CBF Programming Assignment

Overview

In this assignment, you will implement a content-based recommender as a LensKit recommender algorithm. LensKit provides tools to produce recommendations from a user; your task is to implement the logic of the recommender itself.

There are 2 parts to this assignment, implementing two variants of a TF-IDF recommender.

Like you did in the nonpersonalized recommenders, the CBF recommenders are broken into three components:

- A model class, `TFIDFModel`, that you do not need to modify.
- A model provider, `TFIDFModelProvider`, that you will need to complete with the logic to compute TF-IDF vectors for items.
- A scorer/recommender class that uses the precomputed model to score items. In this assignment, you will implement an `ItemScorer`; its only job is to compute user-personalized scores for items. You do not have to rank the items: LensKit will take care of that for you.

In addition, the item scorer will use another component, the `UserProfileBuilder`, to compute user tag interest profiles based on a user's ratings and the tags applied to the movies they like. You will implement two versions of this profile builder.

Downloads and Resources

- Project template (from Coursera)
- LensKit for Teaching website (links to relevant 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.

Notation

Here's the mathematical notation we use in this assignment description:

$\mathbf{p}_u$ The user's profile vector, indicating how much the user likes each tag.
p_{ut} User u's preference for a particular tag t (this is an element of $\mathbf{p}_u$).[1]

[1] If you have reviewed the Standardized Notation resource, you may notice that we generally use $\mathbf{p}$ and $\mathbf{q}$ vectors for matrix factorization, a topic in Course 4. However, in that setting, they serve as user and item

$\mathbf{q}_i$ The tag vector for an item i. In the final computation, this will be a unit-normalized TF-IDF vector.

q_{it} The value for a particular tag t for an item i (an element of $\mathbf{q}_i$).

I_u The set of items rated by user u.

r_{ui} User u's rating for item i.

$\bar{r_u}$ The average of user u's ratings.

$||\mathbf{v}||_2$ The Euclidean (L_2) norm, or length, of a vector $\mathbf{v}$. It is the square root of the sum of the squares of the elements of $\mathbf{v}$.

Part 1: TF-IDF Recommender with Unweighted Profiles

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). The code should compile as-is; you can test this by running the `build` Gradle target from your IDE, or running `./gradlew build` at the command line.

There are 3 things you need to implement to complete the first part of the assignment:

Compute item-tag vectors (the model) For this task, you need to modify the model builder (`TFIDFModelProvider`, your modifications go in the get() method) to compute the unit-normalized TF-IDF vector for each movie in the data set. We provide the skeleton of this; TODO comments indicate where you need to implement missing pieces. When this piece is done, the model should contain a mapping of item IDs to TF-IDF vectors, normalized to unit vectors, for each item.

There are two main steps to this task:

1. Iterate through items, building the term vector $\hat{\mathbf{q}}_i$ for each item and a global document frequency vector $\mathbf{d}$. At this stage, these are unnormalized term and document frequency vectors, storing the number of times the term appears on each document or the number of documents in which it appears.
2. Iterate through each item again, performing the following:
    1. Divide each term value $\hat{q}_{it}$ by the log of the document frequency ($\ln d_t$). The resulting vector $\tilde{\mathbf{q}}_i$ is the TF-IDF vector.
    2. After dividing each term value by the log of the DF, compute the length (Euclidean norm) of the TF-IDF vector $\tilde{\mathbf{q}}_i$, and divide each element of it by the length to yield a unit vector $\mathbf{q}_i$.

The `getItemTags` method of the `ItemTagDAO` class, an instance of which is available as a field in your model builder, will give you the list of all tags applied to the item.

Build user profile for each query user The `UserProfileBuilder` interface defines classes that take a user's history – a list of ratings — and produce a vector $\mathbf{p}_u$ representing

description vectors, respectively; we therefore use them here in that sense, describing users and items by their scores for tags.

that user's profile. Recall that the user profile vectors will also be term vectors, describing how much the user likes each tag. For part 1, the profile should be the sum of the item-tag vectors of all items the user has rated positively (>= 3.5 stars); this implementation goes in `ThresholdUserProfileBuilder`.

Mathematically, this is:

$$p_{ut} = \sum_{i \in I_u : r_{ui} \geq 3.5} q_{it}$$

Generate item scores for each user The heart of the recommendation process in many LensKit recommenders is the score method of the item scorer, in this case `TFIDFItemScorer`. Modify this method to score each item by using cosine similarity: the score for an item is the cosine between that item's tag vector and the user's profile vector. Cosine similarity is defined as follows:

$$\cos(\mathbf{p_u}, \mathbf{q_i}) = \frac{\mathbf{p_u} \cdot \mathbf{q_i}}{\|\mathbf{p_u}\|_2 \|\mathbf{q_i}\|_2} = \frac{\sum_t q_{ut} p_{it}}{\sqrt{\sum_t q_{ut}^2} \sqrt{\sum_t p_{it}^2}}$$

When you are done, you can run your program from the command line using Gradle:

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

Try different user IDs.

We have also provide some unit tests to check basic functionality of your classes, particularly the TF-IDF model builder. Run these tests in your IDE to pre-check your code. They are not extensive, and many incorrect solutions will pass them, but they will help you flush out general data handling bugs and make sure the model builder at least runs before you move on to the scorer and user profile builders.

Example Item Term Vector

Item 2231 has the following final term vector:

```
{cards=0.1076736727697595,
 John Malkovich=0.23970853283741034,
 Matt Damon=0.3159509700099918
 poker=0.5969723646885582
 gambling=0.24572008949856772
 library vhs=0.06053343286414036
 card games=0.12526237038499663
 2.5=0.06789050985047351
```

```
 John Turturro=0.2606167586076272
 Edward Norton=0.5529141975174856
 John Dahl=0.1141651377329722
 watched 2006=0.061105276598972115}
```

Example Output for Unweighted User Profile

The following example gives actual outputs for user 320 in the data set. It was executed using `./gradlew recommendBasic -PuserId=320` in a Unix-like console.

```
recommendations for user 320:
  32 (Twelve Monkeys (a.k.a. 12 Monkeys) (1995)): 0.349
  1748 (Dark City (1998)): 0.307
  1206 (Clockwork Orange, A (1971)): 0.298
  48394 (Pan's Labyrinth (Laberinto del fauno, El) (2006)): 0.288
  1199 (Brazil (1985)): 0.288
  32587 (Sin City (2005)): 0.287
  1270 (Back to the Future (1985)): 0.283
  1089 (Reservoir Dogs (1992)): 0.278
  1210 (Star Wars: Episode VI - Return of the Jedi (1983)): 0.277
  7147 (Big Fish (2003)): 0.264
```

Part 2: Weighted User Profile

For this part, adapt your solution from Part 1 to compute weighted user profiles. Put your weighted user profile code in `WeightedUserProfileBuilder`.

In this variant, rather than just summing the vectors for all positively-rated items, compute a weighted sum of the item vectors for all rated items, with weights being based on the user's rating. Your solution should implement the following formula:

$$\mathbf{p}_u = \sum_{i \in I(u)} (r_{ui} - \mu_u)\mathbf{q}_i$$

Using non-vector notation, each user tag value x_{ut} is computed as follows:

$$p_{ut} = \sum_{i \in I(u)} (r_{ui} - \mu_u)q_{it}$$

If an item i has no data for a tag t, then $q_{it} = 0$.

Example Output for Weighted User Profile

The following example gives actual outputs for user 320 in the data set. It was executed using `./gradlew recommendWeighted -PuserId=320` in a Unix-like console.

```
recommendations for user 320:
  48394 (Pan's Labyrinth (Laberinto del fauno, El) (2006)): 0.204
  7153 (Lord of the Rings: The Return of the King, The (2003)): 0.184
  1210 (Star Wars: Episode VI - Return of the Jedi (1983)): 0.178
  628 (Primal Fear (1996)): 0.177
  2997 (Being John Malkovich (1999)): 0.176
  1089 (Reservoir Dogs (1992)): 0.165
  7147 (Big Fish (2003)): 0.161
  32587 (Sin City (2005)): 0.160
  48780 (Prestige, The (2006)): 0.160
  1199 (Brazil (1985)): 0.155
```

Running the Program

We have provided two Gradle targets to help you run the program. `recommendBasic` computes recommendations using the threshold user profile builder, and `recommendWeighted` uses the weighted user profile builder. You can users to recommend for with the `userId` property, e.g. `-PuserId=91`.

For example:

```
$ ./gradlew recommendBasic -PuserId=91
```

Debugging

If you run the Gradle tasks using IntelliJ's Gradle runner, you can run them under the debugger to debug your code.

The Gradle file also configures LensKit to write log output to `build/recommend-basic.log` and `build/recommend-weighted.log`. If you use the SLF4J logger (the `logger` field on the classes we provide) to emit debug messages, you can find them there.

Submitting

You will submit a compiled `jar` file containing your solution. To prepare your project for submission, run the Gradle `prepareSubmission` task:

```
./gradlew prepareSubmission
```

This will create file `cbf-submission.jar` under `build/distributions` that contains your final solution code in a format the grader will understand. Upload this `jar` file to the Coursera assignment grader.

Grading

Your grade for each part will be based on two components:

- Outputting items in the correct order: 75%
- Computing correct scores for items (within an error tolerance): 25%

The parts themselves are weighted as follows:

- Basic CBF: 70%
- CBF with Weighted User Profiles: 30%