathrm{CosSim}(x, y) = \frac{\sum_{i=1}^{D} x_i \, y_i}{\sqrt{\sum_{i=1}^{D} x_i^2} \, \sqrt{\sum_{i=1}^{D} y_i^2}}$$

- $x, y \in \{0, 1\}^D$: genre-vectors.
- Numerator: count of shared genres.
- Denominator: normalizes by vector lengths.

Python Code: Cosine Similarity

```
def computeGenreSimilarity(movie1, movie2, genres):
    # genres: dict movieID -> binary vector
    g1 = genres[movie1]
    g2 = genres[movie2]
    sumxx = sum(x*x for x in g1)
    sumyy = sum(y*y for y in g2)
    sumxy = sum(x*y for x,y in zip(g1,g2))
    return sumxy / math.sqrt(sumxx * sumyy)
```

Example Similarity Scores

- **Toy Story vs. Grumpier Old Men**:

$$\frac{1}{\sqrt{2} \times \sqrt{1}} \approx 0.707$$

- **Toy Story vs. Monty Python**: identical genres $\rightarrow$ 1.0.
- **Grumpier Old Men vs. Indiana Jones**: no overlap $\rightarrow$ 0.0.

Collaborative Filtering

Similarity Metrics

1. Adjusted Cosine Similarity

- Accounts for different user rating scales.
- Subtract each user's mean rating before computing cosine.

$$\mathrm{sim}_{\mathrm{adjCos}}(x, y) = \frac{\sum_{i \in I_{xy}} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i \in I_{xy}} (x_i - \bar{x})^2} \sqrt{\sum_{i \in I_{xy}} (y_i - \bar{y})^2}}$$

- x_i, y_i: ratings by users x, y on item i.
- $\bar{x}, \bar{y}$: average ratings of users x, y.
- I_{xy}: items co–rated by both users.

2. (Item-based) Pearson Similarity

- Similar to adjusted cosine, but mean–centered per item.
- Captures deviation from average item popularity.

$$\mathrm{sim}_{\mathrm{Pearson}}(x, y) = \frac{\sum_{i \in I_{xy}} (x_i - \bar{i})(y_i - \bar{i})}{\sqrt{\sum_{i \in I_{xy}} (x_i - \bar{i})^2} \sqrt{\sum_{i \in I_{xy}} (y_i - \bar{i})^2}}$$

- $\bar{i}$: mean rating of item i over all users.
- Useful for item–based collaborative filtering.

3. Spearman Rank Correlation (Concept)

- Like Pearson, but uses ranks instead of raw ratings.
- Convert each user's ratings into ranks.
- Measures monotonic relationship between users/items.
- Computationally intensive, less common in large-scale systems.

4. Mean Squared Difference (MSD)

- Direct measure of average squared rating difference.
- Less abstract than cosine; lower = more similar.

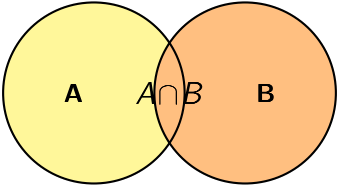
$$\mathrm{MSD}(x,y) = \frac{1}{|I_{xy}|}\sum_{i\in I_{xy}}(x_i - y_i)^2, \qquad \mathrm{sim}_{\mathrm{MSD}}(x,y) = \frac{1}{1+\mathrm{MSD}(x,y)}$$

- $\mathrm{MSD} \in [0,\infty)$, $\mathrm{sim}_{\mathrm{MSD}} \in (0,1]$.

5. Jaccard Similarity

- Ideal for binary / implicit feedback.
- Ignores rating values; only presence vs. absence.

$$\mathrm{sim}_{\mathrm{Jaccard}}(A,B) = \frac{|A \cap B|}{|A \cup B|}$$



Intersection

Union covers all shaded area

Summary of Similarities

- **Cosine / Adjusted Cosine:** angle–based, handles centering.
- **Pearson:** mean–centered per item, good for item–based CF.
- **Spearman:** rank–based, robust to non-linear scales.
- **MSD:** direct squared difference, easy intuition.
- **Jaccard:** set overlap, suited to implicit/binary data.

Cosine remains a solid default choice for most applications.

User-based Collaborative Filtering

Overview

- Find users similar to the target user by comparing their ratings.
- Recommend items that those similar users liked, which the target user hasn't seen.

Example: Ratings Table

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5	—	—	—
Ted	—	—	—	—	1
Ann	—	5	5	5	—

Step 1: Compute User–User Similarities

	Bob	**Ted**	**Ann**
Bob	1.0	0.0	1.0
Ted	0.0	1.0	0.0
Ann	1.0	0.0	1.0

Bob's top neighbors: Ann (1.0), Ted (0.0)

Step 2: Generate Candidates

1. Take Bob's top neighbor(s): Ann (similarity $= 1.0$).
2. Collect items Ann rated that Bob hasn't:
   - *Empire Strikes Back*, *Incredibles*

Step 3: Score Candidates

Normalize ratings to $[0, 1]$ (e.g. 5 stars $\mapsto$ 1.0), then weight by similarity:

$$\text{score}(i) = \sum_{u \in \{\text{Ann}\}} \left(\mathrm{sim}(u, \text{Bob}) \times \mathrm{normRating}(u, i)\right).$$

Here Ann's ratings $\to$ 1.0 and $\mathrm{sim}(\mathrm{Ann}, \mathrm{Bob}) = 1.0$, so

$$\text{score}(\textit{Empire}) = 1.0 \times 1.0 = 1.0, \quad \text{score}(\textit{Incredibles}) = 1.0.$$

Step 4: Filter & Recommend

- Remove items Bob has already rated (none of these).
- Both candidates tie at score 1.0; choose *Empire Strikes Back* as a recommendation.

User-Based CF Pipeline

1. **Build rating matrix:** users $\times$ items.
2. **Compute similarity matrix:** user–user (e.g. cosine).
3. **Neighbor lookup:** top-K similar users for target.
4. **Candidate gen.:** items neighbors rated $\setminus$ items target rated.
5. **Score candidates:** weighted by similarity & neighbor ratings.
6. **Filter & select:** exclude seen; sort by score; present Top-N.

Item-based Collaborative Filtering

Why Item-Based CF?

- Items are *stable* (a book remains a book), users' tastes may drift.
- Catalog size $\ll$ user base $\implies$ smaller similarity matrix.
- New-user friendliness: as soon as a user interacts with one item, you can recommend similar items.

Example: Ratings Matrix

	Bob	**Ted**	**Ann**
Indiana Jones	4		
Star Wars	5		5
Empire Strikes Back			5
Incredibles			5
Casablanca		1	

Cosine Similarity Between Items

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

- U_{ij}: users who rated both items i and j.
- Here, ratings are non-zero only in our tiny example, so many similarities collapse to 0 or 1.

Step 1: Compute Item–Item Similarities

	Indiana	**Star Wars**	**Empire SB**	**Incredibles**	**Casablanca**
Indiana Jones	1	1	0	0	0
Star Wars	1	1	1	1	0
Empire Strikes Back	0	1	1	1	0
Incredibles	0	1	1	1	0
Casablanca	0	0	0	0	1

Step 2: Recommend for Bob

1. Bob's known likes: *Star Wars*.
2. Look up all items similar to *Star Wars*:

$$\{\underline{\text{Indiana Jones}}, \underline{\text{Empire SB}}, \underline{\text{Incredibles}}\}$$

3. Score each by its similarity to *Star Wars* $\times$ Bob's rating of *Star Wars* (5).
4. Since $\mathrm{sim} = 1$ for all three and Bob's rating=5,

$$\text{score} = 1 \times 5 = 5.$$

5. Filter out items Bob already rated (none of these).
6. Final recommendations (tie): *Indiana Jones*, *Empire Strikes Back*, *Incredibles*.

Item-Based CF Pipeline

1. **Build ratings matrix:** items $\times$ users.
2. **Compute similarity matrix:** item–item (e.g. cosine).
3. **For target user:**
   - Gather items they've rated.
   - For each, fetch its top-K similar items.
   - Aggregate & score candidates by $\sum r_{u,i}\,\mathrm{sim}(i,\cdot)$.
   - Remove already-seen items.
   - Present Top-N.