COMP30820 Java Programming (Conv)
Case Study

Michael O'Mahony
michael.omahony@ucd.ie

April 14, 2019

1 Introduction

The purpose of this case study is to provide an example of a larger Java application and how object-oriented concepts such as inheritance and polymorphism can be used to facilitate program design and development, leading to more robust and extensible software.

2 Recommender Systems

Recommender systems is an active area of research and many algorithms have been proposed. For this case study, one particular approach to recommendation — a form of *content-based recommendation* — is considered. Given a set of *target cases* (i.e. items in a user's profile; for example, movies seen, songs listened to, or books read etc.), the objective is to recommend a ranked list of k other cases that are most similar to these target cases.

The main considerations in content-based recommendation are (1) how cases are represented, (2) how the similarity between cases is calculated, and (3) recommending cases to the end user.

2.1 Case Representation

In this study, cases are represented by a set of feature-value pairs. In particular, we consider the movie domain in which each case (movie) is represented by the following features: id, title, genres, directors, and actors. See Figure 1 for an example.

ID	5430	10818
Title	Sleepers	Mystic River
Genres	Crime, Drama, Thriller	Crime, Drama, Mystery, Thriller
Directors	Barry Levinson	Clint Eastwood
Actors	Robert De Niro, Kevin Bacon, Brad Pitt, Jason Patric	Sean Penn, Tim Robbins, Kevin Bacon, Emmy Rossum

Figure 1: Feature-value pairs for two movies. Each movie can have one or more genres, directors, and actors.

2.2 Case-level and Feature-level Similarity

The similarity between two cases, T and C, is calculated as the weighted sum of the similarities between corresponding feature values:

$$sim(T, C) = \frac{\sum_{i=1}^{n} w_i \times fsim(T_i, C_i)}{\sum_{i=1}^{n} w_i} , \quad (1)$$

where n is the number of features considered, w_i is the weight of feature i (assigned a value in the interval [0,1]), T_i (resp. C_i) is the value of feature i in case T (resp. C), and $fsim(T_i, C_i)$ is a feature-level similarity function that calculates the similarity between the corresponding feature values of two cases.

Depending on the type (e.g. numeric, categorical) of feature values, different feature-level similarity functions can be defined. For movie cases, set-based similarity functions such as Jaccard index and overlap can be used to calculate the similarity between the genre, director, and actor features. Given two sets A and B, Jaccard index and overlap are defined as:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} , \tag{2}$$

$$overlap(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)} . \tag{3}$$

For example, referring to Figure 1, the set of genres associated with the two movies are: {Crime, Drama, Thriller} and {Crime, Drama, Mystery, Thriller}. By Equations 2 and 3, the Jaccard index and overlap are $\frac{3}{4}$ and 1, respectively. Likewise, for the actors feature, the Jaccard index and overlap are $\frac{1}{7}$ and $\frac{1}{4}$, respectively, while both are 0 for the directors feature.

We can now write down the following case-level similarity functions based on genres (g), directors (d), and actors (a):

$$sim(T, C) = \frac{w_g \times Jaccard(T_g, C_g) + w_d \times Jaccard(T_d, C_d) + w_a \times Jaccard(T_a, C_a)}{w_g + w_d + w_a} , \tag{4}$$

$$sim(T, C) = \frac{w_g \times overlap(T_g, C_g) + w_d \times overlap(T_d, C_d) + w_a \times overlap(T_a, C_a)}{w_g + w_d + w_a} . \tag{5}$$

The weights w_g, w_d, and w_a can be set to control the relative influence of genre, director, and actor feature-level similarity on the overall case similarity. Case-level similarity functions involving different combinations of Jaccard index and overlap feature-level similarity functions can also be defined.

Other case-level similarity functions are also possible; for example, feature-level similarity based on movie titles can be included. One approach to calculate the similarity between movie titles is to consider movie titles as sets of words and calculating the Jaccard index or overlap between the sets.

2.3 Recommending Cases

Once the pairwise similarity between all cases has been calculated, the final step is to recommend a ranked list of k cases to the end user. In particular, assume the end user has a number of target cases in their profile; the objective is to recommend the k most similar cases to these target cases. Here, we consider two approaches, *mean recommendation* and *max recommendation*.

2.3.1 Mean Recommendation

For a given recommendation candidate C, its rank score is given by the sum of its similarity to each of the target cases. Once the rank scores of all candidates (i.e. all movies in the dataset less the target cases) have been calculated, sort candidates by descending rank score and return the top k as recommendations to the end user.

2.3.2 Max Recommendation

For a given recommendation candidate C, its rank score is given by the maximum of its similarity to each of the target cases. Once the rank scores of all candidates (i.e. all movies in the dataset less the target cases) have been calculated, sort candidates by descending rank score and return the top k as recommendations to the end user.

3 Code

The content-based recommender algorithm is implemented in its own Eclipse project. Figure 2 shows the packages and classes in the project. In what follows, a description of the main features of the code is provided.

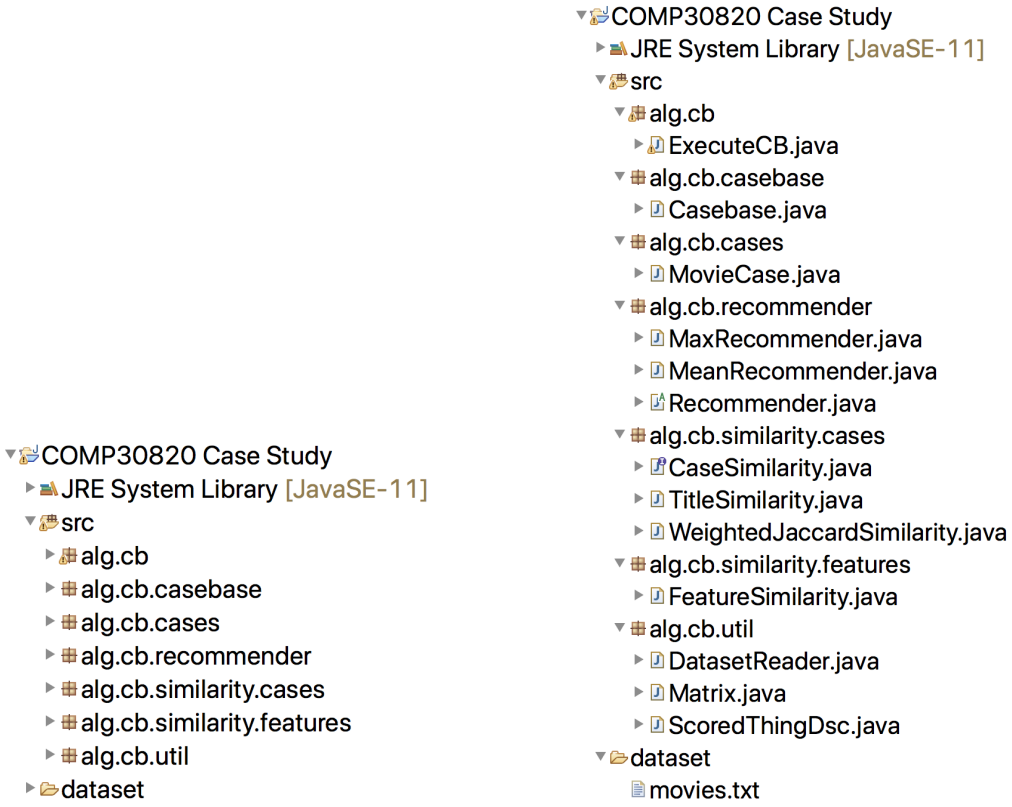

Figure 2: Screenshots of Eclipse project showing (a) packages and (b) expanded packages and classes.

3.1 Package `alg.cb.cases`

Class `MovieCase` is used to represent a case (i.e. a movie). Data fields are included for the following movie features:

```
private int id; // the movie id
private String title; // the movie title
private Set<String> genres; // the movie genres
private Set<String> directors; // the movie directors
private Set<String> actors; // the lead actors
```

3.2 Package `alg.cb.casebase`

Class `Casebase` contains one data field, an `ArrayList` object, which is used to store cases (movies):

```
private List<MovieCase> cb; // stores case objects
```

3.3 Package `alg.cb.similarity.features`

Class `FeatureSimilarity` contains implementations of various feature-level similarity functions (for example, Jaccard index and overlap as described in Section 2.2). The constructor of the class is private, which means the class cannot be instantiated. The feature-level similarity functions are implemented as static methods. These methods are invoked in the case-level similarity function classes.

3.4 Package `alg.cb.similarity.cases`

This package contains case-level similarity functions. Firstly, an interface (named `CaseSimilarity`) is defined which contains a single method to return the similarity between two movie cases, `c1` and `c2`:

```
public double getSimilarity(MovieCase c1, MovieCase c2);
```

Then, we use a separate class for each case-level similarity function we define. For example, the project contains an implementation (class `WeightedJaccardSimilarity`) of the case-level similarity function defined by Equation 4. Note that this class implements the `CaseSimilarity` interface.

Any new case-level similarity class you create should also implement the `CaseSimilarity` interface. Using this approach, it is very easy to modify your code to change to a different case-level similarity function. For example, consider the following code:

```
CaseSimilarity caseSimilarity = new WeightedJaccardSimilarity();
...
double sim = caseSimilarity.getSimilarity(c1, c2);
```

To change to a different case-level similarity function (e.g. `TitleSimilarity`), you just need to change the first line, and the rest of your code does not need to be changed:

```
CaseSimilarity caseSimilarity = new TitleSimilarity();
...
double sim = caseSimilarity.getSimilarity(c1, c2);
```

In the above, classes `WeightedJaccardSimilarity` and `TitleSimilarity` contain a different implementation of method `getSimilarity(MovieCase, MovieCase)`; the use of polymorphism simplifies the code and dynamic binding ensures the appropriate method is invoked at runtime (depending on the *actual type* of the object).

3.5 Package `alg.cb.recommender`

This package contains implementations of different content-based recommender system algorithms. Firstly, an abstract class (named `Recommender`) is defined which contains the following data fields:

```
private Matrix matrix; // a Matrix object to store case similarities
private Casebase cb; // the case base
```

The `Matrix` object (see Section 3.6) stores the pairwise similarities between all movie cases and the `Casebase` object contains all movies in the dataset.

The constructor defined in `Recommender` takes as arguments `CaseSimilarity` and `Casebase` objects, and calculates the pairwise similarity between all movie cases which are stored in the `Matrix` object.

The `Recommender` class also contains an abstract method which returns a ranked list of recommendations based on a specified set of target movies:

```
public abstract List<Integer> getRecommendations(List<Integer> targetIds);
```

Then, we use a separate class for each recommender algorithm we define. For example, the project contains an implementation (class `MeanRecommender`) which implements the algorithm described in Section 2.3.1. Note that this class extends the abstract `Recommender` class.

Any new recommender algorithm class you create should also extend the abstract `Recommender` class. As with the case-level similarity functions, using this approach it is very easy to modify your code to change to a different recommender algorithm. For example, consider the following code:

```
Recommender recommender = new MeanRecommender(caseSimilarity, cb);
...
List<Integer> recIds = recommender.getRecommendations(targetIds);
```

To change to a different recommender algorithm (e.g. `MaxRecommender`), you just need to change the first line, and the rest of your code does not need to be changed:

```
Recommender recommender = new MaxRecommender(caseSimilarity, cb);
...
List<Integer> recIds = recommender.getRecommendations(targetIds);
```

In the above, classes `MeanRecommender` and `MaxRecommender` contain a different implementation of method `getRecommendations(List<Integer>)`. Here again, the use of polymorphism simplifies the code and dynamic binding ensures the appropriate method is invoked at runtime (depending on the *actual type* of the object).

3.6 Package `alg.cb.util`

Class `DatasetReader` reads the movie metadata from a text file (see Section 3.8). It contains one data field, a `Casebase` object, which is used to store cases (movies):

```
private Casebase cb; // stores case objects
```

Class `Matrix` contains one data field, a two-dimensional array of `double` values, which is used to store the pairwise similarities between all movie cases:

```
private double[][] matrix; // the data structure used to store matrix elements
```

Class `ScoredThingDsc` is used as a way to collect a number of objects (each of which is associated with a score) and then to inspect the set sorted in descending order by the scores. It contains the following data fields:

```
private double score; // the score associated with the object
private Object thing; // the object to be sorted
```

This class is used in the recommender algorithm classes to rank recommendations by descending order of similarity to the target cases.

3.7 Package `alg.cb`

Class `ExecuteCB` contains a `main` method to execute the code. To begin, the path and filename of the movie metadata file is specified. Next, a `DatasetReader` object is created to read in the data. Then, the method `getCasebase()` is invoked on the `DatasetReader` object to return a reference to a `Casebase` object which contains all movies in the dataset:

```
// set the path and filename of the movie file and read in the data
String movieFile = "dataset" + File.separator + "movies.txt";
DatasetReader reader = new DatasetReader(movieFile);
Casebase cb = reader.getCasebase();
```

Next, the case similarity metric and recommender are set as follows:

```
// configure the case-based recommendation algorithm - set the case similarity
// and recommender
CaseSimilarity caseSimilarity = new WeightedSimilarity();
Recommender recommender = new MeanRecommender(caseSimilarity, cb);
```

In the above, the constructor of the `MeanRecommender` class takes as arguments `CaseSimilarity` and `Casebase` objects. As described above (Sections 3.4 and 3.5), to change to a different case-level similarity function or recommender algorithm, you just need to change these two lines of code; no other changes need to be made to your code.

Subsequent statements in the main method (1) randomly select a number of target cases (given by variable `numberTargetMovies`), (2) display the target cases, (3) generate and (4) display the top-k (given by variable `numberRecommendedMovies`) recommendations based on the target cases. These statements are wrapped in a loop, which keeps iterating until the user quits.

3.8 Folder `dataset`

File `movies.txt` contains movie metadata. The file format is as follows: each line provides metadata for a single movie and consists of a `<id title genres directors actors>` tuple (tab delimited). For example, the first entry in the file provides metadata for the following movie:

```
id: 4402
title: A Nightmare on Elm Street
genres: Horror, Mystery
directors: Wes Craven
actors: Heather Langenkamp, Johnny Depp, Robert Englund, John Saxon
```

In total, there are 1,073 movies in the file.