Programming Assignment 4

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.

We have two main learning objectives for this week's assignments. First, we want you to understand the applicability of different metrics and evaluations to different situations; that is the goal of the coursera assignment. Second, we want you to gain experience using evaluation to tune recommender algorithms; that is the goal of 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 (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; I 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-user collaborative filtering implementation
- A content-based filter built on Apache Lucene, in both normalized and unnormalized variants

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*)
- nDCG (over top-N recommendation, also called *Top-N nDCG*)
- A diversity metric you will implement (entropy over item tags)

Your evaluation will use 5-fold cross-validation over the included ratings data set. **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 submit a report showing the results of your evaluations and answering a series of questions.

Getting Started

Download the project template and extract it. This contains the normal `build.gradle` from previous projects, and a `src/main/java` directory. It also includes 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.

The project contains an algorithm configuration file `algorithms.groovy`. This file is used to tell LensKit what algorithms you want to test and how they should be configured. This file is written in Groovy, but you do not need to really know Groovy to modify the file for this experiment.

This file already includes definitions for the first several algorithms (everything except user-user and Lucene). 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 `ExampleAnalysis.html`.

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, sometimes with error bars.

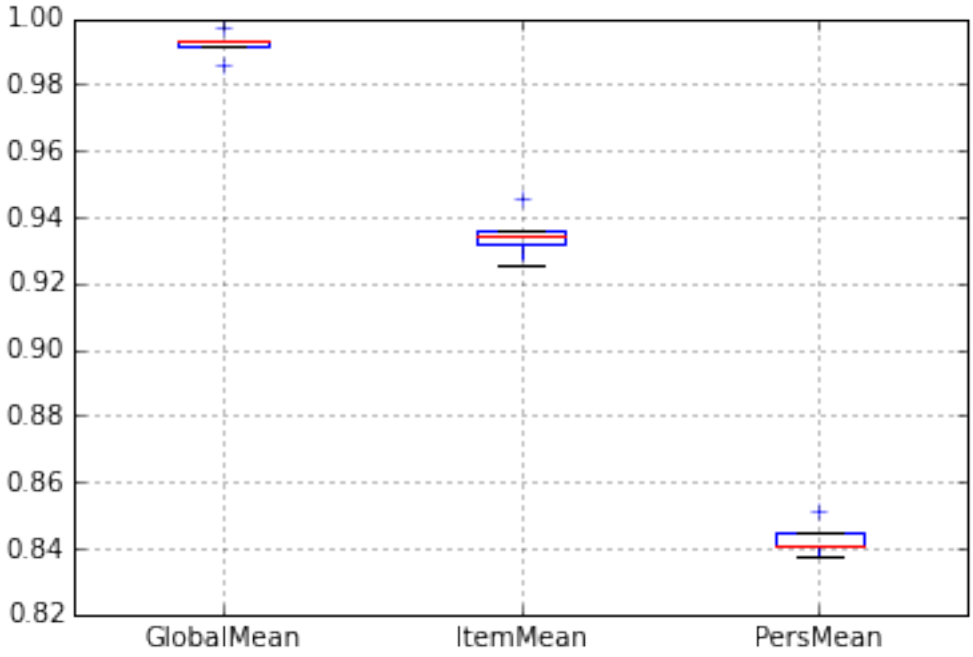

Figure 1: Plot of recommender RMSE

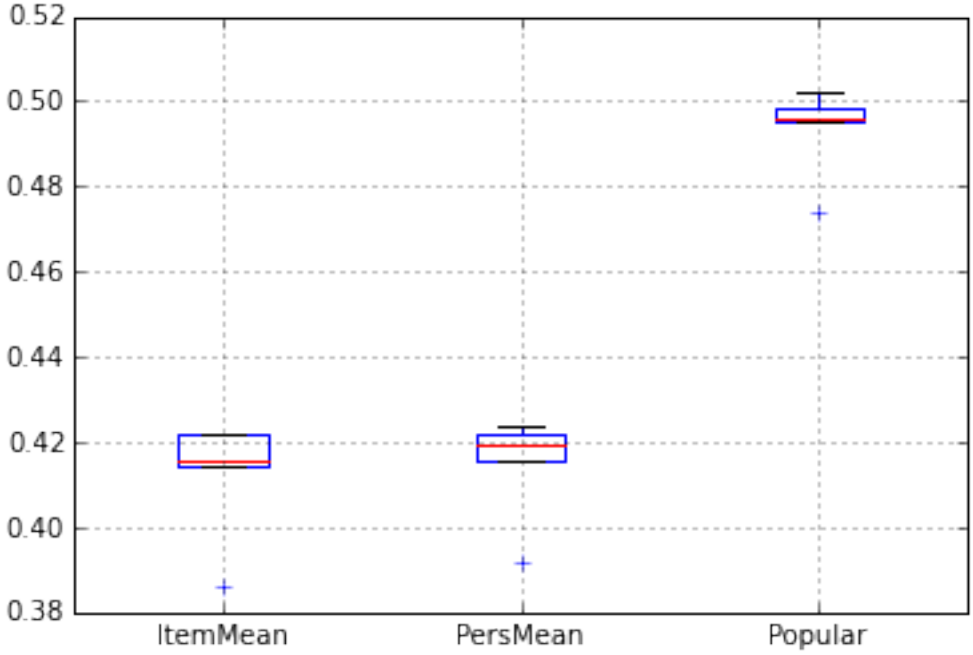

Figure 2: Plot of nDCG

Tuning a Recommender

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 'tag-setup.groovy'
        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 'tag-setup.groovy'
        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.

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 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 moves 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` with a `for` loop like the one in `user-user`, and use the following configuration in that loop:

```
algorithm("Lucene") {
    attributes["NNbrs"] = nnbrs
    include 'tag-setup.groovy'
    include 'fallback.groovy'
    bind ItemScorer to ItemItemScorer
    bind ItemItemModel to LuceneItemItemModel
    set NeighborhoodSize to nnbrs
    // consider using all 100 movies as neighbors
    set ModelSize to 100
}
```

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 'tag-setup.groovy'
    include 'fallback.groovy'
    bind ItemScorer to ItemItemScorer
    bind ItemItemModel to LuceneItemItemModel
    set NeighborhoodSize to nnbrs
    // consider using all 100 movies as neighbors
    set ModelSize to 100

    bind UserVectorNormalizer to BaselineSubtractingUserVectorNormalizer
    within (UserVectorNormalizer) {
        bind (BaselineScorer, ItemScorer) to ItemMeanRatingItemScorer
    }
}
```

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.

The entropy of a set of things x, each of which has a probability $P(x)$, is defined as $\sum_x -P(x)\log_2 P(x)$. 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 by the following process: first, pick a movie at random from the list of recommended movies. Then pick a tag at random from the set of that movie's tags (ignoring how many times the tag was applied to the movie). So, for a tag t and recommendation list L, $P(t)$ is defined as follows (where T_m is the tags of movie m):

$$P(t|L) = \sum_{m \in L} P(t|m)P(m|L) \tag{1}$$

$$= \sum_{m \in L \text{ s.t. } t \in T_m} \frac{1}{|T_m|}\frac{1}{|L|} \tag{2}$$

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:

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

We've provided the `TagVocabulary` class to make it easier to write this metric. It allows you to create tag vectors and map tags to their IDs. Unlike the tag processing in Assignment 2, this time we are ignoring the case of tags; you will not have to do anything to account for this, though; the `ItemTagDAO` interface now returns all-lowercase tags.

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
```

Submitting

There is no code or data to submit. Instead, write a brief report answering the following questions. For each question, you must include one or two plots/graphs if they form the basis of your answer; you must provide some justification for your answers. Some options are multiple choice. Submit your report as a **single PDF file** on TRACS.

1. Why do PersMean and ItemMean have the same nDCG values?
2. How many neighbors are needed for unnormalized Lucene to match or beat the performance of personalized mean on RMSE? Your answer can be one of the tested number of neighbors.
3. How many neighbors are needed for user-user w/ cosine to match or beat the performance of normalized Lucene on prediction nDCG? Your answer can be one of the tested number of neighbors.
4. How many neighbors are needed before unnormalized Lucene's RMSE results begin to stabilize? For normalized Lucene?
5. For the algorithms that have a neighborhood size parameter, does increasing neighborhood size generally increase or decrease the tag entropy of top-10 lists?
6. What algorithm produces the least diverse Top-10 lists (by our entropy metric) throughout the range of neighborhood sizes?
7. Why is Top-N nDCG lower than prediction nDCG?
8. In practice, recommenders cost money to run and it isn't worthwhile to run recommenders that take a lot of computational power and provide little benefit. Based on

this experiment, what algorithm would be best to deploy for recommending items (in ranked lists) from this data set?

9. How many neighbors are needed before unnormalized Lucene's tag entropy results begin to stabilize? For normalized Lucene?
10. Ignoring entropy, what user-user configuration generally performs the best?
11. What algorithms have higher recommendation list tag entropy than Popular?
12. What is the Top-N nDCG of Popular?