# Programming Assignment 5

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.

There are two deliverables in this assignment:

- Your code for your assignment, which we'll evaluate by running test cases on it
- A short submission on evaluation results from running your code

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 course website)
- LensKit for Teaching website
- LensKit evaluator documentation
- 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).
- An IDE; we recommend IntelliJ IDEA Community Edition.

## 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.streamEventsByItem`) 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 `Long`s (items) to `Long2DoubleMap`s (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 `neighborhoodSize` neighbors to score each item; if the user has rated more neighboring items than that, use only the most similar ones. This parameter is set in the constructor, where it comes in via the `@NeighborhoodSize` parameter; later, you will tune this parameter using cross-validation.

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 this:

$$p_{ui} = \mu_i + \frac{\sum_{j \in I_u} (r_{uj} - \mu_j) sim(i, j)}{\sum_{j \in I_u} |sim(i, j)|}$$

### 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=42 -PitemIds=4226,592,2761,33004
```

Output:

```
predictions for user 42:
  592 (Batman (1989)): 2.670
  2761 (Iron Giant, The (1999)): 3.578
  4226 (Memento (2000)): 3.394
  33004 (Hitchhiker's Guide to the Galaxy, The (2005)): 2.919
```

### Recommendations

Command:

```
./gradlew recommend -PuserId=42
```

Output:

```
recommendations for user 42:
  71450 (Capitalism: A Love Story (2009)): 5.449
  3055 (Felicia's Journey (1999)): 5.258
  4350 (Forgotten Silver (1996)): 5.188
  6650 (Kind Hearts and Coronets (1949)): 5.163
  31437 (Nobody Knows (Dare mo shiranai) (2004)): 5.125
  6984 (Tale of Two Cities, A (1935)): 4.966
  3858 (Cecil B. DeMented (2000)): 4.940
  7925 (Hidden Fortress, The (Kakushi-toride no san-akunin) (1958)): 4.916
  6649 (Tunes of Glory (1960)): 4.913
  3622 (Twelve Chairs, The (1970)): 4.893
```

**Similar Items**

Command:

```
./gradlew itemBasedRecommend -PitemIds=77
```

Output:

```
2565 (King and I, The (1956)): 0.451
6380 (Capturing the Friedmans (2003)): 0.441
3397 (Great Muppet Caper, The (1981)): 0.434
83 (Once Upon a Time... When We Were Colored (1995)): 0.427
2176 (Rope (1948)): 0.425
3153 (7th Voyage of Sinbad, The (1958)): 0.420
73 (Misérables, Les (1995)): 0.405
3398 (Muppets Take Manhattan, The (1984)): 0.388
2083 (Muppet Christmas Carol, The (1992)): 0.360
30822 (In Good Company (2004)): 0.360
```

## Running the Evaluator

Copy your `TagEntropyMetric` file over from the last programming assignment, and put in in the `src/main/java/edu/umn/cs/recsys/` folder, just like before. Make sure you run the tests included in the last programming assignment to make sure your `TagEntropyMetric` code is correct, otherwise you may get incorrect evaluation results.

Now that you have your recommender working, let's evaluate it. The `build.gradle` file runs the evaluator on the algorithms defined in `algorithms.groovy`, just like before. It will run your item-item recommender with a range of neighborhood sizes. It will also compares it against the user-user CF implementation from LensKit.

Run the evaluator as follows:

```
./gradlew evaluate
```

This run will take a while! Consider letting it run overnight, or trying to run it on one of the faster CS department servers.

In the output (`build/eval-results.csv`), you will see the metrics over the two algorithms and various neighborhood sizes. Plot and examine the results; consider the mean of each metric over all partitions of a particular data set (so you'll have one number for each combination of algorithm, neighborhood size, and data set).

Submit a PDF file answering the following questions, with supporting plots or graphs:

1. What algorithm and neighborhood size produce the lowest RMSE?
2. Which algorithm has the highest tag entropy? User-user or item-item?

## Submitting

Use the `prepareSubmission` Gradle task to create a zip file and submit it along with a PDF containing your answers to TRACS.