A Simple Rating Prediction System

Joe Carver (jc30g12)

January 6, 2016

1 Introduction

This report details the implementation and testing of a simple Java-based prediction system, designed with the aim of estimating a set of ratings that a set of users might give a set of previously un-rated tracks. The data provided was divided into two `.csv` files (training, testing) which contain around 17m and 444k entries respectively. Together, the sets contain roughly 200,000 users and 30,000 songs. The investigation found that a prediction function based on simple cosine-similarity measures was able to predict unknown user-track ratings, with a MAE of marginally over 1 when compared to the actual (hidden) ratings.

Section 2 presents an overview of the design of the system. Section 3 outlines the technologies used in development and the difficulties that were encountered in using them, as well as some justification for decisions made, and section 4 explains and discusses the techniques used to evaluate the algorithms. Section 5 reflects upon the actual results obtained, and potential ways in which to improve the system.

2 Design Overview

Java was chosen as the language in which to develop the prediction system, largely due its familiarity, and also as there exists a very straightforward, built-in SQL API library, Java Database Connectivity (JDBC) [2]. This allows us to view, modify and retrieve data from the database directly in Java code. The ability to do this is absolutely necessary, due to the huge amount of data that would have been impossible for the java run-time to keep track of and process itself. Persistent storage of data in these SQLite tables also facilitated the separation of key functionality into two distinct runnable programs.

2.1 SQLite & JDBC

Communication between the Java run-time and the persistent SQLite database was key in dealing with memory-related issues. The database was initialised to hold all of the training and testing data in separate tables; achievable through some simple SQLite commands in the following format:

```
CREATE TABLE trainingData (userID INT, trackID INT, rating INT)
.mode csv
.import comp3208-trainingset.csv trainingData
```

The above process was repeated for the test data set, for which the resulting table did not contain any values in its `rating` column.

The JDBC API allows for very straightforward access, insertion and manipulation of values and tables in an SQLite database from within the Java application itself. For example, connecting to the database initially on each execution can be achieved as such:

```
Class.forName("org.sqlite.JDBC");
Connection c = DriverManager.getConnection("jdbc:sqlite:rating_db.db");
```

where `rating_data.db` is the SQLite file containing all tables needed to work with.

Once the Java runtime has a reference to the SQLite database containing the data, we can use `java.sql.Statement` objects to communicate directly with the database. Instances of this class are created by the corresponding connection, `c` and contain themselves a function `executeQuery(String Sql)`, which, as expected, executes the specified SQL query given in string form, and returns an object of type `java.sql.ResultSet`. If the query itself was, for example, of type `SELECT`, then the obtained values can be accessed through use of a function such as `ResultSet.getInt()` - columns can be referenced by either their index or name, and different `get` methods exist depending on the type of value being retrieved.

Inserting values into the a table from Java follows an equivalent pattern, and the library provides reusable `PreparedStatement` objects with a function `setInt(int i)`, enabling the same block of code to be called repeatedly to gradually update table rows. When dealing with huge datasets like this, it is important to turn off the *Autocommit* functionality of JDBC, the reasons for which are explained in section 2.2.

2.2 Java System Design

The system was split into two independent classes - `CalculateSimilarities` and `PredictRatings`, both containing their own `Main` method and no reference to the other class. The specific functionality of these two classes is described in detail in section 3, but the general purpose of each is as follows:

Both classes handle the raw data in an identical fashion - reading the *user-track-rating* entries from database before restructuring and storing into two separate data structures. These are both nested structures that contain identical data, indexed inversely to one another:

```
HashMap<Integer, HashMap<Integer, Integer>> allRatings_byTrack;
HashMap<Integer, HashMap<Integer, Integer>> allRatings_byUser;
```

These initial operations require time (roughly 30 seconds per structure) to generate and obviously memory to store, but these costs are offset by the reduction in retrieval time-complexity that is inherent when using correctly indexed data structures. These structures in fact proved to be entirely requisite for a system that required so many calculations, as explained further in section 4.2.

CalculateSimilarities compares every item (track) in the training dataset with every other item, computing their similarity (as in section 3.1), and storing this value in an SQLite database if it is above a specified threshold (0, in this case). Committing entries to the database is an irreversible and relatively expensive operation, and as such was performed only once per 750,000 similarity values. The JDBC library automatically handles temporary storage of these rows in the mid-commit periods. The periodicity with which values are committed to the database is a vital factor in determining the system's efficiency; too frequent and execution time will rapidly increase, too infrequent and we will encounter memory issues. One final measure was taken to optimise the memory and speed requirements at this stage by closing and re-opening the JDBC connection periodically (after the full analysis of 250 items).

PredictRatings iterates over every entry in the test-set (i.e. *user-track* pairs) and applies to each entry a prediction function (section 3.2) This predicted value is then output along with its corresponding user and track ID number to a `.csv` file ready for submission or evaluation.

3 Implementation Details

The nature of the data provided meant that there was a complete absence of knowledge or content-based information on which to base the recommendations, so the inevitable decision made was to use a collaborative-filtering recommender, which exploits the relationships and patterns between users or items that exhibit similar distributions of ratings.

3.1 Computing Similarity

At the heart of any collaborative-filtering recommender system there is a function that can accurately estimate the similarity between two equivalently-structured entities. The entities to be compared are most often users of the system, or the items they are viewing, rating or purchasing. A similarity matrix is the data structure most commonly used as a record of how similar pairs of entities are. In this case, the similarity matrix was stored persistently in an SQLite table that took the following schema:

```
CREATE TABLE itemSimilarities(Item1 INT, Item2 INT, Similarity FLOAT)
```

It is vital that the similarity value be represented as a floating point number, as only in very specific cases does the function (discussed below) return a rounded integer for any two items. The actual similarity function implemented is described in the following section 3.1.1.

Once the similarity matrix has been fully computed and stored in the SQLite table, a final optimisation-oriented operation is to index the table by track ID,

which takes a few minutes to compute initially but vastly improves performance when retrieving a whole set of *item-item* similarities.

3.1.1 Similarity Function - Adjusted Cosine Similarity

Item-Based collaborative filtering formed the basis for the vast majority of the prediction function. Adjusted Cosine Similarity measures the size of the angle between two vectors of item ratings and is known to produce results that perform competitively with user-based collaborative filtering methods, whilst retaining insensitivity to growing collections of users, which was very relevant in this case. The measure is referred to as 'adjusted' as for each co-rated pair of items, the user average rating is subtracted.
Adapted roughly from [7], the adjusted cosine similarity between two items `a, b` is calculated as follows:

Algorithm 1 Calculating Similarity

$mutuals \leftarrow$ all users who rated a, b
$mutualSize \leftarrow$ total size of $mutuals$
if $mutualSize < 1$ **then return** 0

$sumRatingsA \leftarrow \sum_{u \epsilon mutuals} rating_{u,a}$
$sumRatingsB \leftarrow \sum_{u \epsilon mutuals} rating_{u,b}$
$sumRatingsSqA \leftarrow \sum_{u \epsilon mutuals} rating_{u,a}^2$
$sumRatingsSqA \leftarrow \sum_{u \epsilon mutuals} rating_{u,b}^2$
$sumRatingsAB \leftarrow \sum_{u \epsilon mutuals} rating_{u,a} * rating_{u,b}$

where $rating_{u,a}$ *is user* `U`*'s rating for track* `A`

By substituting the values obtained from running the above algorithms into equation, an estimation of the similarity of 2 items (in this case tracks) can be obtained.

$$sim(a,b) = \frac{sumRatingsAB - \frac{(sumRatingsA * sumRatingsB)}{mutualSize}}{\sqrt{(sumRatingsSqA - \frac{sumRatingsA}{mutualSize}) * (sumRatingsSqB - \frac{sumRatingsB}{mutualSize})}} \quad (1)$$

As the above algorithm and equation suggest, computing the similarity of two arbitrary length vectors is no trivial task, and when dealing with datasets as large as this, optimisation becomes a key consideration. A satisfactory time-complexity reduction was achieved through use of the indexed structures described in section 2.2, whilst a largely static-based system design prevented significant memory leak.

3.2 Predicting Ratings

Given a similarity matrix containing values for all pairs of items, the task of predicting unknown ratings from these similarities is comparatively simple, both computationally and intuitively. To predict user `U`'s rating for track `A`, the following algorithm is executed:

Algorithm 2 Predicting Rating

$similarities \leftarrow$ similarities of track **A** to every other track
$ratedA \leftarrow$ all users that also rated track **A**

$simSum \leftarrow \sum_{u \epsilon ratedA} similarity_u$
$simRateSum \leftarrow \sum_{u \epsilon ratedA} similarity_u * rating_{u,a}$

if $simRateSum == 0$ **then** prediction = $alternativePred$

else $prediction = \frac{simSum}{simRateSum}$

Predicting the rating of a track consists of first retrieving the similarity values from the SQLite database for that track, identifying users who have also provided a rating, and then calculating the sums shown above. The subsequent step in execution depends on the value of *simRateSum*. If the value is 0, we need to use an alternative prediction function, discussed below (section 3.2.1). Usually however, we can calculate a predicted rating simply with eq. (2):

$$rating = \frac{simSum}{simRateSum} \tag{2}$$

Due to the user-adjusted characteristic of the similarity function, it is presumed and proved through testing that no performance improvement would be derived by considering an average track or user rating at this stage.

3.2.1 Alternative Prediction Function

Although an item-based collaborative filtering recommendation system is known to perform comparably with a user-centric system [6], there was still a need in the system for a user-based collaborative approach. For several *user-track* pairs that were to have a rating predicted, there existed no other users with a suitably high similarity score that had also rated the track in question.

This is effectively the problem of a *cold start* for the track, as there are no existing useful ratings for which are system to base its calculation on. The solution when dealing with these sparsely rated tracks was to shift to a user-based collaborative filtering system. To predict User **U** rating of Track **A**:

Algorithm 3 Alternative Similarity

$similarity \leftarrow$ pearson similarity function
$ratedA \leftarrow$ all users that also rated track **A**

$simSum \leftarrow \sum_{v \in ratedA} similarity(u, v)$
$simRateSum \leftarrow \sum_{v \in ratedX} similarity(u, v) * rating_{v,a}$

if $simSum == 0$ **then** prediction = 0

else $prediction = \frac{simSum}{simRateSum}$

The pre-computed similarity matrix is no use in this case due to the sparsity problem. As a result, it is necessary to compute the similarity between two users, and use these relationships in conjunction with the known ratings to generate a rating estimate for the given *user-track* combination.

This kind of all-in-one go technique is far too time consuming to employ for every entry in the test set; this limitation is indeed the main motivation behind the separation of classes and the similarity matrix that bridges them. However, the rarity with which this process must be applied, aided by the pair of inverted indexed structures, means that it can be used to fill in most otherwise-incalculable predictions without seriously affecting overall execution time.

There is a very small amount (129) of *user-track* combinations which suffer from both the user and item cold start problem, and for which no predicted rating has been supplied.

4 Evaluation

There are several options available when carrying out an evaluation of a recommender system. In the absence of online, social, more heuristically-centered evaluation techniques that aim to capture an essence of the actual impression on the user, popular mathematical analytical methods become the default technique. Additionally, the efficiency of the system in terms of time and memory usage can be evaluated in a more intuitive manner.

A third Java class with `Main` method was created and initialised in a similar manner for evaluation purposes.

4.1 Mathematical Accuracy

A very rudimentary approach would be to compare the predicted rating for an item to the average rating for that item among the known values. This technique can be vastly improved upon however, with the most eminent mathematical methods today being Mean-Absolute-Error (section 4.1.2), and Root-Mean-Square-Error (section 4.1.3). [8]

4.1.1 Test Set

In order to carry out a mathematically-verified evaluation of a system, it is first necessary to obtain a set of *user-item* pairs for which we know the true value of the rating, before concealing this rating, and computing an estimate of it. The estimate can then be compared to the true value, and analysis of these differences across a large dataset should give an indication of the accuracy with which the system is performing. SQLite makes it very straightforward to select a pseudo-random sample of the 17million strong dataset for this purpose:

```
SELECT * from trainingData WHERE rowID % 100 = 0
```

By simply retrieving columns 1 and 2 from the `ResultSet` obtained, we can build a list of roughly 170,000 entries and predict a rating for each *user-track* pair. By subsequently retrieving column 3 from the original data, we can build an ordered list of the real ratings for the same *user-track* pairs.

4.1.2 Mean Absolute Error

The primary evaluation metric used was MAE, which is discussed in [11]. It is a highly intuitive measure that simply sums the difference between the actual and predicted rating assigned to each *user-track* pair to give an Absolute Error, before dividing this value by the total entry count, resulting in Mean Absolute Error. The equation for calculating this measure is shown below, where `N` is the set of all ratings to be compared, $\mathtt{p_i}$ is the predicted rating, and $\mathtt{d_i}$ is the real (hidden) rating.

$$MAE = \frac{1}{N} \sum_{i \epsilon test}^{N} |p_i - d_i| \tag{3}$$

When evaluated in the manner outlined above, the system detailed in this report produced ratings with a MAE of 0.98733.

4.1.3 Root Mean Square Error

RMSE differs subtly from MAE in that the differences of each *predicted-actual* rating are squared before being summed, and rooted after the mean is taken. The equation for RMSE is shown below, with equivalent labels as that for MAE.

$$RMSE = \sqrt{\frac{1}{N} \sum_{i \epsilon test}^{N} (p_i - d_i)^2} \tag{4}$$

This means more emphasis is placed on larger deviation, so it is essentially a harsher evaluation metric than MAE. The RMSE for this system was 1.24657, and we can note that a lower error of either kind indicated a greater degree of accuracy in prediction.

4.1.4 Average Rating Comparison

Finally, each predicted *user-item* rating was compared to the average rating for that item by all users, summing the absolute difference between these two values. Dividing this sum by the total number of predicted ratings in the test

evaluation set gave an average difference of 0.8679. Although this is a lower value than the MAE or RMSE, we cannot use it to draw any conclusions about the accuracy of the system as the metric is inferior to both others, and almost entirely meaningless for data sets such as this.

4.2 Thoughts on Computational Costs

The ability to fully evaluate the system with a variety of neighbourhood sizes and algorithm variations was hindered by the optimisation-related difficulties that were encountered and are discussed in section 5.3.

For example, calculating the entire set of *item-item* similarities proved to be the most computationally demanding task, due predominately to the amount of operations involved in calculating just one pair of similarities. These operations involved obtaining a set of mutually-rating users for the two items, using the `Set1.RetainAll(Set2)` method, before iterating over the resulting set and computing the sums listed in algorithm 1. Retrieving the rating values for each entry in this set of mutual items is crucially hastened by use of the data structures described in section 2.2.

Clearly, the length of time required to calculate one entry in the set of all *item-item* similarities is overwhelmingly dependent on the amount of users that have rated both items in question, and in this sense there is little that could be done to improve the time complexity.

5 Results & Reflection

5.1 System Performance

The system achieved satisfactory performance when applied to the test set, producing ratings with a MAE of 1.04797. Although this value is slightly higher than what was observed in the evaluation process, some variance is expected due to the random nature of the figures used. The value given above actually excludes missing or 0-value ratings, the presence of which indicates incomplete performance by the system. The actual MAE value, including these missing values was 1.12579. To this end, it is evident that by including even the most rudimentary prediction function for missing ratings, the overall performance could have been improved.

5.2 Potential Improvements

In a general sense, it is obvious that recommendations enhanced by content analysis can offer superior results [4], but this is clearly out of scope when dealing with this dataset. Research into pure collaborative-filtering systems that utilise aggregated external data [10] show a similarly impressive performance increase and is slightly more relevant to the problem in question, but without an overarching statistical model from which to draw theoretical conclusions, this technique is also slightly out of reach.

Any enhancements to the results produced by this recommender system on this data must therefore be solely concerned with the performance of the algorithms involved.

Neighbourhood selection is always a key consideration when designing recommendation systems, as it is vital not to consider any vectors with rating patterns wildly different to that of the vector in question. The system described in this report uses only a very rudimentary method of selecting neighbours which is simply to ignore any that exhibit positive correlation (by the adjusted cosine similarity) and include every other item in the neighbourhood for predictions. This is an approach that can clearly be improved on, perhaps as in [5], which attempts to ameliorate the effects of issues like sparsity through a technique reliant on Pareto dominance based k-neighbour selection.

Bobadilla's work in 2011 [1] took a more radical approach and proposed a whole new approach to similarity measures. Their idea was that a genetic algorithm can be used to select, from a family of similarity measures, the most optimal measure - i.e. that with the lowest MAE - to compare pairs of users with sets of ratings vectors. The members of each similarity family comprise functions based on vectors calculated as a ratio involving the number of items rated by pairs of users and the absolute differences between these pairs of ratings.

5.3 Reflection

The main obstacle in developing this recommendation system presented itself in the form of an optimisation problem. The first and most time consuming challenge was to design the system such that similarities and predicted ratings could be calculated in a time frame that was actually feasible when considering the deadlines (i.e. a system that could complete in hours, rather than days). This was achieved making the decisions described in section 2 and section 3, but further improvements that would have facilitated more rigorous and definitive testing were not possible to include.

One key improvement that would have vastly reduced computational costs involves the structure of the similarity matrix. For every *item-item* pair, the matrix contained two identical entries, making it, in other words, a symmetric matrix. When each value in the matrix represents a similarity between two entities, it becomes entirely unnecessary to store this value twice. The amount of duplicate entries in a symmetric matrix rises exponentially with its size, so avoiding these is often a key concern. The fairly rudimentary nature of the code used to construct and access the SQL table corresponding to the matrix meant that removal of these duplicates was not possible, adding a significant memory and time cost.

Although it did not have any effect on the time or memory requirements of initially computing the matrix, adding an index to the SQL table post-construction (as described in section 3.1) did offer some considerable optimisation. It facilitated very quick access to a complete set of similarity values for an item, which vastly sped up the prediction process of section 3.2.

Finally, and perhaps crucially, all evaluation measures used focussed on the accuracy of the predicted ratings and not on the accuracy of the perceived similarities between two entities. This meant that any evaluation-inspired changes or tweaks to the system occurred during the prediction stage of the process (section 3.2), and hence were all based on the original similarity values as computed in section 3.1. Considering the centrality of the similarity function to any recommender system, this is clearly not ideal, and a framework allowing for robust evaluation of similarity functions, [3; 1], or even a more in-depth investigation into possible measures on offer as in [9] may have led to an algorithm able to produce values more indicative of the true similarity between two items.

Unfortunately, however, the challenges described earlier in this section meant that such an approach was not a viable option.

References

[1] Bobadilla, J., Ortega, F., Hernando, A., and Alcalá, J. (2011). Improving collaborative filtering recommender system results and performance using genetic algorithms. *Knowledge-Based Systems*, 24(8):1310–1316.

[2] Das, T., Iyer, V., Perry, E. H., Wright, B., and Pfaeffle, T. (2008). Oracle ® Database. 1(July).

[3] Heuser, C. A., Krieser, F. N. A., and Orengo, V. M. (2007). SimEval - A Tool for Evaluating the Quality of Similarity Functions. *ER '07 Tutorials, Posters, Panels and Industrial Contributions at the 26th International Conference on Conceptual Modeling - Vol. 83*, pages 71–76.

[4] Melville, P., Mooney, R. J., and Nagarajan, R. (2002). Content-boosted collaborative filtering for improved recommendations. *"Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)"*, (July):187–192.

[5] Ortega, F., Sánchez, J. L., Bobadilla, J., and Gutiérrez, A. (2013). Improving collaborative filtering-based recommender systems results using Pareto dominance. *Information Sciences*, 239:50–61.

[6] Papagelis, M. and Plexousakis, D. (2005). Qualitative analysis of user-based and item-based prediction algorithms for recommendation agents. *Engineering Applications of Artificial Intelligence*, 18:781–789.

[7] Segaran, T. (2007). *Programming Collective Intelligence*. O'REILLY, United States of America, first edition.

[8] Shani, G. and Gunawardana, A. (2011). Evaluating recommendation systems. *Recommender systems handbook*, pages 257–298.

[9] Sorensen, S. (2012). Accuracy of Similarity Measures in Recommender Systems Categories and Subject Descriptors.

[10] Umyarov, A. and Tuzhilin, A. (2008). Improving collaborative filtering recommendations using external data. *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 618–627.

[11] Zaier, Z., Godin, R., and Faucher, L. (2008). Evaluating recommender systems. *Proceedings - 4th International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution, Axmedis 2008*, page 27.