Programming SVD Collaborative Filtering

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:

- Your SVD collaborative filtering implementation
- 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 (from Coursera)
- LensKit for Learning website
- LensKit evaluator documentation
- Project code JavaDoc
- 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 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 the baseline bias model (mean ratings). 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 bias model 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 `KeyIndex` objects 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 DAO, it takes two important configurable parameters:

- A feature count (`@LatentFeatureCount`); if 0, to indicate the SVD should not be truncated.
- A *bias model* (`BiasModel`) to provide mean ratings

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

When populating the matrix, subtract the appropriate bias from each rating. The `BiasModel`

stores the data needed for a user-item bias of the form $b_u i = b + b_u + b_i$, where b is obtained with the `getIntercept()` method, b_i from `getItemBias(i)`, and b_u from `getUserBias(u)`. User and item biases will be 0 when no data is available for that entity, so you can just request them and add them without handling missing-data cases.

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 bias model to get the initial values, and then add the scores computed from the SVD matrices to these base 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.

Running the Code

Run the recommender and predictor as you have done in previous assignments, using the `predict`and `recommend` Gradle tasks.

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

- `global`
- `item`
- `user`
- `user-item` (default)

Note that the singular value decomposition takes some time! Following are some expected model build times on different processors; this is the time required to build a single model, so the evaluator will take significantly longer.

Processor	JVM	Build time
2.4GHz Core i5-6300U	Oracle Java 1.8.0_92	2m10s
2.13GHz Atom D2701	OpenJDK 1.8.0_122	31 min
2.16GHz Pentium N3540	Java 1.8.0_102	6m45s
4GHz Core i7-6700K	Java 1.8.0_102	1m45s

Example Predict Output

Command:

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

Output:

```
predictions for user 320:
  153 (Batman Forever (1995)): 2.659
  260 (Star Wars: Episode IV - A New Hope (1977)): 4.189
  527 (Schindler's List (1993)): 4.064
  588 (Aladdin (1992)): 3.558
```

Example Recommend Output

Command:

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

Output:

```
recommendations for user 320:
  858 (Godfather, The (1972)): 4.365
  318 (Shawshank Redemption, The (1994)): 4.337
  4973 (Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)): 4.271
  7502 (Band of Brothers (2001)): 4.266
  1221 (Godfather: Part II, The (1974)): 4.263
  1248 (Touch of Evil (1958)): 4.252
  1203 (12 Angry Men (1957)): 4.246
  2859 (Stop Making Sense (1984)): 4.227
  2019 (Seven Samurai (Shichinin no samurai) (1954)): 4.205
  1939 (Best Years of Our Lives, The (1946)): 4.197
```

Running the Evaluator

Now that you have your recommender working, let's evaluate it. The `evaluate` task runs your evaluation as before. It runs your SVD recommender with a range of feature counts.

Note that a feature count of 0 tells your algorithm 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. What is the best variant of SVD for per-user RMSE?
2. What is the best variant of SVD for top-N nDCG?
3. When is SVD better than item-item?
4. On what metric is SVD with global mean the best algorithm (for reasonably large feature counts)?
5. What is generally the best feature count to use for this data set (looking at multiple metrics)?

Submitting

1. Create a submission `jar` file with `./gradlew prepareSubmission`
2. Submit the compiled `jar` to the Coursera grader
3. Answer the Coursera quiz about the evaluation results