Programming Hybrid Recommenders

In this assignment, you will create and optimize a hybrid recommender for LensKit. This recommender will blend content-based filtering with collaborative filtering to produce recommendations.

There are four parts to the assignment:

1. Implement a configurable blending hybrid recommender.
2. Run an experiment to tune the recommender's blending weights.
3. Implement a self-tuning logistic regression hybrid recommender.
4. Run an experiment to compare the logistic regression hybrid recommender to the individual component recommenders.

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.

Changelog

21 April Consistently use α instead of β_0
Change names of tasks from `runFoo` to `recommendFoo`
20 April Change δ to Δ in update rules
Fix additional notational errors
Fix linear blend score rule and document how to handle missing component scores
Add notes on math/Java mapping and improve logistic regression notation

Resources

- Project template (on course website)
- LensKit for Learning website
- LensKit evaluator documentation
- JavaDoc for included code

Layout

The project includes sources for your recommenders and configuration files to configure the following recommenders:

- LensKit's FunkSVD matrix factorization recommender
- The Lucene-based CBF recommender from Module 3
- LensKit's item-item collaborative filter

- Your hybrid recommender

Linear Blending Hybrid

Implementing the Hybrid

Finish implementing the class `LinearBlendItemScorer`. It takes three inputs:

`@Left ItemScorer left` The 'left-hand' item scorer (in our configurations, FunkSVD).
`@Right ItemScorer right`
The 'right-hand' item scorer (in our configurations, Lucene). `@BlendWeight double weight`
The weight $0 \leq \alpha \leq 1$ for blending the item scorers.

The class must score items using the following formula:

$$s(u,i) = b_{ui} + (1-\beta)(s_{\text{left}}(u,i) - b_{ui}) + \beta(s_{\text{right}}(u,i) - b_{ui})$$

If one of the component scorers cannot score the item (returns `null`), then treat its score *offset* as 0; that is, assume $s_x(u,i) - b_{ui} = 0$. If both are null, then your scorer will just return the baseline score b_{ui}.

Running the Recommender

Run the recommender as follows:

```
./gradlew recommendLuceneSVD -PuserId=42 -PblendWeight=0.5
```

Example output:

```
recommendations for user 320:
  318 (Shawshank Redemption, The (1994)): 4.342
  2859 (Stop Making Sense (1984)): 4.337
  858 (Godfather, The (1972)): 4.251
  4973 (Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)): 4.250
  2019 (Seven Samurai (Shichinin no samurai) (1954)): 4.241
  926 (All About Eve (1950)): 4.238
  7502 (Band of Brothers (2001)): 4.234
  1248 (Touch of Evil (1958)): 4.233
  1203 (12 Angry Men (1957)): 4.224
  2360 (Celebration, The (Festen) (1998)): 4.201
```

Due to the randomness in training the FunkSVD algorithm, the exact scores will differ somewhat from run to run, but they should be close.

Optimizing the Blend

The Gradle task `sweepHybrid` will run an experiment that evaluates the recommender with several different blending weights.

Examine the results and select the best blending weights for RMSE and MRR. Answer the quiz questions.

Logistic Recommender

For the next part of the assignment, you will implement a logistic regression hybrid recommender that uses several component recommenders to optimize the recommender's ability to predict whether or not the user will *rate* the item.

Logistic Regression

The formula for a logistic regression is:

$$\Pr[y_{ui} = 1] = \text{sigmoid}(\alpha + \beta_1 x_1 + \cdots + \beta_n x_n)$$

$y_{ui} = 1$ if user u has rated item i, and $y_{ui} = -1$ otherwise. α is the *intercept*, and $\beta_1, \ldots, \beta_n$ are *coefficients* that weight the *explanatory variables* $x_1, \ldots, x_n$. The β_i values are represented in Java as the `coefficients` on the logistic model and the `params` array in your training code. The sigmoid function $\text{sigmoid}(t) = \frac{1}{1+e^{-t}}$ lets us map scores into probabilities, and is available as a static method on the logistic model class.

To train the logistic regression model, you will use stochastic gradient ascent to learn α and β_i values that maximize the *log likelihood*: $\ln \Pr[y_{ui}]$ across the data.

In the stochastic gradient ascent, you will use the following update rules:

$$\Delta\alpha = \lambda y_{ui} \text{sigmoid}(-y_{ui}(\alpha + \beta_1 x_1 + \cdots + \beta_n x_n))$$
$$\Delta\beta_j = \lambda y_{ui} x_j \text{sigmoid}(-y_{ui}(\alpha + \beta_1 x_1 + \cdots + \beta_n x_n))$$

λ is the *learning rate*; it is defined by the `LEARNING_RATE` constant in the `LogisticModelProvider` class.

Training Logistic Regression

Implement the logistic regression training using stochastic gradient ascent in `LogisticModelProvider`.

The `LogisticModelProvider` class receives several things as inputs:

- A `LogisticTrainingSplit` that contains the ratings used to train the subsidiary recommenders and to tune the logistic blend.
- A bias model for preference biases.
- A `RatingSummary` for providing rating counts (for popularity).
- A `RecommenderList` containing component recommenders.

The training split's `getTuneRatings()` method will give you a list of `Rating` objects representing the tuning ratings; these are modified so that they have a value of 1 for items the user has rated, and -1 for items they have not. You can use their values directly as the y_{ui} values.

For your logistic regression, the explanatory variables for each tuning rating are as follows:

- x_1 is the bias b_{ui} from the bias model.
- x_2 is the log popularity of the item, $\log_{10}|U_i|$ (get this from the rating summary).
- x_3 through x_n are the scores produced by each recommender in the `RecommenderList`, minus the bias. So $x_3 = s_1(i;u) - b_{ui}$.

Use `ITERATION_COUNT` total iterations; for each iteration, loop over the tune ratings and update the logistic regression for each rating in turn. Randomize the tuning ratings (using `Collections.shuffle`) at the beginning of each iteration. `LogisticModel.create` will make a copy of its parameters, so you can call it and then update the arrays for the next data point without needing to re-allocate or copy arrays yourself.

One way to structure that code is to create a working array that you will use to update the current parameter values, and after each update pass it to `LogisticModel.create` to make an updated model for computing the next update. It is also helpful to cache the scores from each component recommender instead of recomputing them for each update.

The `LogisticModel.evaluate` function will compute the value of the logistic regression model for specified inputs and output. It takes care of the sigmoid function for you as well. You can use it both in training and in scoring.

Using Logistic Regression

Implement `LogisticItemScorer` to compute item scores using your logistic regression and the rating counts, bias model, and configured recommenders.

Running Logistic Regression

The `recommendSomeAlgoLog` Gradle tasks, such as `recommendSVDLog`, will run your logistic hybrid with one or more underlying algorithms.

The `recommendLuceneSVDLog` task produces the following output in one particular run for user 320:

```
recommendations for user 320:
  318 (Shawshank Redemption, The (1994)): 0.817
  356 (Forrest Gump (1994)): 0.803
  7153 (Lord of the Rings: The Return of the King, The (2003)): 0.801
  593 (Silence of the Lambs, The (1991)): 0.781
  858 (Godfather, The (1972)): 0.770
  1270 (Back to the Future (1985)): 0.764
  1210 (Star Wars: Episode VI - Return of the Jedi (1983)): 0.758
  58559 (Dark Knight, The (2008)): 0.752
  1 (Toy Story (1995)): 0.745
  6874 (Kill Bill: Vol. 1 (2003)): 0.744
```

Another run yielded:

```
recommendations for user 320:
  318 (Shawshank Redemption, The (1994)): 0.823
  7153 (Lord of the Rings: The Return of the King, The (2003)): 0.808
  356 (Forrest Gump (1994)): 0.805
  593 (Silence of the Lambs, The (1991)): 0.786
  858 (Godfather, The (1972)): 0.777
  1270 (Back to the Future (1985)): 0.772
  1210 (Star Wars: Episode VI - Return of the Jedi (1983)): 0.765
  58559 (Dark Knight, The (2008)): 0.762
  6874 (Kill Bill: Vol. 1 (2003)): 0.752
  4973 (Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)): 0.752
```

Due to the random aspects of the process, particularly in splitting the training and tuning data, these outputs will vary from run to run. It should contain most of the same movies, but they will be in slightly different orders and the scores may vary, within a range of about 0.05. Our auto-grading will account for this.

Evaluating Logistic Regression

Run `./gradlew evaluateLogistic` to run an experiment comparing various versions of the logistic regression recommender. Use this to answer questions in the second Evaluation Quiz.

Submitting

1. Create a compiled `jar` file with `./gradlew prepareSubmission` and submit it to the Coursera assignment grader.
2. Answer the questions in the *Evaluation Quiz*.