Programming Assignment 6

As discussed in class...

- This assignment is optional, but highly encouraged (as it will help you understand matrix factorization recommenders);
- If you wish to have it graded, you must submit it by Wednesday, December 16, by 11:59PM;
- Completing this assignment can only help your grade, not hurt it. If you perform better on this assignment than any prior one, we'll replace the prior assignment grade with this one.

In this assignment, you will create a simple matrix factorization recommender. This will be a collaborative filter, computing the SVD over the rating matrix. You will use a third-party linear algebra package (Apache commons-math) to compute the SVD.

There are two deliverables in this assignment:

- Output from your SVD collaborative filter
- A short quiz on evaluation results

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.

Resources

- Project template (on course website)
- LensKit for Teaching website
- LensKit evaluator documentation
- JavaDoc for included code
- commons-math API documentation

Implementing SVD CF

The primary component of this assignment is your implementation of SVD-based collaborative filtering. The class `SVDModel` stores decomposed rating matrix. Your task is to write the missing pieces of the following classes:

`SVDModelBuilder` Builds the item-item model from the rating data
`SVDItemScorer` Scores items with item-item collaborative filtering

Your SVD CF implementation will compute the decomposition of the *normalized* (mean-centered) data. This means that missing data is represented in the matrix as a 0, which means no deviation from the mean (user, item, or personalized user-item) that is subtracted. The particular mean to subtract will be configurable, and you will experiment with different options.

SVD Layout

As we discussed in the lectures, the singular value decomposition factors the ratings matrix R so that $R \approx U\Sigma V^{\mathrm{T}}$. We work with V^{T} (the transpose of V) so that both the user-feature matrix (U) and item-feature matrix (V) have users (or items) along their *rows* and latent features along their *columns*.

Apache commons-math refers to the singular value matrix Σ as `S`.

Computing the SVD

The biggest piece of building the SVD recommender is setting up and computing the singular value decomposition. For this assignment, you'll be using an external library to do the decomposition itself, but need to set up the rating matrix and process the SVD results.

Apache Commons Math has a few key classes that you will need to use:

- RealMatrix is the core API for commons-math matrices.
- MatrixUtils provides the factory methods for building matrices.
- SingularValueDecomposition computes and stores a singular value decomposition of a matrix.

The job of `SVDModelBuilder` is to build the SVD model by doing the following:

1. Normalize the ratings data. We want to normalize the rating values so that items have a negative or positive rating based on how the user's rating compares to a mean. Because this is a sparse matrix, we'll leave unrated items with their default normalized rating of 0. As we will see below, we parameterize the baseline scorer so that we can test different mean values for normalizing our ratings.
2. Populate a `RealMatrix` with the rating data. This matrix will have users along the rows and items along the columns, so a user's rating goes at `setEntry(u, i, v)`, where `u` is the user's index, `i` the item's index, and `v` the rating. We've set up `Long2LongMap`s for you to use to get user and item indexes; these provide mappings between IDs and 0-based indexes suitable for use as matrix row/column numbers.
3. Compute the SVD of the matrix.

4. Construct an `SVDModel` containing the results (and user/item index mappings, as they are necessary to interpret the matrices).

The `SVDModelBuilder` class contains TODO comments where you need to write code. Besides the DAOs, it takes two important parameters:

- A feature count (`@LatentFeatureCount`)
- A baseline scorer to provide mean ratings

We want to experiment with different means for computing the SVD. Therefore, your `SVDModelBuilder` will be configurable. It takes a baseline item scorer as a constructor parameter; this scorer will provide the mean that you are to subtract.

When populating the matrix, process the ratings user-by-user and subtract each user's baseline scores from their ratings. If a user has not rated an item, leave that entry blank (0).

Computing the SVD itself is just a matter of instantiating a SingularValueDecomposition class from the rating matrix; we have provided this code for you.

The commons-math SVD class computes a complete SVD. For recommenders, we usually want to truncate the SVD to only include the top N latent features. Truncate the matrices from the SVD before you create the `SVDModel`. The SVD class has `getU()`, `getV()`, and `getS()` methods to get the left, right, and singular value matrices, respectively. You will need to truncate each of these matrices. The `getSubMatrix` method of RealMatrix is good for this. Truncate them to the specified latent feature count.

Scoring Items

Fill out the `SVDItemScorer` class to use the `SVDModel` to score items for a user. It will need to consult the baseline scorer to get the initial baseline scores (mean ratings), and then add the scores computed from the SVD matrices to these baseline scores.

The model exposes `getItemVector` and `getUserVector` methods to access item and user data. These methods return `RealVector` *row vectors* — matrices with a single row. The `RealMatrix` class provides transposition, multiplication, and many other matrix operations.

There are no configurable parameters to the item scorer, it just uses the model and user event DAO to compute user-personalized scores.

Example Output

Command:

```
./gradlew predict -PuserId=1024 -PitemIds=77,268,462,393,36955 -Pbaseline=pers-mean
```

Output:

```
predictions for user 1024:
  77 (Memento (2000)): 4.269
  268 (Batman (1989)): 2.870
  393 (Kill Bill: Vol. 2 (2004)): 3.379
  462 (Erin Brockovich (2000)): 3.560
  36955 (True Lies (1994)): 2.776
```

The `predict` (and `recommend`) tasks in this assignment take an additional property, `baseline`. There are several values for this property:

- `global-mean`
- `item-mean`
- `user-mean`
- `pers-mean` (personalized mean, the default)

Another example:

```
./gradlew recommend -PuserId=1024 -Pbaseline=item-mean
```

Output:

```
recommendations for user 1024:
  238 (The Godfather (1972)): 4.414
  629 (The Usual Suspects (1995)): 4.396
  424 (Schindler's List (1993)): 4.374
  14 (American Beauty (1999)): 4.327
  550 (Fight Club (1999)): 4.270
  641 (Requiem for a Dream (2000)): 4.234
  122 (The Lord of the Rings: The Return of the King (2003)): 4.107
  120 (The Lord of the Rings: The Fellowship of the Ring (2001)): 4.081
  121 (The Lord of the Rings: The Two Towers (2002)): 4.031
  453 (A Beautiful Mind (2001)): 4.024
```

Running the Evaluator

Now that you have your recommender working, let's evaluate it. The `eval.groovy` script runs the evaluation, as before. It runs your SVD recommender with a range of feature counts. Note that a feature count of 0 tells LensKit to use all features. It also runs the raw personalized mean recommender

and LensKit's item-item implementation with a moderate neighborhood size, so you can compare performance.

Run the evaluator as follows:

```
./gradlew evaluate
```

In the output (`build/eval-results.csv`), you will see the metrics over the two algorithms, and several feature counts.

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, feature count, and data set). Use these results to answer the following questions:

1. Looking at by-user RMSE and prediction nDCG, which version of SVD performs the best?
2. On what metric is SVD with global mean the best algorithm (for reasonably large model sizes)?
3. What is generally the best model size (latent feature count) to use for this data set (looking at multiple metrics)?
4. For this assignment, why is it important that we truncate the number of features? What would happen if we didn't perform any truncation? Hint: this has nothing to do with performance. If you don't know what the result of not truncating would be, try setting the feature count to 0 to tell LensKit to use all features. See what happens.

Submitting

1. Create a PDF file containing your answers to the evaluation questions.
2. Create a submission zip file with `./gradlew prepareSubmission`
3. Submit the resulting files to the TA (taijala@cs.umn.edu).

Going Further

If you would like to further explore the SVD recommender, there are a few things you can try:

- Import your item-item code and run it in the evaluator alongside the SVD code, to see how it compares (and compares to LensKit's item-item).
- Use the item-item scorer as the baseline for the SVD recommender. You'll need to actually do this with a FallbackItemScorer that falls back to the user-item personalized mean, since item-item can't produce a score for all user-item pairs.