Programming Item-Item Collaborative Filtering

In this assignment, you will create a simple implementation of item-item collaborative filtering. Note that LensKit already has an implementation of item-item that is different from what we're asking you to build; do not try to copy that implementation as it will not produce the correct results for this assignment.

The deliverable for this assignment is your code, which we will test in the online grading infrastructure.

Start by downloading the project template. This is a Gradle project; you can import it into your IDE directly (IntelliJ users can open the build.gradle file as a project). This contains files for all the code you need to implement, along with the Gradle files needed to build, run, and evaluate.

Downloads and Resources

- Project template (on Coursera))
- LensKit for Teaching website
- JavaDoc for included code

Additionally, you will need:

- Java — download the Java 8 JDK. On Linux, install the OpenJDK 'devel' package (you will need the devel package to have the compiler).
- A development environment.

Implementing Item-Item Collaborative Filtering

Your task is to write the missing pieces of the following classes:

`SimpleItemItemModelBuilder` Builds the item-item model from the rating data
`SimpleItemItemScorer` Scores items with item-item collaborative filtering
`SimpleItemBasedItemScorer` Finds similar items

The primary component of this assignment is your implementation of item-item CF. The provided `SimpleItemItemModel` class stores the precomputed similarity matrix.

Computing Similarities

The `SimpleItemItemModelBuilder` class computes the similarities between items and stores them in the model. It also needs to create a vector mapping each item ID to its mean rating, for use by the item scorer. Use the following configuration decisions:

- Normalize each item rating vector by subtracting the **item's** mean rating from each rating prior to computing similarities
- Use cosine similarity between normalized item rating vectors
- Only store neighbors with positive similarities (> 0)

One way to approach this is to process the ratings item-by-item (using `ItemEventDAO.streamEventsByIte` convert each item's ratings to a rating vector (`Ratings.itemRatingVector`), and normalize and store each item's rating vector. The stub code we have provided starts you in this direction, but it is not the only way to implement it.

The similarity matrix should be in the form of a `Map` from `Longs` (items) to `Long2DoubleMaps` (their neighborhoods). Each `Long2DoubleMap` stores a neighborhood, where each neighbor's id (the key) is associated with a similarity score (the value).

Scoring Items

The `SimpleItemItemScorer` class uses the model of neighborhoods to actually compute scores. Score the items using the weighted average of the users' ratings for similar items.

Use at most 20 neighbors to score each item; if the user has rated more neighboring items than that, use only the most similar ones.

Normalize the user's ratings by subtracting the **item's** mean rating from each rating prior to averaging (this is necessary to get good results with the item-mean normalization above). You can get the item mean ratings from the model class. The resulting score function is as follows, where $w_{ij} = \text{sim}(i, j)$, the similarity between the two items:

$$s(i; u) = \mu_i + \frac{\sum_{j \in I_u} (r_{uj} - \mu_j) w_{ij}}{\sum_{j \in I_u} |w_{ij}|}$$

Basket Recommendation

The item-item similarity matrix isn't just useful for generating personalized recommendations. It is also useful for 'find similar items' features.

The LensKit `ItemBasedItemScorer` and `ItemBasedItemRecommender` interfaces provide this functionality. `ItemBasedItemScorer` is like `ItemScorer`, except that it scores items with respect to a set of items rather than a user.

The item-based item scorer receives a `basket` (the set of reference items) and `items` (the set of items to score) vector, similar to `ItemScorer`. For our implementation, you will score each item with the *sum* of its similarity to each of the reference items in the basket. Note that you aren't using the `neighborhoodSize` parameter here—you're using all of the reference items in the basket.

Fill in the missing pieces of `SimpleItemBasedItemScorer`.

Example Output

Use Gradle to build and run your program and the evaluations. Make sure to check your program's output against the sample output given below to make sure your implementation is correct. Once you've done that, you can move on to running your evaluations.

Predictions

Command:

```
./gradlew predict -PuserId=320 -PitemIds=153,260,527,588
```

Output:

```
predictions for user 320:
  153 (Batman Forever (1995)): 2.476
  260 (Star Wars: Episode IV - A New Hope (1977)): 4.262
  527 (Schindler's List (1993)): 4.167
  588 (Aladdin (1992)): 3.565
```

Recommendations

Command:

```
./gradlew recommend -PuserId=320
```

Output:

```
recommendations for user 320:
  7502 (Band of Brothers (2001)): 4.484
  1224 (Henry V (1989)): 4.423
  858 (Godfather, The (1972)): 4.408
  318 (Shawshank Redemption, The (1994)): 4.403
  1203 (12 Angry Men (1957)): 4.386
  3462 (Modern Times (1936)): 4.379
  99114 (Django Unchained (2012)): 4.376
  4973 (Amelie (Fabuleux destin d'Am?lie Poulain, Le) (2001)): 4.376
  898 (Philadelphia Story, The (1940)): 4.371
  922 (Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)): 4.357
```

Similar Items

Command:

```
./gradlew itemBasedRecommend -PitemIds=153,260,527,588
```

Output:

```
1196 (Star Wars: Episode V - The Empire Strikes Back (1980)): 1.103
1210 (Star Wars: Episode VI - Return of the Jedi (1983)): 1.099
364 (Lion King, The (1994)): 1.012
595 (Beauty and the Beast (1991)): 1.005
1 (Toy Story (1995)): 0.925
500 (Mrs. Doubtfire (1993)): 0.893
5349 (Spider-Man (2002)): 0.891
480 (Jurassic Park (1993)): 0.888
1291 (Indiana Jones and the Last Crusade (1989)): 0.885
150 (Apollo 13 (1995)): 0.871
```

Submitting

Use the `prepareSubmission` Gradle task to create a `jar` file and upload it to the Coursera assignment tool, as with the previous assignments.

Grading

Your grading will be based on output with randomly-selected inputs; 75% scores having the correct order, and 25% scores being correct.

The parts are weighted as follows:

- 70% personalized item-item
- 30% item-based scores