Recommender Systems: Modern Libraries and Approaches

Greg Baker

Week 12 — 28th October 2025

MACQUARIE University

Today's journey

- What are recommender systems and why do they matter?
- **Two approaches**: Collaborative vs Content-Based filtering
- Content-based: Personalized TV show recommender with TF-IDF
- **K-Nearest Neighbors**: The simplest collaborative filtering
- Explicit vs implicit feedback: two different worlds
- Matrix factorization: the "alchemist" discovering hidden flavors
- Train-test splitting: why (user, item) pairs matter
- Building recommenders with modern Python libraries
- **scikit-learn**: for explicit ratings (stars, thumbs)
- **implicit**: for behavioral data (clicks, views, plays)
- Netflix vs YouTube: when optimization goals diverge
- Ethical considerations: engagement $\neq$ well-being

What is a recommender system?

Definition

A system that predicts what items a user might like and presents personalized suggestions, helping users navigate information overload.

Examples everywhere:

- **Entertainment**: Netflix (movies), Spotify (music), YouTube (videos)
- **E-commerce**: Amazon (products), eBay (items)
- **Social**: TikTok (content), LinkedIn (jobs), Tinder (people)
- **Information**: Google News (articles), Goodreads (books)

Why do we need recommender systems?

The paradox of choice:

- Netflix has 15,000+ titles
- Spotify has 100+ million songs
- Amazon has 350+ million products
- No human can evaluate all options

Business impact:

- 80% of Netflix viewing comes from recommendations
- YouTube: 70% of watch time, 20× increase over 3 years
- 35% of Amazon revenue driven by recommendations
- $1 million Netflix Prize (2006-2009) for 10% improvement

User benefit: Discover content you wouldn't find otherwise

Netflix: Optimizing for satisfaction

Netflix's approach:

- **Data**: Thumbs up/down, star ratings (historical), completion rates
- **Goal**: User *satisfaction* and retention
- **Metric**: Do users find content they enjoy?
- **Business model**: Monthly subscription

Optimization strategy:

- Recommend content users will *love* (not just watch)
- Balance popular hits with niche content
- Reduce churn by keeping subscribers happy
- Completion rate matters (did you finish it?)

Result: 80% of viewing from recommendations, low churn rate

YouTube: Optimizing for engagement

YouTube's approach:

- **Data**: Watch time, clicks, shares, session duration (all implicit!)
- **Goal**: Maximize *engagement* and watch time
- **Metric**: How long do users stay on the platform?
- **Business model**: Ad revenue (more viewing = more ads)

Optimization strategy:

- Recommend videos that keep you watching
- Autoplay next video before you can leave
- Optimize for click-through rate and session duration
- "Up next" based on what keeps viewers engaged

Result: Average session >40 minutes, massive daily active users

But at what cost?

The ethics of engagement optimization

The problem: Engagement $\neq$ User well-being

YouTube's algorithm has been criticized for:

- **Filter bubbles**: Only seeing content similar to past views
- **Rabbit holes**: Progressively more extreme content
- **Algorithmic radicalization**: Echo chambers amplify extreme views
- **Clickbait incentives**: Creators optimize for algorithm, not quality

Why this happens:

- Outrage and controversy drive engagement
- Algorithm learns: extreme content $\rightarrow$ longer watch time
- Positive feedback loop: more extreme $\rightarrow$ more views $\rightarrow$ more recommendations

Lesson: *What you optimize for matters!*

Ethical design principles

1. **Optimize for user satisfaction, not just engagement**
   - Measure long-term happiness, not just clicks
2. **Promote diversity and serendipity**
   - Don't just show more of the same
   - Help users discover new perspectives
3. **Provide user control and transparency**
   - Explain why items are recommended
   - Let users adjust preferences
4. **Monitor for harmful feedback loops**
   - Detect and break radicalization patterns
   - Audit for bias and unfairness
5. **Consider societal impact**
   - Your algorithm shapes what billions see
   - With great power comes great responsibility

Collaborative vs Content-Based Filtering

Content-Based

"Show me more like what I've liked"

Uses item *features*:

- Movie: genre, actors, director
- Song: tempo, key, artist
- Article: keywords, topics

No need for other users!

Collaborative Filtering

"Show me what similar users liked"

Uses *behavior patterns*:

- Find users like you
- See what they enjoyed
- Recommend those items

The "wisdom of the crowd"

This lecture: We'll cover *both* approaches and when to use each

Content-Based vs Collaborative: When to use each

Aspect	Content-Based	Collaborative
Data needs	Rich item features/metadata	User interaction history
Best when	• New items (no ratings yet) • Few users • Need explanations	• Lots of user data • Rich interaction patterns • Want serendipity
Discovers	Items similar to known preferences	Unexpected connections from crowd wisdom
Limitations	• Over-specialization • Limited novelty • Needs good metadata	• Cold start problem • Needs interaction data • Popular item bias
Examples	News recommendation, TF-IDF similarity	Netflix, Amazon, Spotify

In practice: Most systems use *hybrid* approaches combining both!

Content-Based Filtering: How it works

Core idea: Match item features to user preferences

The process:

1. **Build item profiles**: Extract features from each item
   - Movie: [Genre: Sci-Fi, Director: Nolan, Year: 2010, ...]
   - Article: [Keywords: AI, machine learning, neural networks, ...]
2. **Build user profile**: Aggregate features from items they liked
   - User watched 3 Nolan films → weight "Director: Nolan" highly
   - User reads AI articles → weight "AI" keywords highly
3. **Match**: Compute similarity between user profile and item profiles
   - Use cosine similarity, dot product, or other similarity metrics
   - Rank items by similarity score

Key advantage: Works even when no other users exist!

Demo: Personalized TV show recommender

A hybrid approach: Content features + Personalized learning

The approach:

1. Use **TF-IDF on TV show descriptions** (83k+ shows from TVMaze)
   - Each show represented as a vector of word importance
   - Similar content = similar vectors
2. Ask you to **rate a few popular shows** (1-5 scale)
3. **Train a regression model** that learns YOUR preferences
   - Discovers which content features predict your ratings
   - E.g., if you rate sci-fi highly, learns "space", "future" = high score
4. **Predict ratings** for all other shows and recommend top-N

Why hybrid?

- Content-based features (TF-IDF) work for any show with description
- Personalized model learns *your* unique taste
- Works immediately — no need to wait for millions of users!

Step 1: TF-IDF on TV show descriptions

Convert text descriptions into numerical vectors

```
print(f"Loaded {len(shows):,} TV shows with descriptions")
print()

# Create TF-IDF vectors from show descriptions
print("Step 1: Creating TF-IDF vectors from show
    descriptions...")
print("          (This represents each show as a vector of
    word importance)")
tfidf = TfidfVectorizer(
    max_features=500,       # Use top 500 most important
        words
    stop_words='english',  # Remove common words like
        'the', 'a', etc.
    min_df=2               # Word must appear in at least 2
        shows
)
```

Step 2: Train your personalized model

Learn which content features predict YOUR ratings

```
# Get TF-IDF features for the shows the user rated
X_train = tfidf_matrix[rated_shows]
y_train = np.array(user_ratings)

# Train a Ridge regression model
model = Ridge(alpha=1.0)
model.fit(X_train, y_train)

print("Model trained!")
print()
```

The magic:

- Ridge regression learns: `your_rating = f(TF-IDF features)`
- If you rate sci-fi high, it learns "space", "future" predict high scores
- Then predicts your ratings for 83,000+ other shows!

K-NN: The simplest collaborative filtering approach

Core idea: "Find similar users (or items), recommend what they liked"

Two flavors:

- **User-based**: Find users with similar taste, recommend what they liked
  - "Users who rated movies like you also enjoyed..."
  - Calculate user-user similarity based on rating patterns
- **Item-based**: Find items similar to what you liked
  - "Movies similar to ones you rated highly..."
  - Calculate item-item similarity based on who rated them
  - Amazon's "Customers who bought X also bought Y" uses this!

Similarity metrics:

- Cosine similarity (most common for sparse ratings)
- Pearson correlation (accounts for rating scale differences)
- Euclidean distance

User-based vs Item-based k-NN

Aspect	User-Based	Item-Based
Finds	Users with similar rating patterns	Items with similar rating patterns
Question	"Who are users like me?"	"What items are like this one?"
Recommends	Items that similar users liked	Items similar to ones you liked
Best when	Few users, many items (e.g., new startup)	Many users, few items (e.g., Amazon, Netflix)
Stability	User preferences change frequently	Item similarities are more stable over time
Scalability	Recompute when users change	Can precompute item similarities
Example	Music discovery based on listeners with your taste	Amazon: "bought X also bought Y"

K-NN with scikit-learn: NearestNeighbors

Using sklearn's NearestNeighbors class

```
# Use item-based approach (find similar movies)
# Fill NaN with 0 for KNN (we'll only use non-zero values
    for similarity)
train_matrix_filled = train_matrix.fillna(0)

# Create sparse matrix for efficiency
train_sparse = csr_matrix(train_matrix_filled.T.values)  #
    Transpose for item-item

# Fit KNN model (cosine similarity)
knn_model = NearestNeighbors(metric='cosine',
    algorithm='brute', n_neighbors=20)
knn_model.fit(train_sparse)

print(f"Trained KNN with {knn_model.n_neighbors} neighbors")
print("Using cosine similarity metric\n")
```

Finding neighbors with .kneighbors() (part 1)

Step 1: Find the k most similar items

```
def predict_knn(user_id, movie_id, train_matrix, knn_model,
    k=10):
    """Predict rating using item-based KNN collaborative
        filtering."""

    # Check if movie exists in training data
    if movie_id not in train_matrix.columns:
        return train_data['rating'].mean()

    # Get the movie's index
    movie_idx = train_matrix.columns.get_loc(movie_id)

    # Find k nearest neighbor movies
    movie_vector = train_matrix_filled.iloc[:,
        movie_idx].values.reshape(1, -1)
    distances, indices =
```

Finding neighbors with .kneighbors() (part 2)

Step 2: Use neighbor ratings to predict target rating

```
    # Get ratings from the user for similar movies
    user_ratings = []
    similarities = []

    for idx in similar_movies:
        similar_movie_id = train_matrix.columns[idx]
        if user_id in train_matrix.index and
            similar_movie_id in train_matrix.columns:
            rating = train_matrix.loc[user_id,
                similar_movie_id]
            if not pd.isna(rating) and rating > 0:
                user_ratings.append(rating)
                similarities.append(1 -
                    distances.flatten()[list(similar_movi
                    + 1])
```

K-NN: Advantages and limitations

Advantages:

- **Intuitive**: Easy to explain and understand
- **Simple to implement**: Just similarity + averaging
- **No training phase**: Just compute similarities on the fly
- **Flexible**: Can use different similarity metrics easily
- **Interpretable**: "Because users like you enjoyed it"

Limitations:

- **Doesn't scale well**: Must compute many similarities
- **Sparsity problem**: Hard to find good neighbors with sparse data
- **Cold start**: Can't recommend for new users/items with no ratings
- **Popular item bias**: Tends to recommend popular items
- **Computation cost**: Finding neighbors is expensive for large datasets

Next step: Matrix factorization solves many of these limitations!

Two types of user signals

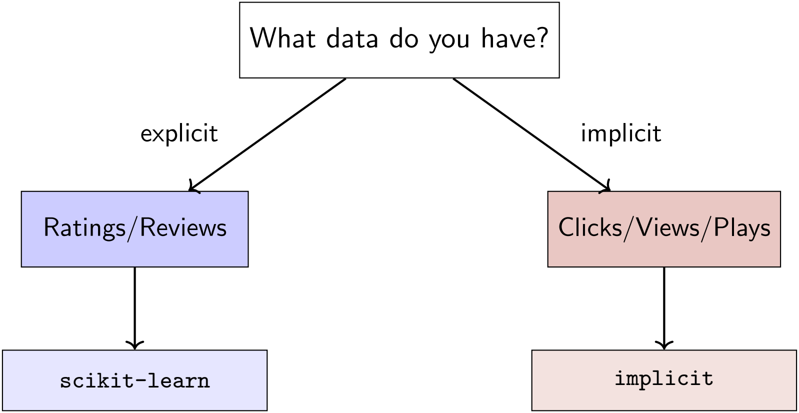
Explicit Feedback: User directly tells you their preference

- Star ratings (1-5 stars), thumbs up/down
- Numerical scores, like/dislike buttons
- **Pros**: Strong signal, clear preference
- **Cons**: Sparse (users rarely rate), requires effort
- **Examples**: MovieLens ratings, Netflix thumbs, Yelp reviews

Implicit Feedback: Inferred from behavior

- Clicks, purchases, views, watch time, plays
- Search queries, browsing history
- **Pros**: Abundant data, no user effort
- **Cons**: Noisy signal (watched $\neq$ enjoyed)
- **Examples**: YouTube watch history, Amazon purchases, Spotify plays

Which type of data do you have?



Linear algebra in 90 seconds

- A matrix is a grid of numbers, e.g., $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
- A vector is a column or row of numbers (a special case of a matrix), e.g., $\begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix}$
- You can multiply matrices together, but $AB \neq BA$
- A 2x3 matrix takes a 2D object and embeds it in 3D space
- A 3x2 matrix projects a 3D object down to 2D space (like a shadow)
- (There are higher dimensional equivalents of this)
- Other matrices correspond to rotations, reflections, shifts and other transformations of high dimensional data

Introducing the @ operator

Note: The `@` operator performs matrix multiplication in Python

- Introduced in Python 3.5 (PEP 465)
- `user_factors @ movie_factors.T` multiplies the matrices
- Equivalent to `numpy.dot(user_factors, movie_factors.T)`
- Much more readable for linear algebra operations

The user-item matrix

The fundamental data structure:

	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	$\cdots$
Alice	5	?	4	?	3	$\cdots$
Bob	?	3	?	5	?	$\cdots$
Carol	4	?	5	?	?	$\cdots$
Dave	?	4	?	2	5	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

- Rows = users, Columns = items, Values = ratings/interactions
- Most cells are empty (sparsity problem)
- **Goal**: Fill in the "?" cells — predict unknown preferences!

Matrix Factorization: The core idea

Instead of working with the huge sparse matrix directly...

Discover hidden "latent factors":

- Maybe Factor 1 = "Blockbuster vs Indie"
- Maybe Factor 2 = "Action vs Romance"
- Maybe Factor 3 = "Serious vs Lighthearted"
- Maybe Factor 4 = "New vs Classic"

Key insight: We can learn these factors automatically

$$\underbrace{\text{User-Item Matrix}}_{n\_users \times n\_items} \approx \underbrace{\text{User Factors}}_{n\_users \times k} \times \underbrace{\text{Item Factors}}_{k \times n\_items}$$

where k (e.g., 20-100) $\ll$ n_items (e.g., 10,000)

The Alchemist Analogy

Imagine: You have a vast cookbook with thousands of recipes and thousands of picky eaters. Most pages are blank.

Matrix Factorization is like a master alchemist:

1. **Discovers fundamental "flavor profiles"**
   - Spiciness, sweetness, crunchiness, umami, sourness
   - Nobody tells the alchemist these — they're learned from data!
2. **Deconstructs each recipe into flavor signature**
   - Recipe 42: [high spice, low sweet, medium umami]
3. **Profiles each eater's palate**
   - Alice: [loves spice, hates sweet, neutral on umami]
4. **Predicts compatibility**
   - Will Alice like Recipe 42? → Take dot product!
   - rating ≈ user_vector · item_vector

Why "Truncated" SVD?

You may have seen SVD in linear algebra:

Regular SVD decomposes: $A = U\Sigma V^T$ (keeps *all* dimensions)

TruncatedSVD keeps only top k dimensions:

- "Truncated" = Keep only the k most important factors
- Dimensionality reduction: 1000s of movies $\rightarrow$ 50 latent factors
- Captures most of the patterns with far fewer numbers
- Much faster and prevents overfitting

Example:

- MovieLens: 943 users $\times$ 1682 movies = 1,586,126 cells
- TruncatedSVD with $k = 50$: $943 \times 50 + 50 \times 1682 = 131{,}250$ values
- 12$\times$ fewer numbers, captures main patterns!

In scikit-learn: `TruncatedSVD(n_components=50)`

Truncated SVD: A concrete example

Setup: A tiny 4-document, 5-term matrix

$$X = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 2 \end{bmatrix}$$

docs d_1, d_2, d_3, d_4
terms t_1, t_2, t_3, t_4, t_5

Two clear themes:

- d_1, d_2 use t_1, t_2 (first theme)
- d_3, d_4 use t_3, t_4, t_5 (second theme)

Full SVD: $X = U\Sigma V^T$ where U is 4×4, Σ has 4 singular values

- $\sigma_1 = 3.000$, $\sigma_2 = 2.613$, $\sigma_3 = 1.082$, $\sigma_4 = 1.000$
- Question: Can we capture most patterns with fewer dimensions?

Truncating to $k = 2$: Keeping top factors

Truncated SVD: Keep only top $k = 2$ singular values/vectors

$$U_2 = \begin{bmatrix} 0.707 & 0 \\ 0.707 & 0 \\ 0 & 0.383 \\ 0 & 0.924 \end{bmatrix}, \quad \Sigma_2 = \begin{bmatrix} 3.0 & 0 \\ 0 & 2.61 \end{bmatrix}, V_2^T = \begin{bmatrix} 0.707 & 0.707 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0.707 \end{bmatrix}$$

Rank-2 approximation: $X_{(2)} = U_2 \Sigma_2 V_2^T$

$$X_{(2)} \approx \begin{bmatrix} 1.500 & 1.500 & 0 & 0 & 0 \\ 1.500 & 1.500 & 0 & 0 & 0 \\ 0 & 0 & 0.500 & 0.500 & 0.707 \\ 0 & 0 & 1.207 & 1.207 & 1.707 \end{bmatrix}$$

How good? Explained variance: $\frac{3.000^2 + 2.613^2}{3.000^2 + 2.613^2 + 1.082^2 + 1.000^2} \approx 87.9\%$

What truncated SVD discovers: 2D embeddings

Two latent axes discovered:

1. Axis 1: explains t_1/t_2 and d_1/d_2 (first theme)
2. Axis 2: explains $t_3/t_4/t_5$ and d_3/d_4 (second theme)

Document embeddings: $Z_{\text{docs}} = U_2\Sigma_2$ (or equivalently XV_2)

$$Z_{\text{docs}} \approx \begin{bmatrix} 2.121 & 0 \\ 2.121 & 0 \\ 0 & 1.000 \\ 0 & 2.414 \end{bmatrix} \qquad \begin{aligned} d_1, d_2 &: (2.121, 0) \\ d_3 &: (0, 1.000) \\ d_4 &: (0, 2.414) \end{aligned}$$

Term embeddings: $Z_{\text{terms}} = V_2\Sigma_2$

$$Z_{\text{terms}} \approx \begin{bmatrix} 2.121 & 0 \\ 2.121 & 0 \\ 0 & 1.307 \end{bmatrix} \qquad \begin{aligned} t_1, t_2 &: (2.121, 0) \\ t_3, t_4 &: (0, 1.307) \end{aligned}$$

Projecting new documents into the space

Given a new document x_{new} (1×5 vector), embed it:

$$z_{\text{new}} = x_{\text{new}} V_2$$

Example: New doc $x_{\text{new}} = [0, 0, 1, 0, 1]$ (uses t_3 and t_5)

$$z_{\text{new}} = [0, 0, 1, 0, 1] \begin{bmatrix} 0.707 & 0 \\ 0.707 & 0 \\ 0 & 0.500 \\ 0 & 0.500 \\ 0 & 0.707 \end{bmatrix} = [\,0,\ 1.207\,]$$

Interpretation: Lands between d_3 (0,1.000) and d_4 (0,2.414) — exactly as expected!

Model's reconstruction: $\hat{x}_{\text{new}} = z_{\text{new}} V_2^T \approx [0,\ 0,\ 0.604,\ 0.604,\ 0.854]$

Why splitting is different for recommenders

In standard ML: Split data points randomly

- Classification: Split rows (samples)
- Image recognition: Split images
- Simple and straightforward

In recommender systems: The atomic unit is a (user, item) pair!

Key Insight

We want to test the model's ability to predict *unseen interactions* between *known users* and *known items*.

Three possible approaches:

1. Split by users (test on new users)
2. Split by items (test on new items)
3. Split by (user, item) pairs ← usually what we want!

Three ways to split: What are we testing?

Split Method	What It Tests	When To Use
By Users	Can we recommend to users we've never seen before?	Cold-start problem: new user signup
By Items	Can we recommend items we've never seen before?	Cold-start problem: new product launch
By Pairs	Can we predict interactions between known users & items?	General evaluation: how well does the model generalize?

Important

Splitting by pairs ensures every user and every item appears in *both* train and test sets — but the specific interactions differ.

Most common: Split by pairs (general case evaluation)

Time-based splitting: The production reality

In production, recommenders predict the future!

Time-based split:

- **Train**: All interactions before time t (e.g., before 2025)
- **Test**: All interactions after time t (e.g., during 2025)
- Better reflects real-world deployment
- No “data leakage” from the future

Example scenarios:

- Netflix: Train on 2024 viewing, test on January 2025
- E-commerce: Train on Q1-Q3, test on Q4 holiday season
- Music streaming: Train on weekdays, test on weekends

Trade-off: Time-based is more realistic but less data in test set

What about cross-validation?

In standard ML: K-fold cross-validation is straightforward

- Shuffle data, split into k folds
- Train on $k-1$ folds, test on remaining fold
- Repeat k times, average results

In recommender systems: It's more complicated!

The Problem

User-item interactions are *not independent*. Standard k-fold CV can leak information and produce inflated accuracy scores.

Three types of folds you might use:

- **User-based folds**: Hold out entire users (test cold-start)
- **Item-based folds**: Hold out entire items (test new products)
- **Interaction-based folds**: Hold out random pairs (test general case)

The Netflix seasons problem

Why random splits can be misleading:

Scenario: You've watched Breaking Bad on Netflix

- Training set: Seasons 1, 2, 3, 5, 6
- Test set: Season 4

The problem:

- Predicting Season 4 is *trivially easy* — it's just interpolation!
- The model fills the obvious gap in a series you've already committed to
- High accuracy here doesn't mean the model is good

What we really want to know:

- Should we recommend Season 1 of a *new* show?
- Will you like Season 7 when it comes out?
- Can we predict your behavior on *future* content?

The feedback loop problem

Recommenders change user behavior:

1. System recommends items based on your past behavior
2. You watch/buy/click those recommendations
3. Your preferences shift based on what you consumed
4. Future recommendations trained on this changed behavior
5. The cycle repeats. . .

This is called: Algorithmic confounding or feedback bias

Implication

The user *after* 100 algorithmic recommendations is not the same person as before. Static evaluation can never fully capture this dynamic.

Why companies run A/B tests:

Using scikit-learn for recommenders

scikit-learn: Python's machine learning library works for recommenders too!

Key components we'll use:

- **TruncatedSVD**: Matrix factorization for collaborative filtering
- **NearestNeighbors**: Find similar users/items (K-NN approach)
- **train_test_split**: Split data for evaluation
- Standard metrics: RMSE, MAE from sklearn.metrics
- Works seamlessly with pandas DataFrames

Installation:

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

Loading data with pandas

```
# Download MovieLens 100K if not already present
if not os.path.exists('ml-100k'):
    print("Downloading MovieLens 100K dataset...")
    url = 'https://files.grouplens.org/datasets/moviele
    urllib.request.urlretrieve(url, 'ml-100k.zip')

    print("Extracting dataset...")
    with zipfile.ZipFile('ml-100k.zip', 'r') as zip_ref:
        zip_ref.extractall('.')
    print("Dataset downloaded and extracted!\n")
else:
```

Dataset statistics

```
print(f"Number of ratings: {len(ratings):,}")
print(f"Number of users:
    {ratings['user_id'].nunique()}")
print(f"Number of movies:
    {ratings['movie_id'].nunique()}")
print(f"Rating scale: {ratings['rating'].min()}
    to {ratings['rating'].max()}")
print(f"Average rating:
    {ratings['rating'].mean():.3f}")

# Calculate sparsity
n_users = ratings['user_id'].nunique()
n_movies = ratings['movie_id'].nunique()
sparsity = 1 - (len(ratings) / (n_users *
    n_movies))
```

Training an SVD model

```
# Apply SVD
n_factors = 50
svd = TruncatedSVD(n_components=n_factors,
    random_state=42)
user_factors =
    svd.fit_transform(train_matrix_svd)
movie_factors = svd.components_.T

print(f"Learned {n_factors} latent factors")
print(f"User factors shape:
    {user_factors.shape}")
print(f"Movie factors shape:
    {movie_factors.shape}")
print(f"Explained variance ratio:
    {svd.explained_variance_ratio_.sum():.3f}\n")
```

SVD Results

Test set evaluation results:

- **RMSE**: ∼0.95
- **MAE**: ∼0.75

What does this mean?

- On average, predictions are off by ∼0.75 stars
- On a 1-5 star scale, this is pretty good!
- Compare to: always predicting mean (RMSE ∼1.06)

The algorithm discovered 50 latent factors describing user taste!

Making predictions

```
# Reconstruct ratings matrix
predicted_ratings = user_factors @
    movie_factors.T
predicted_ratings_df =
    pd.DataFrame(predicted_ratings,
                                                  index=train_ma
                                                  columns=train_

# Clip predictions to valid range [1, 5]
predicted_ratings_df =
    predicted_ratings_df.clip(1, 5)
```

Example output:

```
SVD Results: RMSE: 0.9531, MAE: 0.7522
```

Generating top-N recommendations

```python
def get_top_n_recommendations(user_id, predicted_ratings_df, train_matrix, n=10):
    """Get top-N movie recommendations for a user."""

    if user_id not in predicted_ratings_df.index:
        return []

    # Get predicted ratings for this user
    user_predictions = predicted_ratings_df.loc[user_id]

    # Remove movies the user has already rated
    if user_id in train_matrix.index:
        rated_movies = train_matrix.loc[user_id]
        rated_movies = rated_movies[rated_movies > 0].index
        user_predictions = user_predictions.drop(rated_movies, errors='ignore')

    # Sort and get top N
    top_movies = user_predictions.sort_values(ascending=False).head(n)

    return list(zip(top_movies.index, top_movies.values))
```

Introducing the implicit library

implicit: Python library optimized for implicit feedback

Why a separate library?

- Implicit data behaves differently than explicit ratings
- "Not interacted" $\neq$ "Dislike" (it's just unknown!)
- Need different algorithms: ALS, BPR, Logistic MF
- Scalability is critical (millions of interactions)

Key features:

- Fast C++ implementations (via Cython)
- Multi-threaded (uses all CPU cores)
- GPU support for even faster training
- Algorithms: ALS, BPR, item-item KNN with TF-IDF/BM25

Installation:

Data preparation for implicit

Key difference: Requires sparse matrix format

```

# Evaluate on test set
test_sample = test_data.sample(min(1000, len(test_data)), random_state=42)
svd_predictions = []

for _, row in test_sample.iterrows():
    if row['user_id'] in predicted_ratings_df.index and row['movie_id'] in
         predicted_ratings_df.columns:
        pred = predicted_ratings_df.loc[row['user_id'], row['movie_id']]
        svd_predictions.append(pred)
    else:
        svd_predictions.append(train_data['rating'].mean())

rmse_svd = np.sqrt(mean_squared_error(test_sample['rating'], svd_predictions))
mae_svd = mean_absolute_error(test_sample['rating'], svd_predictions)

print(f"Training completed in {training_time:.2f} seconds")
print(f"\nExplicit Feedback Results (SVD):")
print(f"  RMSE: {rmse_svd:.4f}")
print(f"  MAE:  {mae_svd:.4f}")
```

Alternating Least Squares (ALS)

ALS: The workhorse algorithm for implicit feedback

The "Two-Artist Portrait" Analogy:

1. Two artists recreate a painting (the user-item matrix)
2. Artist A draws vertical lines (user factors)
3. Artist B draws horizontal lines (item factors)
4. They take turns:
    - Artist A holds still → Artist B adjusts to match
    - Artist B holds still → Artist A adjusts to match
5. They *alternate* improving until convergence

Why "alternating"? Can't optimize both at once, so we alternate!

This makes a hard problem tractable.

What is ALS minimizing?

The objective: Weighted squared reconstruction error

$$\text{minimize} \sum_{u,i} c_{ui} \cdot (p_{ui} - \mathbf{x}_u^T \mathbf{y}_i)^2 + \text{regularization}$$

Where:

- p_{ui} = preference (1 if user u interacted with item i, 0 otherwise)
- c_{ui} = confidence weight (higher for actual interactions)
- $\mathbf{x}_u^T \mathbf{y}_i$ = predicted value (dot product of latent factors)

Key insight: Model *confidence in preferences*, not exact ratings

- Observed interactions get high confidence weights
- Unobserved $\neq$ dislike (just unknown, low confidence)
- Different from explicit ratings where we predict 1-5 stars

Training ALS model

```
example_movie = 302

if example_user in predicted_ratings_df.index
    and example_movie in
    predicted_ratings_df.columns:
     pred_rating =
        predicted_ratings_df.loc[example_user,
        example_movie]
     print(f"Example: User {example_user} on
        Movie {example_movie}")
     print(f"  Predicted rating:
        {pred_rating:.2f} stars")
else:
     print(f"Example: User {example_user} on
        Movie {example_movie}")
```

BPR: A different approach to implicit feedback

Bayesian Personalized Ranking (BPR): Optimizes for ranking order

Key difference from ALS:

- **ALS**: Predicts interaction strength (pointwise)
  - "User will rate this item 4.2"
  - Tries to predict actual values
- **BPR**: Learns to rank items (pairwise)
  - "User prefers item A over item B"
  - Only cares about relative order

Why this matters for implicit feedback:

- We don't know actual preference strength (no ratings!)
- But we *do* know: clicked items > unclicked items
- BPR directly optimizes this ranking property

How BPR works: Pairwise comparisons

The BPR training approach:

For each user:

1. Take an item they **interacted with** (positive: i)
2. Take an item they **didn't interact with** (negative: j)
3. Create training triple: (user u, positive i, negative j)

Loss function: Maximize $\hat{x}_{ui} - \hat{x}_{uj}$

- $\hat{x}_{ui}$: predicted score for positive item
- $\hat{x}_{uj}$: predicted score for negative item
- Goal: make positive items score higher than negative items

Example:

- User watched "Breaking Bad" $\rightarrow$ positive
- User never clicked "Teletubbies" $\rightarrow$ negative (sampled)
- Train so: score("Breaking Bad") $>$ score("Teletubbies")

BPR vs ALS: When to use each

Aspect	ALS	BPR
Optimizes	Reconstruction error (predict values)	Pairwise ranking (relative order)
Training	Alternating optimization (fast)	Stochastic gradient descent (slower)
Best for	Large-scale problems, production systems	When ranking quality is critical
Scalability	Very fast, parallel	Slower, more iterations needed
Use case	Music streaming (plays), e-commerce (purchases)	Search results, recommendations where order matters

In practice:

- Start with ALS (faster, simpler)

Using BPR in implicit library

Code is almost identical to ALS:

```
from implicit.bpr import BayesianPersonalizedRanking

# Create model
model = BayesianPersonalizedRanking(
    factors=50,              # Number of latent factors
    learning_rate=0.01,      # SGD learning rate
    regularization=0.01,     # Regularization strength
    iterations=100           # Training iterations
)

# Train (item-user matrix, transposed!)
model.fit(item_user_sparse)

# Get recommendations (same API as ALS)
recommendations = model.recommend(
    userid=user_id,
```

Evaluation with ranking metrics

```
print("PART 2: IMPLICIT FEEDBACK - Modeling
   Viewing Behavior")
print("="*80)
print("Data: Binary interactions (user watched
   movie or not)")
print("Goal: Rank movies by likelihood user will
   watch them")
print("Method: ALS (Alternating Least Squares)
   for implicit data")
print("="*80)
print()
```

Precision@10: What % of top-10 recommendations are relevant?

Different from RMSE! We care about *ranking*, not exact scores.

Getting recommendations

```
import implicit.als
import implicit.evaluation

# Suppress OpenMP warnings
os.environ['OMP_NUM_THREADS'] = '1'

# Convert to implicit data: any rating >= 4 is
    considered a "positive interaction"
# This simulates implicit feedback (like
    watching a movie)
print("Converting MovieLens to implicit
    format...")
implicit_df = ratings_df[ratings_df['rating'] >=
    4].copy()
implicit_df['interaction'] = 1  # Binary:
```

Side-by-side comparison

Aspect	Explicit (sklearn)	Implicit (implicit)
Data	Star ratings (1-5)	Binary interactions (0/1)
Signal	Strong (user opinion)	Weak (inferred)
Volume	Sparse	Dense
Goal	Predict rating value	Rank by preference
Output	Estimated rating (3.5 stars)	Confidence score (0.85)
Evaluation	RMSE, MAE	Precision@K, MAP@K
Algorithm	SVD, KNN	ALS, BPR
Cold Start	Severe (no ratings)	Better (track views)

What we covered today

Key concepts:

- **Two filtering approaches**: Content-based vs Collaborative
- **Content-based**: Uses item features (TF-IDF on descriptions/metadata)
- **K-NN collaborative filtering**: Find similar users/items, simple but intuitive
- **Two data types**: Explicit ratings vs Implicit interactions
- **Matrix factorization**: Discovering hidden taste dimensions
- **Analogies**: The alchemist (SVD), two artists (ALS)
- **Train-test splitting**: Split (user, item) pairs, not users or items

Two Python libraries:

- **scikit-learn**: For explicit ratings (MovieLens, Jester)
- **implicit**: For behavioral data (clicks, views, plays)

Real-world lessons:

Choosing the right approach

What data do you have?

If you have. . .	**Use. . .**
Star ratings, reviews, thumbs	`scikit-learn` (SVD, KNN)
Clicks, views, plays, purchases	`implicit` (ALS, BPR)
Both!	Hybrid system (best of both)
Item features (genres, tags)	Content-based filtering
Neither (cold start)	Popularity baseline + ask user

Remember: The best recommender system is the one that *helps users*, not just the one with the lowest RMSE!