COMP9318 (15S1) PROJECT

DUE DATE: 23:59 26 APR 2015 (SUN)

1. Overview

In this project, you are required to implement a simplified recommendation system using UV-decomposition and similarity search based on the RP-tree.

The input is a training data in the form of a sparse utility matrix $R_{n\times m}$. It is made up of ratings of n users (i.e., rows) on m items (i.e., columns)[1]. You need to first decompose it into two matrices: $P_{n\times f}$ and $Q_{f\times m}$, for f factors. You need to implement a specific version stochastic gradient descent algorithm to compute a good decomposition. Then, given that each item can be represented as a f-dimensional vector, you need to find top-3 approximate nearest neighbors for a given item in terms of Euclidean distance. You need to implement the RP-tree index to support such operations efficiently.

2. Task I: Stochastic Gradient Descent

You are required to write a Java class named `SGD`, and the following **nine** parameters will be given from the command line:

- n is the number of users.
- m is the number of items.
- f is the number of factors.
- r is the number of ratings.
- μ is the learning rate.
- λ is the regularization parameter.
- `RatingFile` is the path of the file that contains all the rating. The ratings are in the following format:

  `RatingID::UserID::ItemID::Rating`

  - `RatingID`s are integers between 1 and r. It uniquely identifies each rating.
  - `UserID`s are integers between 1 and n.
  - `ItemID`s are integers between 1 and m.
  - `Rating`s are integers between 1 and 5.
- `IterFile` is the path of the file that contains order of the iterative optimization steps used in the stochastic gradient descent algorithm. The file contains multiple lines, where each line contains only one integer denoting a valid `RatingID`. This gives the specific rating that the algorithm needs to pick and optimize in the corresponding step.

[1]Note: this is a transposed matrix of the R used in the lecture slides.

- `RandomFile` is the path of the file that contains a sequence of random numbers. You will need to access these numbers in a particular order (specified later) in the initialization phase.

2.1. **Output the Matrix.** You are required to output the factor-item matrix Q to the standard output. The matrix should be in the following format space):

```
a_1_1 a_1_2 ... a_1_m
a_2_1 a_2_2 ... a_2_m
...
a_f_1 a_f_2 ... a_f_m
```

where

- `a_i_j` refers to the element in i-th row and j-th column. You should print it out as a floating point number without any truncation (i.e., using `%f` format).
- numbers in the same line are seperated by one space character (i.e., `0x20`).
- lines are separated by a single `\n`.

2.2. **Use of the `IterFile`.** The standard SGD algorithm consists of multiple iterations; in each iteration (also called a "step"), it picks up a random rating in R to optimize. It stops when convergence is achieved.

In our project, we adapt the algorithm in two ways.

(1) Firstly, you do not need to perform the random sampling by yourself, but instead, just use the `RatingID`s in the `IterFile`.

Please **strictly follow the order** in the `IterFile`, otherwise you may get different result than our reference implementation, and this will result in a low or even zero mark in our automated test for your submission.

(2) Let the `IterFile` contain t `RatingID`s. Then the algorithm shall stop after performing t iterations, i.e., when your implementation consumes all the `RatingID`s in the `IterFile`.

2.3. **Initialization of the P and Q Matrices.** We follow the textbook (Section 9.4.5) and initialize every cell of the **P** and **Q** Matrices as $\sqrt{\frac{a}{f}} + b$, where a is the average rating (over all ratings given in the `RatingFile`), f is the number of factors, and b is a random number drawn from some distribution. In this project, you obtain b sequentially from the given `RandomFile` for each cells in **P** and then for each cells in **Q**, both in **row-wise order**.

2.4. **Fomula used in the SGD Algorithm.** We use formulae on Slide 34 of "Recommendation System (b)", which are reproduced below.[2]

$$\begin{aligned}\epsilon_{xi} &= 2(r_{xi} - \langle \mathbf{q}_i, \mathbf{p}_x \rangle) \\ \mathbf{q}_i &= \mathbf{q}_i + \mu(\epsilon_{xi}\mathbf{p}_x - 2\lambda\mathbf{q}_i) \\ \mathbf{p}_x &= \mathbf{p}_x + \mu(\epsilon_{xi}\mathbf{q}_i - 2\lambda\mathbf{p}_x)\end{aligned}$$

[2]Do not use the ones in the original Stanford slides, which contain some minor errors.

Please note that we use the atomic update rule for the above calculation (see the additional slide in the lecture notes for details).

3. Task II: Random Projection Trees

After generating the factor-item matrix, your next task is to build a set of random projection trees (aka., RP-Trees) for the items, and then answer a set of top-k Nearest Neighbor queries. Since each item is represented as a f-dimensional vector in the matrix, we use the term "items" and "points" interchangeably hereafter.

You are required to write a Java class named **RPTrees**, and the following **seven** parameters will be given from the command line:

- n is the number of points that need to be indexed (i.e., m in Task I).
- d is the dimensionality of the points (i.e., f in Task I).
- `NumOfTrees` is the number of RP-Trees that you need to build.
- `MaxSizeOfLeaf` is the maximum number of points that you can put in a leaf node of a RP-Tree.
- `DataFile` is the path of the file that contains the points that need to be indexed. Its format is exactly the same as the output of Task I.
- `RandomGaussianFile` is the path of the file that contains a sequence of floating point numbers which are randomly sampled from the *Standard Gaussian distribution*. You obtain random direction vectors based on this file when building the RP-Trees. Each line in the file contains one floating point number. We guarantee that there are sufficient numbers for the program to use. It is, however, possible that your implementation does not consume all the numbers in the file.
- `QueryFile` is the path of the file that contains the queries. Each line of `QueryFile` contains one integer which refers to an `ItemID`, indicating that we would like to find the top-3 nearest neighbors (i.e., *other* items) of the current `ItemID`.

  For example, the following file contains two queries

```
12
147
```

3.1. Answer the Query.

For each query, you will need to output its **top-3** nearest neighbors by their identifiers (i.e., `ItemID`s). Note that we do not count the query itself as one of its top-3 NNs.

Assume that there are q queries, then your output should contains q lines, where each line contains three integers, indicating the `ItemID`s of the first NN, second NN, and third NN, respectively. These three `ItemID`s shall be seperated by a space. For example:

```
7 113 46
76 52 7
```

indicates that Item 7, 113, and 46 are the top-3 NNs of Item 12, respectively.

3.2. How to use the `RandomGaussianFile`.

According to the index construction algorithm, you need to generate a random direction vector U at each internal node of a RP-tree.

In this project, you need to source the standard Gaussian random values from the `RandomGaussianFile`. For the i-th generated vector, you will have to use the $((i-1)d+1)$-th to $((i-1)d+d)$-th floating point numbers in the file.

To guarantee the correct use of these random numbers, you **must**

(1) generate RP-Trees one by one;
(2) for each RP-Tree, you must build it in the **pre-order** (i.e., following the order of the algorithm in the lecture slides (or the paper)).

Please **strictly follow these orders**, otherwise you may get different result than our reference implementation, and this will result in a low or even zero mark in our automated test for your submission.

3.3. **Split Rule of RP-Tree.** As mentioned in the index construction algorithm in the slides, we use a specific definition of `median`, which takes a set of n sorted numbers and returns a value.

- if n is odd, return the $(\lfloor n/2 \rfloor + 1)$-th number.
- if n is even, return the average of the $(n/2)$-th and $(n/2+1)$-th number.

All the points whose projection is no larger than the median value (defined above) shall go to the left child, and the rest shall go to the right child.

4. Marking

Your proj1 marks consist of the following parts:

- Auto-mark your implementation on some test datasets and parameters (90%)
- The coding style (10%).

4.1. **Automarking Your Implementation.** Auto-marking will be used to mark your implementation on a linux machine. Specifically, we will

(1) Untar the `proj1.tar` and run `javac *.java` to compile your `.java` files.
(2) Run your programs (`SGD` and `RPTrees`) with some valid parameters and record your outputs and compare them with sample outputs.
(3) Repeat Step b) several times, each with a different set of parameters.

Therefore, you need to make sure:

- Your java program can be correctly compiled. Make sure you do **not** put some of your java source files at a sub-directory. Since most CSE servers (e.g., `wagner.cse.unsw.edu.au`) are running **Java 1.7**, so make sure your program do not use new feature that can only be compiled on higher version of Java (e.g., Java 8).
- Your java program can be run by `java SGD ...` or `java RPTrees ...` from the command line. Make sure you do **not** use `package` for your code.
- Your java program output results in a format exactly identical to that described in this specification. Make sure your output does **not** contain other messages (e.g., debugging messages).

Be warned that you will recieve **0 mark** if your program cannot be compiled or run correctly in our testing environment (e.g., program exit abnormally due to some `Exception`). If you develop your program under another OS (e.g., Windows), it is **strongly** recommended that you test your program on any of the linux machines at CSE.

We will implement some sanity check for your submission in the `give` system. So watch out for error messages when you submit.

4.2. **Coding Style.** We are mainly after readability and assume you know the very basics such as proper indentation. Specifically, use sensible variable and function names, and add comments especially at key places.

5. Submission

Please `tar` your source codes (i.e., `*.java` files) into a file named `proj1.tar`.[3] All your source codes **must** be in the same directory (i.e., no sub-directory allowed).

Your submission should contain at least two files named `SGD.java` and `RPTrees.java`. You are free to add any other Java classes.

You can submit your file by

```
give cs9318 proj1 proj1.tar
```

Late Panelty. -10% for each of the first three days, and -20% for each of the following days.

Plagiarism. Make sure you read "8. Academic honesty and plagiarism" in `http://www.cse.unsw.edu.au/~cs9318/15s1/intro.html`

[3]The command line is `tar cvf proj1.tar *.java`.