COMP2200/COMP6200 Practical Exercise (Week 12): Recommender Systems

School of Computing

LEU Survey

You have been invited to participate in a Learner Experience of Unit (LEU) survey for COMP2200/COMP6200-Data Science. You will have already received an individual email invitation and link to the survey. You can also access the survey on the 'Student Feedback Surveys' block of your iLearn homepage (not the unit homepage). Take some time to complete it now.

Preparation

Choose your environment: either run locally with `uv` or use Google Colab.

1. **Local with `uv`**:
   (a) Install `uv`.
   MacOS/Linux: `curl -LsSf https://astral.sh/uv/install.sh | sh`
   Windows: `powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"`
   (b) Create a folder for this practical (for example, `week12-prac`) and open a terminal in it.
   (c) Set up your environment:
   ```
   uv init
   uv add implicit pandas numpy matplotlib scipy scikit-learn
   ```
   (d) Launch Jupyter with `uv run --with jupyter jupyter lab` (or `... notebook`).
2. **Google Colab**:
   (a) Open Colab and create a new notebook.
   (b) Install packages with `!pip install implicit pandas numpy matplotlib scipy scikit-learn`.

Part A — Joke Recommender with Jester Dataset (60 min)

In this part, you'll build a joke recommendation system using the Jester dataset. Jester contains ratings of jokes from real users on a continuous scale from -10 (worst joke ever) to +10 (best joke ever). This is a classic example of **explicit feedback** — users directly tell us their opinion.

A1. Import required libraries

Import the necessary Python modules for this practical.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import TruncatedSVD
from sklearn.neighbors import NearestNeighbors
from sklearn.model_selection import train_test_split
from sklearn.dummy import DummyRegressor
from sklearn.metrics import root_mean_squared_error, mean_absolute_error
```

A2. Load the Jester dataset

Load the Jester dataset from the local file and examine its shape.

```
print("Loading Jester dataset...")
jester_raw = pd.read_csv('jester-data-1.csv', header=None, usecols=range(1, 101))
print("Dataset loaded!")

# Get dimensions
n_users, n_jokes = jester_raw.shape
print(f"Number of users: {n_users:,}")
print(f"Number of jokes: {n_jokes}")
print(f"\nSample of raw data (99 means 'not rated'):")
print(jester_raw.head())
```

A3. Explore the dataset statistics

Convert from wide to long format using pandas `melt`.

```
# Convert from wide to long format
# Columns are jokes (0-99), rows are users, 99 means "not rated"

# Add user ID column from the index
jester_raw['user'] = jester_raw.index

ratings_df = jester_raw.melt(
    id_vars='user',
    var_name='item',
    value_name='rating'
)

# Filter out missing ratings (99 means not rated)
ratings_df = ratings_df[ratings_df['rating'] != 99]

# Convert item to 0-indexed integers (columns are 1-100, need 0-99 for indexing)
ratings_df['item'] = ratings_df['item'].astype(int) - 1

# Print statistics
n_ratings = len(ratings_df)

print(f"Number of ratings: {n_ratings:,}")
print(f"Rating scale: {ratings_df['rating'].min():.1f} to {ratings_df['rating'].max():.1f}")
print(f"Global mean rating: {ratings_df['rating'].mean():.2f}")
```

Calculate the matrix sparsity (what percentage of cells are empty):

```
total_possible = n_users * n_jokes
sparsity = 1 - (n_ratings / total_possible)
print(f"Matrix sparsity: {sparsity:.1%}")
```

Question: The Jester dataset has about 27% sparsity. Typical movie ratings datasets (like MovieLens) have 93–95% sparsity. Why is Jester so much less sparse? What does this tell you about how the Jester data was collected compared to movie ratings?

A4. Train a baseline model

Let's start with a baseline: a simple model that always predicts the mean rating.

```
# Split into train/test sets
train_df, test_df = train_test_split(ratings_df, test_size=1000, random_state=42)

print(f"Training set: {len(train_df):,}")
print(f"Test set: {len(test_df):,}")
```

```
# Train baseline model (always predicts mean rating)
baseline = DummyRegressor(strategy='mean')
baseline.fit(train_df[['user', 'item']], train_df['rating'])
y_pred = baseline.predict(test_df[['user', 'item']])

# Evaluate
rmse_baseline = root_mean_squared_error(test_df['rating'], y_pred)
mae_baseline = mean_absolute_error(test_df['rating'], y_pred)

print(f"\nBaseline Results:")
print(f"  RMSE: {rmse_baseline:.4f}")
print(f"  MAE:  {mae_baseline:.4f}")
```

A5. Train an SVD model

SVD (Singular Value Decomposition) is a matrix factorization technique that finds latent factors for users and jokes. The idea is that each user and each joke can be represented as a vector in a lower-dimensional "latent space". Users and jokes that are similar will be close together in this space.

Step 1: Create the user-item matrix

First, create a user-item matrix from the training data:

```
# Create user-item matrix from training data
# Rows = users, columns = jokes, cells = ratings
user_item_matrix = train_df.pivot_table(
    index='user',
    columns='item',
    values='rating',
    fill_value=0  # Fill missing ratings with 0
)

print(f"User-item matrix shape: {user_item_matrix.shape}")
```

This `pivot_table` operation converts our long-format dataframe (one row per rating) into a wide-format matrix where each row represents a user and each column represents a joke. Missing ratings are filled with 0.

Step 2: Apply SVD decomposition

Now apply SVD to decompose this matrix into user and item factors:

```
# Apply SVD using scikit-learn's TruncatedSVD
# n_factors controls the dimensionality of the latent space
# More factors = more expressive but risk of overfitting
n_factors = 20

svd_model = TruncatedSVD(n_components=n_factors, random_state=42)
user_factors = svd_model.fit_transform(user_item_matrix)
item_factors = svd_model.components_.T

print(f"User factors shape: {user_factors.shape}")
print(f"Item factors shape: {item_factors.shape}")
```

SVD decomposes the user-item matrix into two smaller matrices: `user_factors` (users $\times$ latent factors) and `item_factors` (jokes $\times$ latent factors). Each user is now represented by 20 numbers, and each joke is represented by 20 numbers.

Step 3: Reconstruct the ratings matrix

Reconstruct the full ratings matrix and make predictions:

```
# Reconstruct the full ratings matrix by multiplying user and item factors
# Matrix multiplication: (users x factors) @ (factors x items) = (users x items)
# This gives us predicted ratings for all user-item pairs
predicted_ratings = user_factors @ item_factors.T

print(f"Predicted ratings matrix shape: {predicted_ratings.shape}")
```

The reconstructed matrix contains predicted ratings for every user-joke pair, even those we never observed in the training data. This is the power of collaborative filtering — it can predict ratings for unseen user-joke combinations!

Step 4: Evaluate on test set

Now evaluate the model on the test set:

```
# Get predictions for our test set
# For each test rating, look up the predicted value for that (user, item) pair
test_users = test_df['user'].values
test_items = test_df['item'].values
svd_predictions = predicted_ratings[test_users, test_items]

# Calculate error metrics
rmse_svd = root_mean_squared_error(test_df['rating'], svd_predictions)
mae_svd = mean_absolute_error(test_df['rating'], svd_predictions)

print(f"\nSVD Results:")
print(f"  RMSE: {rmse_svd:.4f}")
print(f"  MAE:  {mae_svd:.4f}")
```

Calculate the improvement over the baseline:

```
improvement = (1 - rmse_svd/rmse_baseline) * 100
print(f"\nImprovement over baseline: {improvement:.1f}%")
```

Question: How much better is SVD than the baseline (predicting mean rating)? What does this tell you about patterns in humor preferences?

A6. Train a KNN model

KNN (K-Nearest Neighbors) implements user-based collaborative filtering. The idea is simple: to predict what rating you'll give a joke, we find users similar to you and use their ratings to make a prediction.

Step 1: Build the KNN model

First, build the KNN model to find similar users:

```
# Build KNN model on user vectors
# Each user is represented by their ratings across all jokes
k = 30  # How many similar users to consider
knn_model = NearestNeighbors(n_neighbors=k+1, metric='cosine')
knn_model.fit(user_item_matrix.values)

# Calculate global mean from training data for fallback
# We'll use this if we can't find similar users who rated an item
global_mean = train_df['rating'].mean()
print(f"Global mean rating: {global_mean:.2f}")
```

Cosine similarity measures how similar two users are based on their rating patterns. The model will find the k=30 most similar users for any given user.

Step 2: Make predictions

Now make predictions for each test rating by averaging similar users' ratings:

```
# Make predictions for each test rating
knn_predictions = []

for _, row in test_df.iterrows():
    user_id = int(row['user'])
    item_id = int(row['item'])

    # Find this user's k nearest neighbors
    user_vector = user_item_matrix.iloc[user_id].values.reshape(1, -1)
    distances, neighbor_indices = knn_model.kneighbors(user_vector)

    # Get ratings from neighbors for this specific joke
    neighbor_ratings = []
    for neighbor_idx in neighbor_indices[0][1:]:  # Skip user itself (first result)
        rating = user_item_matrix.iloc[neighbor_idx, item_id]
        if rating != 0:  # Only use actual ratings (0 means missing)
            neighbor_ratings.append(rating)

    # Predict as average of neighbor ratings, or global mean if no neighbors rated it
    if len(neighbor_ratings) > 0:
        knn_predictions.append(np.mean(neighbor_ratings))
    else:
        knn_predictions.append(global_mean)

knn_predictions = np.array(knn_predictions)
print(f"Made {len(knn_predictions)} predictions")
```

The prediction algorithm works like this: For each test rating, we find similar users who also rated that joke, then average their ratings. If no similar users rated it, we fall back to the global mean.

Step 3: Evaluate performance

Evaluate the KNN model:

```
# Calculate error metrics
rmse_knn = root_mean_squared_error(test_df['rating'], knn_predictions)
mae_knn = mean_absolute_error(test_df['rating'], knn_predictions)

print(f"KNN Results:")
print(f"  RMSE: {rmse_knn:.4f}")
print(f"  MAE:  {mae_knn:.4f}")
```

A7. Try content-based filtering with TF-IDF

So far we've used collaborative filtering (learning from user ratings). Now let's try content-based filtering (using the text of jokes).

First, load the joke text:

```
# Load joke text
jokes_df = pd.read_csv('jester-jokes.csv')
print(f"Loaded {len(jokes_df)} jokes")
print(f"\nSample joke:")
print(f"Joke {jokes_df.iloc[0]['joke_id']}: {jokes_df.iloc[0]['joke_text']}")
```

Now build TF-IDF vectors and compute joke similarities:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

```python
# Build TF-IDF vectors for jokes
vectorizer = TfidfVectorizer(max_features=100, stop_words='english')
tfidf_matrix = vectorizer.fit_transform(jokes_df['joke_text'])

print(f"\nTF-IDF matrix shape: {tfidf_matrix.shape}")

# Compute joke-to-joke similarity
joke_similarity = cosine_similarity(tfidf_matrix)
print(f"Joke similarity matrix shape: {joke_similarity.shape}")
```

Make predictions using content-based filtering:

```python
# Content-based predictions:
# For each test rating, find jokes the user liked and recommend similar jokes
print("Making content-based predictions...")
cb_predictions = []

for _, row in test_df.iterrows():
    user_id = row['user']
    item_id = int(row['item'])

    # Get user's training ratings
    user_train = train_df[train_df['user'] == user_id]

    if len(user_train) == 0:
        # Cold start: use global mean
        cb_predictions.append(train_df['rating'].mean())
        continue

    # Weight user's ratings by joke similarity
    similarities = joke_similarity[item_id]
    weighted_sum = 0
    weight_total = 0

    for _, train_row in user_train.iterrows():
        train_item = int(train_row['item'])
        train_rating = train_row['rating']
        similarity = similarities[train_item]

        if similarity > 0:
            weighted_sum += similarity * train_rating
            weight_total += similarity

    if weight_total > 0:
        cb_predictions.append(weighted_sum / weight_total)
    else:
        cb_predictions.append(train_df['rating'].mean())

cb_predictions = np.array(cb_predictions)

rmse_cb = root_mean_squared_error(test_df['rating'], cb_predictions)
mae_cb = mean_absolute_error(test_df['rating'], cb_predictions)

print(f"\nContent-Based (TF-IDF) Results:")
print(f"  RMSE: {rmse_cb:.4f}")
print(f"  MAE:  {mae_cb:.4f}")
```

Examine what TF-IDF considers “similar”:

```python
# Show similar jokes by TF-IDF
example_id = 0
print(f"\nJoke {example_id}: {jokes_df.iloc[example_id]['joke_text']}")
```

```
similarities = joke_similarity[example_id]
similar_indices = np.argsort(similarities)[::-1][1:4]  # Top 3 (excluding self)

print(f"\nTop 3 most similar jokes by TF-IDF:")
for rank, idx in enumerate(similar_indices, 1):
    print(f"\n{rank}. Joke {idx} (similarity: {similarities[idx]:.3f})")
    print(f"   {jokes_df.iloc[idx]['joke_text'][:100]}...")
```

Question: Are the "similar" jokes actually funny in the same way? What does TF-IDF capture about jokes, and what does it miss?

Now let's explore what words appear in funny vs unfunny jokes:

```
# Merge jokes with their mean ratings
joke_stats = ratings_df.groupby('item')['rating'].agg(['mean', 'count'])
joke_stats = joke_stats.sort_values('mean', ascending=False)
jokes_with_ratings = jokes_df.merge(
    joke_stats, left_on='joke_id', right_index=True
)

# Sort by rating to get actual funniest/worst
jokes_with_ratings = jokes_with_ratings.sort_values('mean', ascending=False)

# Show funniest and worst jokes
print("Top 5 funniest jokes:")
for i, row in jokes_with_ratings.head(5).iterrows():
    print(f"\nJoke {row['joke_id']} (rating: {row['mean']:+.2f}):")
    print(f"  {row['joke_text']}")

print("\n\nTop 5 worst jokes:")
for i, row in jokes_with_ratings.tail(5).iterrows():
    print(f"\nJoke {row['joke_id']} (rating: {row['mean']:+.2f}):")
    print(f"  {row['joke_text']}")
```

Use TF-IDF with Ridge Regression to find words associated with humor:

Step 1: Build word-level TF-IDF features

First, import Ridge Regression and build TF-IDF features for individual words:

```
# Use TF-IDF + Ridge Regression to find words associated with humor
from sklearn.linear_model import RidgeCV

# Build TF-IDF features for all jokes (individual words only)
vectorizer_words = TfidfVectorizer(
    max_features=100,      # Keep only top 100 words
    stop_words='english',  # Remove common words like "the", "a", "is"
    ngram_range=(1, 1),    # Single words only (unigrams)
    min_df=2               # Word must appear in at least 2 jokes
)

X_words = vectorizer_words.fit_transform(jokes_with_ratings['joke_text'])
y_ratings = jokes_with_ratings['mean'].values

print(f"TF-IDF matrix shape: {X_words.shape}")
print(f"Target ratings shape: {y_ratings.shape}")
```

TF-IDF (Term Frequency-Inverse Document Frequency) converts text into numbers. Each joke becomes a vector of 100 numbers representing how important each word is in that joke.

Step 2: Train Ridge Regression on words

Now train a Ridge Regression model to predict joke ratings from words:

```
# Train Ridge regression to predict ratings from words
# RidgeCV automatically selects the best regularization strength
ridge = RidgeCV(alphas=[0.01, 0.1, 1.0, 10.0, 100.0])
ridge.fit(X_words, y_ratings)

print(f"\nWord-based rating prediction R^2: {ridge.score(X_words, y_ratings):.4f}")
print(f"Best alpha: {ridge.alpha_}")
```

The R^2 score tells us how well words predict joke ratings. A score near 1.0 means words are very predictive; near 0 means they're not helpful.

Step 3: Analyze word coefficients

Examine which words are associated with funny vs unfunny jokes:

```
# Get coefficients (positive = associated with funny, negative = unfunny)
coefs = ridge.coef_
feature_names = vectorizer_words.get_feature_names_out()
sorted_indices = np.argsort(coefs)

print("\nTop 10 words associated with FUNNY jokes:")
for idx in sorted_indices[-10:][::-1]:
    print(f"  {feature_names[idx]:20s}: {coefs[idx]:+.4f}")

print("\nTop 10 words associated with UNFUNNY jokes:")
for idx in sorted_indices[:10]:
    print(f"  {feature_names[idx]:20s}: {coefs[idx]:+.4f}")
```

Positive coefficients mean the word appears more in highly-rated jokes; negative coefficients mean it appears more in poorly-rated jokes.

Step 4: Build bigram features

Now try bigrams (two-word phrases) instead of individual words:

```
# Build TF-IDF features for bigrams (two-word phrases)
vectorizer_bigrams = TfidfVectorizer(
    max_features=100,
    stop_words='english',
    ngram_range=(2, 2),  # Only two-word phrases
    min_df=2
)

X_bigrams = vectorizer_bigrams.fit_transform(jokes_with_ratings['joke_text'])
print(f"Bigram TF-IDF matrix shape: {X_bigrams.shape}")
```

Bigrams can capture phrases like "knock knock" or "chicken cross" that might be more meaningful than individual words.

Step 5: Train and evaluate bigram model

Train and evaluate the bigram model:

```
# Train Ridge regression with bigrams
ridge_bigrams = RidgeCV(alphas=[0.01, 0.1, 1.0, 10.0, 100.0])
ridge_bigrams.fit(X_bigrams, y_ratings)

print(f"\nBigram-based rating prediction R^2: {ridge_bigrams.score(X_bigrams, y_ratings):.4f}")

# Show top bigrams
coefs_bigrams = ridge_bigrams.coef_
features_bigrams = vectorizer_bigrams.get_feature_names_out()
sorted_indices_bigrams = np.argsort(coefs_bigrams)
```

```
print("\nTop 10 bigrams associated with FUNNY jokes:")
for idx in sorted_indices_bigrams[-10:][::-1]:
    print(f"  {features_bigrams[idx]:30s}: {coefs_bigrams[idx]:+.4f}")

print("\nTop 10 bigrams associated with UNFUNNY jokes:")
for idx in sorted_indices_bigrams[:10]:
    print(f"  {features_bigrams[idx]:30s}: {coefs_bigrams[idx]:+.4f}")
```

Question: Look at the funniest vs worst jokes. Can you identify patterns? Are funny jokes longer stories while unfunny ones are short Q&A puns? Can word counts alone predict humor?

A8. Compare models with a bar chart

Visualize the performance of all four models.

```
models = ['Baseline', 'Content-Based', 'KNN', 'SVD']
rmse_values = [rmse_baseline, rmse_cb, rmse_knn, rmse_svd]

plt.figure(figsize=(10, 5))
plt.bar(models, rmse_values, color=['gray', 'orange', 'lightblue', 'lightgreen'],
        edgecolor='black', alpha=0.7)
plt.ylabel('RMSE (Lower is Better)')
plt.title('Model Comparison on Jester Dataset')
plt.grid(axis='y', alpha=0.3)
for i, v in enumerate(rmse_values):
    plt.text(i, v + 0.05, f'{v:.4f}', ha='center', va='bottom')
plt.show()
```

Question: Which model performs best? Which model is worst?

A9. Analyze joke ratings

The SVD model is already trained on the full dataset. Let's analyze joke ratings.

```
# Calculate mean ratings per joke
joke_stats = ratings_df.groupby('item')['rating'].agg(['mean', 'count'])
joke_stats = joke_stats.sort_values('mean', ascending=False)

print(f"\nTop 5 funniest jokes (highest average rating):")
for idx, (joke_id, row) in enumerate(joke_stats.head(5).iterrows(), 1):
    print(f"  {idx}. Joke {joke_id}: avg rating = {row['mean']:+.2f} ({int(row['count'])} ratings)")

print(f"\nTop 5 worst jokes (lowest average rating):")
for idx, (joke_id, row) in enumerate(joke_stats.tail(5).iterrows(), 1):
    print(f"  {idx}. Joke {joke_id}: avg rating = {row['mean']:+.2f} ({int(row['count'])} ratings)")
```

A10. Generate personalized joke recommendations

Use the SVD model to recommend jokes for a specific user.

```
# Get recommendations for user 0
user_id = 0

# Get user's predictions from the reconstructed ratings matrix
user_predictions = predicted_ratings[user_id, :]

# Find items the user hasn't rated yet
user_rated_items = ratings_df[ratings_df['user'] == user_id]['item'].values
unrated_items = [i for i in range(n_jokes) if i not in user_rated_items]

# Get predictions for unrated items and sort
```

```
unrated_predictions = [(item, user_predictions[item]) for item in unrated_items]
unrated_predictions.sort(key=lambda x: x[1], reverse=True)

# Show top 10
print(f"\nTop 10 joke recommendations for User {user_id}:")
for i, (joke_id, pred_rating) in enumerate(unrated_predictions[:10], 1):
    print(f"  {i}. Joke {joke_id}: predicted rating = {pred_rating:+.2f}")
```

Question: If you were building a real joke app, how would you present these recommendations to users? Would you just show the top-rated jokes, or try to balance quality with diversity?

Part B — Implicit Feedback Comparison (40 min)

In Part A, we used **explicit feedback** (users rated jokes). Now we'll explore **implicit feedback**, where we only observe user behavior (clicks, views) without explicit ratings.

B1. Import the implicit library

Import the `implicit` library (you already installed it in Preparation):

```
import implicit.als
import implicit.evaluation
from scipy.sparse import csr_matrix
```

B2. Load MovieLens for implicit feedback

We'll use MovieLens 100K, but convert it to implicit feedback.

```
# Load MovieLens 100K dataset
print("Loading MovieLens 100K dataset...")
url = "https://files.grouplens.org/datasets/movielens/ml-100k/u.data"
ml_data = pd.read_csv(url, sep='\t', names=['user', 'item', 'rating', 'timestamp'])

# Convert to 0-indexed
ml_data['user'] = ml_data['user'] - 1
ml_data['item'] = ml_data['item'] - 1

n_users_ml = ml_data['user'].nunique()
n_items_ml = ml_data['item'].nunique()

print(f"MovieLens 100K:")
print(f"  Users: {n_users_ml}")
print(f"  Movies: {n_items_ml}")
print(f"  Ratings: {len(ml_data):,}")
```

B3. Convert to implicit interactions

Convert ratings to binary interactions: any rating ≥ 4 means "watched and liked."

```
# Convert to implicit interactions (rating >= 4 means positive interaction)
implicit_data = ml_data[ml_data['rating'] >= 4.0].copy()

print(f"Positive interactions (rating >= 4): {len(implicit_data):,}")
print(f"This is {len(implicit_data)/len(ml_data)*100:.1f}% of all ratings")
```

B4. Create sparse user-item matrix

Build the sparse matrix that the `implicit` library expects.

```
# Create sparse user-item matrix
user_item_matrix = csr_matrix(
    (np.ones(len(implicit_data)),
     (implicit_data['user'].values, implicit_data['item'].values)),
    shape=(n_users_ml, n_items_ml)
)

print(f"Matrix shape: {user_item_matrix.shape}")
print(f"Sparsity: {100 * (1 - user_item_matrix.nnz / (n_users_ml * n_items_ml)):.2f}%")
```

B5. Train ALS model

Train an Alternating Least Squares model on the implicit data.

```
model_als = implicit.als.AlternatingLeastSquares(
    factors=50,
    regularization=0.01,
    iterations=20,
    random_state=42
)

# Note: implicit library expects item-user matrix (transposed!)
model_als.fit(user_item_matrix.T)
print("ALS training complete!")
```

B6. Evaluate with ranking metrics

For implicit data, we can't use RMSE (there are no rating values!). Instead, we use ranking metrics like Precision@K, which measures: "Of the top K recommendations, how many were actually relevant?"

Step 1: Split data into train and test

First, split the data into train and test sets:

```
# Split implicit data into train/test for evaluation
# For each user, hold out 20% of their interactions for testing
from sklearn.model_selection import train_test_split

train_data, test_data = train_test_split(implicit_data, test_size=0.2, random_state=42)

print(f"Train interactions: {len(train_data):,}")
print(f"Test interactions: {len(test_data):,}")
```

Step 2: Create sparse matrices

Now create separate sparse matrices for training and testing:

```
# Create train matrix (used for model training)
train_user_item = csr_matrix(
    (np.ones(len(train_data)),
     (train_data['user'].values, train_data['item'].values)),
    shape=(n_users_ml, n_items_ml)
)

# Create test matrix (held out for evaluation)
test_user_item = csr_matrix(
    (np.ones(len(test_data)),
     (test_data['user'].values, test_data['item'].values)),
    shape=(n_users_ml, n_items_ml)
)
```

```
print(f"Train matrix shape: {train_user_item.shape}")
print(f"Test matrix shape: {test_user_item.shape}")
```

The sparse matrix format stores only the non-zero entries (user-item interactions), which is memory-efficient for large datasets.

Step 3: Re-train ALS model

Re-train the ALS model on training data only:

```
# Re-train model on training data only
model_als = implicit.als.AlternatingLeastSquares(
    factors=50,           # Latent factors (like SVD)
    regularization=0.01,  # Prevent overfitting
    iterations=20,        # How many training iterations
    random_state=42
)

# Note: implicit library expects item-user matrix (transposed!)
model_als.fit(train_user_item.T)
print("ALS training complete!")
```

Step 4: Evaluate with Precision@K

Evaluate using Precision@10 metric:

```
# Evaluate on test set using Precision@K
# Precision@10 = "Of the top 10 recommendations, how many were actually relevant?"
# Note: precision_at_k may have compatibility issues with some scipy versions
try:
    precision = implicit.evaluation.precision_at_k(
        model_als, train_user_item, test_user_item, K=10,
        show_progress=False, num_threads=1
    )
    print(f"\nPrecision@10: {precision:.4f}")
    print(f"Interpretation: {precision*100:.1f}% of top-10 recommendations are relevant")
except Exception as e:
    print(f"\nNote: Precision evaluation skipped due to library compatibility issue")
    print(f"Error: {type(e).__name__}")
```

B7. Get recommendations from implicit model

Generate recommendations for a user.

```
user_idx = 0  # First user

# Get recommendations (use training data to filter out items already seen)
item_ids, scores = model_als.recommend(user_idx, train_user_item[user_idx], N=10)

print(f"\nTop 10 recommendations for User {user_idx}:")
for i, (item_idx, score) in enumerate(zip(item_ids, scores), 1):
    print(f"  {i}. Movie {item_idx}: confidence score = {score:.3f}")
```

Question: How do these “confidence scores” differ from the predicted ratings in Part A?

B8. Compare explicit vs implicit approaches

Create a comparison table in your notebook.

```
comparison = pd.DataFrame({
    'Aspect': ['Data Type', 'Signal Strength', 'Data Volume',
               'Goal', 'Evaluation', 'Example'],
    'Explicit (Jester)': ['Star ratings (-10 to +10)', 'Strong', 'Moderate',
                          'Predict rating value', 'RMSE, MAE', 'Netflix ratings'],
    'Implicit (MovieLens)': ['Binary (watched/not)', 'Weak', 'Abundant',
                             'Rank by preference', 'Precision@K', 'YouTube views']
})
comparison
```

Easy Extensions (Optional)

If you have time, try this personalized recommendation exercise!

Rate some jokes and get recommendations

Let's make this personal! Rate a few jokes from the database and get recommendations for jokes you might like.

Step 1: Rate some jokes

```
# Display a few random jokes for you to rate
import random

print("Let's find jokes you'll like! Rate these jokes on a scale from -10 to +10:")
print("(-10 = terrible, 0 = neutral, +10 = hilarious)\n")

# Select 5 random jokes from the dataset
random_joke_ids = random.sample(range(n_jokes), 5)
your_ratings = {}

for i, joke_id in enumerate(random_joke_ids, 1):
    joke_text = jokes[joke_id] if joke_id < len(jokes) else f"Joke {joke_id}"
    print(f"--- Joke {i}/5 (ID: {joke_id}) ---")
    print(joke_text)
    print()

    # Get rating from user
    while True:
        try:
            rating = float(input(f"Your rating (-10 to +10): "))
            if -10 <= rating <= 10:
                your_ratings[joke_id] = rating
                break
            else:
                print("Please enter a number between -10 and +10")
        except ValueError:
            print("Please enter a valid number")
    print()

print("\nThanks for rating! Now let's find jokes you'll love...\n")
```

Step 2: Generate personalized recommendations

Now use the SVD model to predict ratings for all unrated jokes and recommend the best ones:

```
# Create a new user ID (next available user number)
your_user_id = n_users
```

```python
# Get the SVD model's predictions for your ratings
# We'll use the average user's latent factors as a starting point
avg_user_factors = svd_model.pu.mean(axis=0)

# Adjust based on your ratings vs. average ratings
your_factors = avg_user_factors.copy()
for joke_id, rating in your_ratings.items():
    # Get the joke's latent factors
    joke_factors = svd_model.qi[joke_id]
    # Update your factors based on the difference from expected rating
    avg_joke_rating = ratings_df[ratings_df['item'] == joke_id]['rating'].mean()
    rating_diff = rating - avg_joke_rating
    your_factors += 0.1 * rating_diff * joke_factors

# Predict ratings for all jokes
your_predictions = np.dot(your_factors, svd_model.qi.T)

# Find unrated jokes and sort by predicted rating
unrated_jokes = [j for j in range(n_jokes) if j not in your_ratings]
joke_predictions = [(j, your_predictions[j]) for j in unrated_jokes]
joke_predictions.sort(key=lambda x: x[1], reverse=True)

# Show top 5 recommendations
print("Based on your ratings, here are 5 jokes we think you'll love:\n")
for i, (joke_id, pred_rating) in enumerate(joke_predictions[:5], 1):
    joke_text = jokes[joke_id] if joke_id < len(jokes) else f"Joke {joke_id}"
    print(f"--- Recommendation {i} (Joke {joke_id}, predicted rating: {pred_rating:+.2f}) ---")
    print(joke_text)
    print()
```

Question: Do the recommended jokes match your sense of humor? If you rated them, how close were the predictions to your actual ratings?

Discussion Questions (Optional)

1. **Cold Start Problem**: How would you handle a brand new user who has never rated any jokes? What about a new joke that no one has rated yet?
2. **Explicit vs Implicit**: YouTube uses implicit feedback (watch time, clicks) instead of asking users to rate videos. What are the advantages and disadvantages of this approach?
3. **Ethics**: The lecture discussed how YouTube optimizes for engagement (watch time) while Netflix optimizes for satisfaction. How might this difference affect what content gets recommended? Can you think of potential harms from optimizing purely for engagement?
4. **Filter Bubbles**: If a recommender system only shows you content similar to what you've liked before, what problems might this cause? How would you modify a recommender system to promote diversity while still being useful?
5. **Evaluation Metrics**: We used RMSE for explicit ratings and Precision@K for implicit feedback. Why are these different? What does each metric actually measure?
6. **Real-World Application**: If you were building a recommender system for a streaming music service (like Spotify), would you use explicit ratings, implicit feedback, or both? Justify your answer.

Extension Challenges (Optional)

If you finish early, try these challenges:

1. **Hybrid System**: Combine both SVD and KNN predictions by averaging their outputs. Does this perform better than either alone?
2. **Temporal Patterns**: The Jester dataset may have timestamps. Can you detect whether joke preferences change over time?
3. **User Similarity**: Find the most similar user to User 1 using Pearson correlation on their joke ratings. Do they have similar taste in jokes?
4. **Item-Based CF**: We used user-based collaborative filtering (find similar users). Try item-based filtering (find similar jokes) by setting `user_based=False` in KNN.

Quiz questions

As usual, carry on with some quiz questions.