COMP47580
Recommender Systems & Collective Intelligence

Recommender Systems Assignment – Part 1

1. Introduction

This part of the assignment involves the implementation of non-personalised recommender algorithms and running experiments. A Java framework is provided for you to extend. To begin, download this framework (an Eclipse project) from Moodle and import it into Eclipse.

1.1 Submission

- See Moodle for the submission deadline.
- See section **4. Submission Instructions** of this document for details on what needs to be submitted.
- Late submissions – where coursework is submitted late, without extenuating circumstances/prior late submission approval, the following penalties will apply:
    - Coursework submitted at any time up to and including 5 working days after the due date will have the grade awarded reduced by one grade point (for example, from B- to C+).
    - Coursework submitted more than 5 working days but up to and including 10 working days after the due date will have the grade reduced by two grade points (for example, from B- to C).
    - Coursework received more than 10 working days after the due date will not be accepted.
- **Important** – this is <u>not</u> a team/group assignment – each student must submit her/his own work. Please ask if you have any questions about the UCD plagiarism policy.

2. Implementation

In a non-personalised recommendation setting, the objective is to generate a list of recommendations for a given *target item*. For each target item, the key step in the algorithm is to calculate a *rank score* for all other items; recommendations for the target item are then ranked in descending order according to this score. Therefore, for a given target item, a different list of recommendations will be generated depending on the approach used to rank items.

In this assignment, the task is to consider a number of different raking approaches and to evaluate the relative performance of each approach based on the evaluation criteria as described in section **3. Running Experiments**.

An implementation of one ranking approach based on movie genres – **GenreRanker** – is already provided in the framework (class `GenreRanker` in package `alg.np.ranker`). In this approach, for a given movie *X*, the rank score for movie *Y* is given by the *overlap coefficient* calculated over the set of genres associated with each movie. (For more information on the overlap coefficient, see the lecture notes on Content-based Recommendation.)

The following three ranking approaches should be implemented. Separate **classes** for each of these approaches (`GenomeRanker`, `RatingRanker`, and `AssociationRanker`) have been created in package `alg.np.ranker`:

- Each class implements the `Ranker` interface (in package `alg.np.ranker`) and should override the following method to calculate the rank score for movie *Y* for a given movie *X* (where *X* and *Y* are movie ids):

  ```
  public double getRankScore(final Integer X, final Integer Y,
                             final DatasetReader reader)
  ```

- Currently, this method in each class simply returns a rank score of zero. The task then is to edit this method in each class to implement the ranking approaches described below.
- **Notes:**
  - Other than implementing the method `getRankScore` in classes `GenomeRanker`, `RatingRanker`, and `AssociationRanker`, **do not change** any of the other code in these classes. *If you do, your code may not run or may output incorrect results. If so, you will not be awarded any marks for your code.*
  - Do **not** round the `double` value returned by the method `getRankScore` in any of the classes.

1. **GenomeRanker.** For a given movie *X*, the rank score for movie *Y* is given by *weighted Jaccard* calculated over genome tag scores. For example, assume movies *X* and *Y* have the following scores for tags $t_1, \ldots, t_5$:

	t_1	t_2	t_3	t_4	t_5
X	0.2	0.3	0.8	0.7	0.9
Y	0.1	0.4	0.8	0.5	0.6

In this case, weighted Jaccard is given by:

$$J_w(X,Y) = \frac{\sum_i \min\Big(X(t_i), Y(t_i)\Big)}{\sum_i \max\Big(X(t_i), Y(t_i)\Big)} = \frac{0.1+0.3+0.8+0.5+0.6}{0.2+0.4+0.8+0.7+0.9} = \frac{2.3}{3.0} = 0.77$$

Here, movie *Y* will receive a high rank score if both movies have similar tag scores. Note that, in the dataset provided, each movie has scores for 1,128 tags and weighted Jaccard should be calculated over all 1,128 tags.

2. **RatingRanker.** For a given movie *X*, the rank score for movie *Y* is given by the *cosine* of the angle between the movie rating vectors. Here, movie *Y* will receive a high rank score if both movies have similar ratings patterns. For this ranking approach only, if a user has not rated a movie, assume the rating is zero; therefore, if none of the users have rated both movies, the rank score will be zero. Note that if **division by zero** occurs when calculating cosine, the result of the division should be set to zero. (For more information on calculating cosine, see the lecture notes on Content-based Recommendation.)

3. **AssociationRanker.** For a given movie *X*, the rank score for movie *Y* is based on the *product association* approach – i.e. the rank score for movie *Y* is given by the *increase in liking Y if X is liked*. (For more information, see the lecture notes on Non-personalised Recommendation).

   This approach can be applied to the movie domain as follows. For simplicity, assume *X* and *Y* are individual movies and a movie is considered liked by a user if the assigned rating is ≥ *t* (**where *t* = 4.0**). For a given movie *X*, the rank score for movie *Y* is calculated according to the definitions given below.

   - **supp(*X*)** is given by the number of users who have assigned a rating ≥ *t* to *X*.

     supp(*X* and *Y*) is given by the number of users who have assigned a rating ≥ *t* to both *X* and *Y*.

     conf(*X* => *Y*) is given by supp(*X* and *Y*) / supp(*X*)

- **supp(!*X*)** is given by the number of users who have assigned a rating < *t* to *X* (do not consider users who have not rated movie *X*).

  supp(!*X* and *Y*) is given by the number of users who have assigned a rating < *t* to *X* (do not consider users who have not rated movie *X*) and a rating ≥ *t* to *Y*.

  conf(!*X* => *Y*) is given by supp(!*X* and *Y*) / supp(!*X*)

- For movie X, the **rank score** for movie *Y* is then given by:
  conf(*X* => *Y*) / conf(!*X* => *Y*)

It is important to check for **division by zero** in the above calculations. In all cases, if the denominator is zero, the result of the division should be set to zero; otherwise the output of the method will be incorrect.

3. Running Experiments

Run the following experiments to evaluate the performance of the different ranking approaches.

Additional implementation work is **not** required to run these experiments.

3.1 Experiment 1

Once the ranking approaches have been implemented, run the class `ExecuteNP` (in package `alg.np`). This class will output the top-3 recommended movies with the highest rank scores for a sample set of 5 target movies.

Currently, this class is configured to run using the `GenreRanker` (in package `alg.np.ranker`) ranking approach. To run class `ExecuteNP` using a different ranking approach (e.g. `GenomeRanker`), change line #31 in class `ExecuteNP` from:

```
Ranker ranker = new GenreRanker();
```

to:

```
Ranker ranker = new GenomeRanker();
```

Run the class using each of the four ranking approaches and note the difference between the top-3 recommendations that are made for each of the target movies. This will provide you with a sense of the performance of each of the ranking approaches.

3.2 Experiment 2

Next, run the class `ExecuteNP_Expt` (in package `alg.np`). This class will output the results of an evaluation of each of the four ranking approaches (`GenreRanker`, `GenomeRanker`, `RatingRanker`, and `AssociationRanker`) using the criteria described below.

1. **Recommendation relevance:**
    - The relevance of a recommended movie is given by the mean of the ratings the movie received in the training set.
    - For a given target movie, the relevance of the top-k recommendations made is calculated by taking the average of the mean rating over each recommended movie.
    - The statistic reported is the average over all target movies.

2. **Coverage** is given by the percentage of target movies for which at least one recommendation can be made.

3. **Recommendation coverage** is given by the percentage of movies in the dataset which appear at least once in the top-k recommendations made over all target movies.

4. **Item-space coverage:**

   - For a given target movie, the percentage of movies in the system which are capable of being recommended is calculated:
     - For rank scores based on genres, genome tag scores, and ratings, those movies which have a non-zero rank score with respect to the target movie are considered capable of being recommended.
     - For rank scores based on the product association approach, those movies which have a rank score > 1 with respect to the target movie are considered capable of being recommended.
   - The statistic reported is the average percentage over all target movies.

5. **Recommendation popularity:**

   - The popularity of a recommended movie is given by the percentage of users in the system which have rated the movie.
   - For a given target movie, the popularity of the top-k recommendations made is calculated by taking the average of the popularity over each recommended movie.
   - The statistic reported is the average over all target movies.

Once this experiment has been run, plot the following two **graphs**:

- In the first graph, plot recommendation relevance versus ranking approach.
- The second graph should plot coverage, recommendation coverage, item-space coverage, and recommendation popularity versus ranking approach all on one graph (for example, use a clustered column graph in MS Excel).

4. Submission Instructions

1. Using the **Submit Classes** link available on Moodle, submit the three classes you implemented (`GenomeRanker`, `RatingRanker`, and `AssociationRanker`). Tests of your classes will be performed automatically and your **marks** will be shown. See Moodle for further instructions.

2. Using the **Submit Results** link available on Moodle, submit the **output** of class `ExecuteNP_Expt` (in package `alg.np`). A test of your output will be performed automatically and your **marks** will be shown. See Moodle for further instructions.

3. Finally, using the **Submit Eclipse Project Export** link available on Moodle, submit an **export of your entire Eclipse project**:

   - Create a **zip file** of your **exported Eclipse project** using the following filename: `NP_12345678_Code.zip` (where `12345678` is your student number).
   - In the Eclipse project export, include:
     - All code provided in the framework and the code you implemented yourself.
     - Your **graphs** – save these in folder `graphs` in the Eclipse project.
   - When submitting your code, **please do not edit any of the classes provided in the framework** (except, of course, for the three classes `GenomeRanker`, `RatingRanker`, and `AssociationRanker`)**:**
     - When grading your code, classes `ExecuteNP` and `ExecuteNP_Expt` will be run (these classes are provided in package `alg.np`).
     - *If you do not follow these instructions, your code may not run or may output incorrect results. If so, you may not be awarded any marks for your code.*
     - Of course please feel free to experiment with the framework – but do this using a local copy of the framework, and remember to submit an export of the correct version (i.e. unedited except for the classes `GenomeRanker`, `RatingRanker`, and `AssociationRanker`) of the framework to Moodle.