CS2112—Fall 2012

Assignment 1

Radio Recommender System

Due: Tuesday, September 4, 11:59PM

This assignment is intended to refresh your knowledge of Java and the Java API. You will write a complete application that performs I/O and handles errors. You will need to design data representations and implement algorithms to process the data. It is roughly equivalent to a comprehensive CS1110 assignment.

In this assignment, you are building a music recommendation system. In recent years, there has been much research into how to recommend products to users based on the ratings of other, similar users. This application might be used by Internet radio stations such as Pandora, Grooveshark, and Spotify. We describe the key recommendation algorithms below; your job is to implement them as part of a larger program.

0 Instructions

0.1 Grading

Solutions will be graded on both correctness and style. A correct program compiles without errors or warnings, and behaves according the the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variables names and proper indentation. Your code should include comments as necessary to explain how it works, but without explaining things that are obvious.

0.2 Partners

You *must* work alone for this assignment. But remember that the course staff is more than happy to help with problems you run into. Use Piazza for questions, attend office hours, or set up a meeting with any course staff member.

0.3 Updates

Keep an eye on this section for a summary of any updates that have occurred since the initial release of the assignment.

1 Familiarizing yourself with the code

1.1 Building the documentation

Open the provided code in Eclipse. Follow the instructions at

http://www.cis.upenn.edu/~matuszek/General/Pages/eclipse-faq.html#run_javadoc

to build Javadoc for this project from within Eclipse. When the documentation finishes building, you should be able to view all the documentation from your web browser.

We have given you some documentation so that you can get started on the assignment. Feel free to expand upon this documentation in your final submission.

1.2 Project structure

You will build four classes for this assignment:

Station represents and stores information relevant to a radio station. This information includes, but not limited to, a name, a unique station ID, and the number of songs it has played so far.

Song represents and stores information relevant to a song. This information includes, but not limited to, a name, an artist, a unique song ID, and an array containing how many times the song is played on each radio station.

Parser parses the files containing the information about radio stations, songs, and song logs. After parsing, it constructs the list of songs and radio stations. The song logs are handled in a separate method.

RadioRecommenderSystem provides an interactive console, recommends songs for radio stations, displays song statistics, and calculates similarity between songs or radio stations. The class contains the main method which loops indefinitely asking for user inputs.

2 Stations and songs

2.1 Input files

The homework's zip file contains a list of stations, a list of songs, and a song log. The first line of the station file indicates the number of radio stations r. The remaining r lines are semicolon-delimited lines describing a single radio station in the format `<Station Name>;<Station ID>`. The station IDs are arbitrary nonnegative integers and may be given in any order. There may be additional spaces surrounding a station name. Your task is to trim these spaces so that each name begins and ends with non-space characters. For instance, the following represents a valid station file, where ␣ indicates a space.

```
2
␣␣Radio␣Disney␣␣␣␣␣;42
␣␣␣␣BBC␣␣␣;17
```

There are two stations described above. One is "Radio Disney" (after trimming spaces) with an ID of 42 and the other is "BBC" with an ID of 17.

The song file is similar to the station file, where the first line indicates the number of songs s and the remaining s lines are in the format `<Song Name>;<Song Artist>;<Song ID>`. The song IDs are arbitrary nonnegative integers and may be given in any order. There may be additional spaces surrounding a song name or an artist and you need to trim these spaces. For instance, the following represents a valid song file.

```
3
Thriller␣␣;␣␣Michael␣Jackson␣␣␣␣␣␣␣␣␣;25
␣␣␣␣␣␣␣Skeleton␣Boy;␣Friendly␣Fires␣␣␣;47
Rainbow␣in␣the␣Dark;Das␣Racist;12
```

In our example set, the first line tells us that there are three songs in this file. The second song has an ID of 47, is named Skeleton Boy, and was performed by Friendly Fires.

The song log file contains songs played at radio stations in chronological order. Each line of the log has the format `<Station ID>;<Song ID>`. For simplicity, the song indicated in line i of the log was played at time i. You may assume that exactly one station plays one song for each time period. That is, if there are n lines in the log, for any i satisfying $1 \leq i \leq n$, exactly one station plays exactly one song at time i. The following represents a valid song log file.

```
42;25
42;25
17;12
42;25
42;25
42;25
42;25
17;47
42;27
```

In the above log, BBC plays Rainbow in the Dark at time 3 and Skeleton Boy at time 8. Radio Disney plays Thriller at time $1, 2, 4, 5, 6, 7$. Perhaps BBC is an indie-rock radio station, and Radio Disney is likely a Michael Jackson tribute station.

Keep in mind that we want to characterize stations by their playlists so that we can make suggestions for other similar songs to play on these stations.

You may assume that every file we will provide (and use to test) will be formatted correctly.

2.2 Class: Station

The `Station` class has a constructor, a name, an ID, and an instance variable that stores the length of the playlist on this station. Create appropriate accessor methods for this class. Insert additional

instance variables as necessary to make the recommendation process—described below—efficient.

Furthermore, rewrite the `toString()` method to return the following string-based representation of a station: "`<Station ID>. <name>`", such as "`17.␣BBC`".

2.3 Class: Song

The `Song` class has a constructor, private instance variables for the name, artist, and ID of the song, and appropriate accessors for these variables. It also has an array `stationPlays`, a list of the number of times this song is played on each radio station's playlist. This could be zero or more times per station. Insert additional instance variables as necessary to make the recommendation process efficient.

Furthermore, rewrite the `toString()` method to return the following string-based representation of a song: "`<Song ID>. <artist> - <name>`", such as:

`47.␣Friendly␣Fires␣-␣Skeleton␣Boy.`

There are two additional public methods for you to implement. `getStatistics()` returns an array of length 6 containing basic statistics about this song. This array should contain the following fields in this order:

0. The average number of plays of this song on stations that carry it. That is, we only calculate stations that play this song at least once. Radio Disney plays Thriller 6 times but BBC never plays it. Therefore, the average number of times Thriller is played is 6 times across all the stations that carry it.
1. The total number of plays across all stations.
2. The ID of the station that plays this song the most. If there is a tie, this should be the lowest station ID.
3. Maximum number of plays on any one station.
4. The ID of the station that plays this song the least. If there is a tie, this should be the highest station ID.
5. Minimum number of plays on any one station. This could be zero.

`getLastPlayed()` takes a station ID and returns the last time step this song is played on the given station, or zero if this song has never played on the given station. In the above example, calling the method on *Skeleton Boy* object with BBC, we should receive 8 in return.

2.4 Class: Parser

The Parser class is responsible for turning the datasets into arrays of Song objects and Station objects containing the relevant information including that from the song log. The constructor should process station and song information. A separate method `processSongLog()` handles the

song log. While there are many ways to handle File I/O, we recommend that you use a class that works well with newlines and custom delimiters. It is inefficient to work character by character the whole time and we will penalize submissions that read the files in that fashion.

You should assume that the number of queries to be performed when the recommendation system is running far exceeds the numbers of stations and songs combined. When storing the information, therefore, you should make the station and song lookups as efficient as possible. In this project, you are not allowed to use classes in `java.util` package such as `HashMap`. Arrays should be the primary storage for this assignment.

3 Class: RadioRecommenderSystem

This class will handle most of the data manipulation in order to get our recommendation system running. You will have to remember to throw and catch exceptions where appropriate.

3.1 Similarity methods

There are two methods that calculate the similarity of two objects: `songSimilarity()` and `stationSimilarity()`. These methods return a double between 0 and 1 (inclusive) which measures how similar two songs or stations are. The higher the value, the more similar the two objects. It follows that a song or a station has a similarity of 1.0 to itself.

The method we will use for measuring similarity is known as *cosine similarity* and is described in Section 3.1.2 of [1]. For example, suppose song A has the `stationPlays` vector $[a_1, a_2, \ldots, a_r]$ and song B has the `stationPlays` vector $[b_1, b_2, \ldots, b_r]$. The similarity between them is defined by

$$\frac{a_1b_1 + a_2b_2 + \cdots + a_rb_r}{\sqrt{a_1^2 + a_2^2 + \ldots + a_r^2}\sqrt{b_1^2 + b_2^2 + \cdots + b_r^2}}.$$

The same formula applies to station similarity, where the vector used contains the number of times each song is played at a particular station.

For example, suppose there are five songs, Station A has played songs $4, 5, 3, 4, 4, 5, 1$, and Station B has played songs $1, 1, 4, 2, 3, 2, 2, 5, 2$. These two stations would have the following similarity:

$$\frac{1 * 2 + 0 * 4 + 1 * 1 + 3 * 1 + 2 * 1}{\sqrt{1^2 + 0^2 + 1^2 + 3^2 + 2^2}\sqrt{2^2 + 4^2 + 1^2 + 1^2 + 1^2}} = 0.431.$$

The two methods `closestStation()` and `closestSong()` should return the ID of the station or song ID that is most similar to the station or song corresponding to the ID provided. In case of a tie, return the lowest ID.

If a station has an empty playlist and similarity is done against it, then `stationSimilarity()` should return zero no matter what the other station is, as we can make no assumptions out of no information. Similarly, if a song has not been played on any station, then `songSimilarity()` should return zero no matter what the other song is.

Useful resources

The paper cited is available through the ACM portal. You may need to be on campus to access this link.

3.2 Building a recommendation system

All the building blocks are in place for your recommendation system. Believe it or not, you have actually completed the hardest parts! All that remains is to use what you have constructed to build a recommendation system.

We adapt the algorithm available in Section 3.2.1 of [1]. It takes the weighted average of the number of plays of all songs *other than* the radio station for which a recommendation is being made. Then, we apply a multiplier to the resulting value from the original algorithm so that songs played less recently on that station are more likely to be chosen. Specifically, suppose there are four radio stations. To recommend song s to station r_3, the system computes

$$\left(e^{-1/\sqrt{T_S}}\right)\left(a_3 + \frac{(c_1 - a_1)\cdot sim(r_1, r_3) + (c_2 - a_2)\cdot sim(r_2, r_3) + (c_4 - a_4)\cdot sim(r_4, r_3)}{sim(r_1, r_3) + sim(r_2, r_3) + sim(r_4, r_3)}\right),$$

where

- T_S is the elapsed time since the most recent play of s on station r_3, calculated as follows. Suppose all the stations have played N songs in total and song s was heard last time on r_3 at time t_s, then $T_S = N + 1 - t_s$.
- a_i is the average number of times any song is played at station r_i, including those played zero times.
- c_i is the number of times song s has been played at station r_i.
- $sim(r_i, r_j)$ is the similarity between stations r_i and r_j.

This algorithm should be implemented in the method

`makeRecommendation(intrecSongID, intradioID)`

The `bestRecommendation(int radioID)` method should return the song which is the best recommendation for the given radio station. In case of a tie, return the song whose most recent play on the given station was earliest in time.

Useful resources

- The paper referenced from the previous section.
- You may find it useful to create a small dataset to test your formulas.

3.3 The `main` Method

The last piece of this program is the main method. It receives the path to the directory where the three files are located as the first argument to the program. The main method should first ask for the names of the files containing the station and song lists and then create a parser with the folder path and the names of the above two input files. Finally, it creates a `RadioRecommenderSystem` instance. Finally, it should loop indefinitely asking for user input, until the command "exit" is issued.

The user may provide any of the following eight commands via the console:

- `importlog <logFilename>`: Appends the current song log with the information provided in the given file. This file is assumed to be in the same folder as given to the main method. The time steps should continue from the current log. For instance, if N songs has been played in total, the first song in the new log is played at time $N + 1$.
- `similarsong <song ID>`: Finds and prints the most similar song to the chosen song.
- `similarradio <station ID>`: Finds and prints the most similar radio station to the chosen station.
- `stats <song ID>`: Prints the statistics of the chosen song.
- `lastheardon <station ID> <song ID>`: Finds and prints the most recent time the given song is played on the given station.
- `lastplayed <song ID>`: Finds and prints the most recent time the given song is played on any station.
- `recommend <station ID>`: Recommends a song to the chosen station.
- `exit`: Exits the program.

If at any point an unrecognized or invalid command is written, the program should output a helpful message explaing what is wrong, and request input again. If no songs have been played, the user should be informed and the program should continue to request input.

4 Extensions

For full credit, you are not required to do anything more than what is specified here. But for good karma you may add additional features. You might allow additional commands, give useful error messages when parsing incorrectly formatted input, support alternative recommender algorithms, or even produce graphical output. Or something else that you think is cool. You only get good karma, not points, so if you do go beyond what is required, make sure that your extensions do not interfere with required functionality. Be sure to tell us about any extensions in `README.txt`.

5 Submission

You should compress exactly these files into a `zip` file that you will then submit on CMS:

- `README.txt`: This file should contain your name, your NetID, all known issues you have with your submitted code, and the names of anyone you have discussed the homework with.
- `Song.java`
- `Station.java`
- `Parser.java`
- `RadioRecommenderSystem.java`

Do not include any files ending in `.class`.

All `.java` files should compile and conform to the prototypes we gave you. We write our own classes that use your classes' public methods to test your code. *Even if you do not use a method we require, you should still implement it for our use in testing your code.*

[1] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009:4:2-4:2, January 2009.