Evaluating Recommenders

In this assignment, you will be using LensKit's evaluator to conduct offline evaluations of several recommender algorithms. You'll explore the relative performance of the various algorithms and write a new metric.

In this assignment, we will walk you through the process of measuring, tuning, and comparing algorithms using a suite of metrics. To make this something we can score, we've focused on the answers to the questions and tunings — we can't grade your process, but we can grade whether you come to the correct conclusions.

Running the code for this assignment can take some time. A completed script with all evaluations takes more than an hour on an Intel Core i5 laptop.

Downloads and Resources

- Project template (in Coursera)
- LensKit for Learning website
- LensKit evaluator documentation
- JavaDoc for included code
- Example analysis in R and Python

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.

Overview

The core of this assignment is doing a comparative offline evaluation of the following algorithms using the included data set:

- Global mean rating (predict only)
- Global popularity (number of ratings, recommend only)
- Item mean rating
- Personalized mean rating ($\mu + b_i + b_u$)
- Three variants of LensKit's user-based collaborative filtering implementation
- Two variants of LensKit's item-based collaborative filtering implementation
- Two variants of a content-based filter built on Apache Lucene

You will be evaluating them with the following metrics:

- Coverage (the fraction of test predictions that could actually be made)
- Per-user RMSE

- nDCG (over predictions, also called *Predict nDCG*; this is a *rank effectiveness* measure)
- nDCG (over top-N recommendation, also called *Top-N nDCG*)
- MRR
- MAP
- A diversity metric you will implement (entropy over item tags)

Your evaluation will use 5-fold cross-validation over the included set of ratings data. **For this assignment, always consider the mean of the metric results for each algorithm configuration.**

In addition, you will need some way of analyzing and plotting the evaluation output. The output is in CSV files, so any tool that can process them such as Excel, Google Docs, R, or Python will work. A sample IPython notebook file has been included to get you started, but you're free to use whatever tool you're comfortable with.

In this assignment, you will not submit any code. Instead, you will answer a quiz about the results of the evaluation.

Getting Started

Download the project template and extract it. This contains the normal `build.gradle` from previous projects, and a `src/main/java` directory. Its sources include the Java source for the Lucene-based recommender and the popularity-based recommender, as well as a skeleton file where you will implement your diversity metric.

We have also provided test cases for your diversity metric to help you test its correctness. The tests do not *guarantee* correctness, but will help detect common problems.

The project contains algorithm configuration files in the `cfg` directory. Currently, `baselines.groovy` provides configuration for the basic mean and popularity recommenders, and the other files provide support logic that are common to many recommender configurations. These files are written in Groovy, but you do not need to know Groovy to modify them for this experiment.

Out of the box, the experiment is configured to run the baseline algorithms. Run it with `./gradlew evaluate`, and it will put the analysis output in `build/eval-results.csv`. Use your favorite data analysis software to do a plot of the mean per-user RMSE (RMSE.ByUser) for each algorithm. This should result in a plot somewhat like that shown in Figure 1.

Or plot top-N nDCG @ 10 (the nDCG of a recommendation list of 10 items, excluding those rated by the user in the training set) and you should get something resembling Figure 2. Note that this graph includes *Popular* but excludes *GlobalMean*, as the former can't do rating prediction and the latter can't rank.

You can see the code to produce these plots in the example analysis notebook.

Since we are using 5-fold cross-validation, we have 5 values for each algorithm (the average RMSE, nDCG, etc. over all test users in each fold). We usually plot the mean of the values,

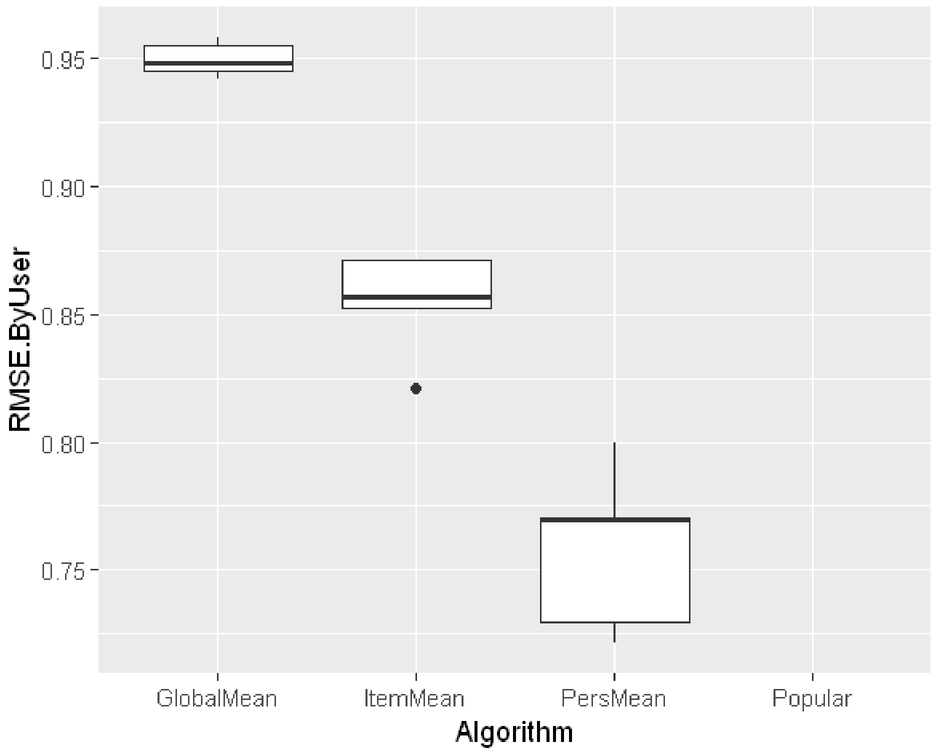

Figure 1: Plot of recommender RMSE

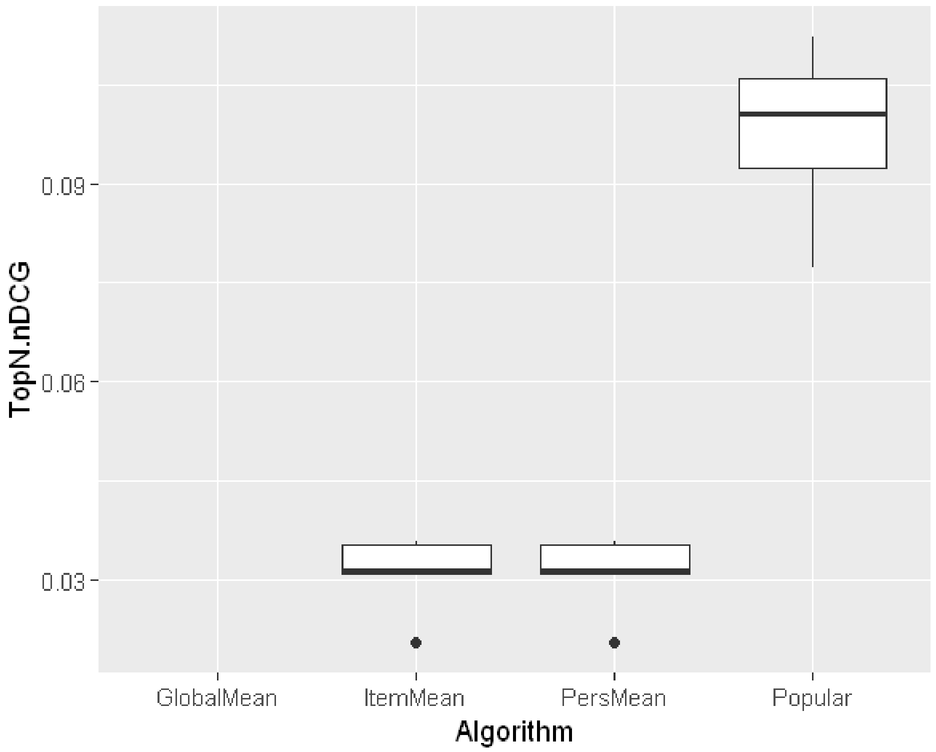

Figure 2: Plot of nDCG

sometimes with error bars.

Hint: If your plotting software makes it easy to include error bars, do so! However, to compute an error bar correctly, computing the standard error of the experimental runs is generally not sufficient. It is better to compute the standard error over the individual user result values. You can obtain these from the file `build/user-results.csv`.

Tuning User-User CF

Next, let's add the user-user collaborative filter and tune its neighborhood size and normalization. We will test 2 variants of user-user (one averaging raw ratings, the other averaging mean-centered ratings) across a range of neighborhood sizes.

Create a new file, `cfg/user-user.groovy`, and write the following:

```
for (nnbrs in [5, 10, 15, 20, 25, 30, 40, 50, 75, 100]) {
    algorithm("UserUser") {
        include 'fallback.groovy'

        // Attributes let you specify additional properties of the algorithm.
        // They go in the output file, so you can do things like plot accuracy
        // by neighborhood size
        attributes["NNbrs"] = nnbrs

        // use the user-user rating predictor
        bind ItemScorer to UserUserItemScorer

        set NeighborhoodSize to nnbrs

        bind VectorSimilarity to PearsonCorrelation
    }

    algorithm("UserUserNorm") {
        include 'fallback.groovy'

        // Attributes let you specify additional properties of the algorithm.
        // They go in the output file, so you can do things like plot accuracy
        // by neighborhood size
        attributes["NNbrs"] = nnbrs

        // use the user-user rating predictor
        bind ItemScorer to UserUserItemScorer

        set NeighborhoodSize to nnbrs
        bind VectorNormalizer to MeanCenteringVectorNormalizer
        bind VectorSimilarity to PearsonCorrelation
    }
}
```

This adds two algorithm definitions for each neighborhood size. One uses a vector normalizer (`MeanCenteringVectorNormalizer`), the other does not. Each definition also stores the neighborhood size in an *attribute*; this will turn into an extra column in the CSV file, so you can use the neighborhood size in data analysis and plotting.

Edit `build.gradle` and add an `algorithm 'cfg/user-user.groovy'` line to the `evaluate` task, right after the existing `algorithm 'cfg/baselines.groovy'` line.

Hint: Right-click the `cfg` directory in IntelliJ and 'mark' it as a 'Sources Root' to get IntelliJ to help you complete the imports.

Run your evaluation with `./gradlew clean evaluate` and see the results!

Let's also consider using cosine similarity instead of Pearson correlation. Create a copy of the mean-centering algorithm (`UserUserNorm`), give it a new name (such as `UserUserCosine`), and change the similarity function:

```
bind VectorSimilarity to CosineVectorSimilarity
```

This line should replace the existing `VectorSimilarity` configuration. The mean-centering normalization will already be used for both rating averages and pre-similarity normalization, since we didn't specify any restrictions on where it is to be used. This was fine for Pearson, since mean-centering does not change the correlation at all, and means we don't need any additional configuration to make the cosine similarity be computed over mean-centered data.

Adding Item-Item

To set it up, write a file called `cfg/item-item.groovy` with a `for` loop like the one in `user-user`, and use the following configuration in that loop to configure an item-item recommender using item-mean centering for its rating normalization:

```
algorithm("ItemItem") {
    attributes["NNbrs"] = nnbrs
    include 'fallback.groovy'
    bind ItemScorer to ItemItemScorer
    bind VectorSimilarity to CosineVectorSimilarity
    set NeighborhoodSize to nnbrs

    bind UserVectorNormalizer to BiasModelUserVectorNormalizer
    within (UserVectorNormalizer) {
        bind BiasModel to UserBiasModel
    }
}
```

And create a user-normalized version by replacing the bias model configuration with the following:

```
bind BiasModel to ItemBiasModel
```

Add your configuration file to `build.gradle`, and your experiment should now take longer but also produce item-item CF results.

Adding Lucene

Let's now add the Lucene recommender. This recommender uses Apache Lucene, a text retrieval and search library, to measure how similar different movies are based on their tags. It scores movies by looking at similar movies the user has rated, and computing the weighted average of the user's ratings of those similar movies (using the score reported by Lucene as the weight). It has a parameter, the neighborhood size, that determines how many similar movies to consider when generating a score.

To set it up, write a file called `cfg/lucene.groovy` like your user-user and item-item files, and include the following configuration:

```
algorithm("Lucene") {
    attributes["NNbrs"] = nnbrs
    // use fallback scorer for unscorable items
    include 'fallback.groovy'
    bind ItemScorer to ItemItemScorer
    bind ItemItemModel to LuceneItemItemModel
    set NeighborhoodSize to nnbrs
}
```

We can also consider a variant of this algorithm that considers how much more or less the user likes each movie than the average user:

```
algorithm("LuceneNorm") {
    attributes["NNbrs"] = nnbrs
    include 'fallback.groovy'
    bind ItemScorer to ItemItemScorer
    bind ItemItemModel to LuceneItemItemModel
    set NeighborhoodSize to nnbrs

    // normalize user rating vectors by subtracting biases
    bind UserVectorNormalizer to BiasUserVectorNormalizer
    // subtract the item bias (so we normalize by item mean rating)
    within (UserVectorNormalizer) {
        bind BiasModel to ItemBiasModel
    }
}
```

This variant subtracts the item's average rating for each movie from the user's rating before computing the weighted average. It might perform better, let's see!

Once you have this configuration added (including adding `algorithm 'cfg/lucene.groovy'` to `build.gradle`), run your evaluation again.

Writing a Metric

The file `TagEntropyMetric.java` is a skeleton for a new evaluation metric that will measure the diversity of recommendations. There are many ways to measure diversity, but for this assignment we will use the *entropy* of the tags of the items in a top-10 recommendation list. Entropy is, roughly, a measurement of how complicated it is to say which one of several possibilities has been picked. If there are many different tags represented among the movies, they will have high entropy; if there are very few tags, entropy will be low. We will use high entropy as an indication that the set of movies is diverse.[1]

The entropy of a set X of things x, each of which has a probability $P(x)$, is defined as

$$H(X) = \sum_{x \in X} -P(x) \log_2 P(x)$$

To compute the entropy of the *tags* seen on a list of recommendations, we need to define $P(t|L)$: the probability of a tag t for a particular recommendation list L.

To do that, we will compute the probability of selecting a particular tag t according the following process:

1. Pick a movie at random from the list of recommended movies.
2. Pick a tag at random from the movie's tags, with probability proportional to the number of times the tag has been applied to the movie.

So, for a tag t and recommendation list L, $P(t)$ is defined as follows (where T_m is the set of tags of movie m and n_{mt} is the number of times tag t was applied to movie m):

$$\begin{aligned} P(t|L) &= \sum_{m \in L} P(t|m) P(m|L) && (1) \\ &= \sum_{m \in L \text{ s.t. } t \in T_m} \frac{n_{mt}}{\sum_{t' \in T_m} n_{mt'}} \frac{1}{|L|} && (2) \end{aligned}$$

[1]If you want to learn more about entropy, read *An Introduction to Information Theory: Symbols, Signals, and Noise* by John R. Pierce.

Plug this into the entropy formula to compute the tag entropy of a list L of recommendations, with T being the set of all tags appearing on any movie in L:

$$H(L) = \sum_{t \in T} -P(t|L) \log_2 P(t|L) \tag{3}$$

The downloaded code also includes `TagEntropyMetricTest`, a set of test cases for the tag entropy metric to help you debug your code. You can run these tests with the following command:

```
./gradlew test
```

In order to get the tags for an item, you will need to get 'item tag' entities for the item. You can do this with the code:

```
List<Entity> itemTags = dao.query(TagData.ITEM_TAG_TYPE)
                            .withAttribute(TagData.ITEM_ID, res.getId())
                            .get();
for (Entity itag: itemTags) {
    String tag = itag.get(TagData.TAG);
    // do something with the tag
}
```

Submitting

There is no code or data to submit. Instead, visit the Evaluation Quiz and answer the questions it contains based on your analysis of the evaluation results.