Algorithms of the Intelligent Web

Haralambos Marmanis
Dmitry Babenko

SAMPLE CHAPTER

MANNING

Algorithms of the Intelligent Web

by Haralambos Marmanis
and Dmitry Babenko

Chapter 3

brief contents

3 Creating suggestions and recommendations

This chapter covers:

- Finding the distance and similarity between objects
- Understanding recommendation engines based on users, items, and content
- Finding recommendations about friends, articles, and news stories
- Creating recommendations for sites similar to Netflix

In today's world, we're overwhelmed with choices; a plethora of options are available for nearly every aspect of our lives. We need to make choices on a daily basis, from automobiles to home theatre systems, from finding Mr. or Ms. "Perfect" to selecting attorneys or accountants, from books and newspapers to wikis and blogs, from movies to songs, and so on. In addition, we're constantly being bombarded by information—and occasionally misinformation! Under these conditions, the ability to recommend a choice is valuable, even more so if that choice doesn't deviate significantly from the preferences of the person who receives the recommendation.

In the business of influencing your choice, no one is interested in good results more than advertising companies. The *raison d'être* of these entities is to convince you that you really *need* product *X* or service *Y*. If you have no interest in products like *X* or services like *Y*, they'll be wasting their time and you'll be annoyed! The "broadcasting" approach of traditional advertising methods (such as billboards, TV ads, radio ads) suffers from that problem. The goal of broadcasting is to alter your preferences by incessantly repeating the same message. An alternative, more pleasant, and more effective approach would be targeting to your preferences. It would entice you to select a product based on its relevance to your personal wants and desires. That's where the online world and the intelligent advertisement business on the internet distinguish themselves. It may be the searching functionality that made Google famous, but advertisements are what make Google rich!

In this chapter, we'll tell you everything you need to know about building a recommendation engine. You'll learn about *collaborative filtering* and content-based recommendation engines. You'll also learn how to optimize the classical algorithms and how to extend them in more realistic applications. We'll start by describing the problem of recommending songs in an online music store, and we'll generalize it so that our proposed solutions are applicable to a variety of circumstances. The online music store is a simple example, but it's concrete and detailed, making it easy to understand all the basic concepts involved in the process of writing a recommendation engine.

Once we cover all the basic concepts in our online music store, we'll make things a lot more interesting by presenting more complicated cases. We'll adhere to the important principle of commercial proselytism and we'll cover recommendation engines that are crucial in online movie rentals (see our coverage of Netflix in the introduction), online bookstores, and general online stores.

3.1 An online music store: the basic concepts

Let's say that you have an online store that sells music downloads. Registered users log in to your application and can play samples of the available songs. If a user likes a particular song, she can add it to her shopping cart and purchase it later when she's ready to check out from your store. Naturally, when users complete their purchase, or when they land on the pages of our hypothetical application, we want to suggest more songs to them. There are millions of songs available, myriad artists, and dozens of music styles of broad interest to choose from—classical, ecclesiastical, pop, heavy metal, country, and many others more or less refined! In addition, many people are quite sensitive to the kind of music that they don't like. You'd be better off throwing me in the middle of the Arctic Ocean than showing me anything related to rap! Someone else could be allergic to classical music, and so on.

The moral of the story is that, when you display content for a user, you want to target the areas of music that the user likes and avoid the areas of music that the user doesn't like. If that sounds difficult, fear not! Recommendation engines are here to help you deliver the right content to your users!

A recommendation engine examines the selections that a user has made in the past, and can identify the degree to which he would like a certain item that he hasn't seen yet. It can be used to determine what types of music your user prefers, and the extent to which he does so, by comparing the *similarity* of his preferences with the characteristics of music types. In a more creative twist, we could help people establish a social network on that site based on the *similarity* of their musical taste. So, it quickly becomes apparent that the crucial functional element of recommendation engines is the ability to define how similar to each other two (or more) users or two (or more) items are. That similarity can later be leveraged to provide recommendations.

3.1.1 The concepts of distance and similarity

Let's take some data and start exploring these concepts in detail. The basic concepts that we'll work with are `Items`, `Users`, and `Ratings`. In the context of recommendation engines, similarity is a measure that allows us to compare the proximity of two items in much the same way that the proximity between two cities tells us how close they are to each other geographically. For two cities, we'd use their longitude and latitude coordinates to calculate their geographical proximity. Think of the `Ratings` as the "coordinates" in the space of `Items` or `Users`. Let's demonstrate these concepts in action. We'll select three users from a list of `MusicUsers` and will associate a list of songs (items) and their hypothetical rankings with each user.

As it is typically the case on the internet, ratings will range between 1 and 5 (inclusive). The assignments for the first two users (Frank and Constantine) involve ratings that are either 4 or 5—these people really like all the songs that we selected! But the third user's ratings (Catherine) are between 1 and 3. So clearly, we expect the first two users to be similar to each other and be dissimilar to the third user. When we load our example data in the script (the second line in the script of listing 3.1), we have available the users, songs, and ratings shown in table 3.1.

Table 3.1 The ratings for the users show that Frank and Constantine agreemore than Frank and Catherine (see also figure 3.2).

User	Song	Rating
Frank	Tears In Heaven	5
	La Bamba	4
	Mrs. Robinson	5
	Yesterday	4
	Wizard of Oz	5
	Mozart: Symphony #41 (Jupiter)	4
	Beethoven: Symphony No. 9 in D	5

Table 3.1 The ratings for the users show that Frank and Constantine agreemore than Frank and Catherine (see also figure 3.2). *(continued)*

User	Song	Rating
Constantine	Tears in Heaven	5
	Fiddler on the Roof	5
	Mrs. Robinson	5
	What a Wonderful World	4
	Wizard of Oz	4
	Let It Be	5
	Mozart: Symphony #41 (Jupiter)	5
Catherine	Tears in Heaven	1
	Mrs. Robinson	2
	Yesterday	2
	Beethoven: Symphony No. 9 in D	3
	Sunday, Bloody Sunday	1
	Yesterday	1
	Let It Be	2

We can execute all these steps in the shell using the script shown in listing 3.1.

Listing 3.1 A small list of `MusicUsers` and their `Ratings` on `MusicItems`

```
MusicUser[] mu = MusicData.loadExample();

mu[0].getSimilarity(mu[1],0);

mu[0].getSimilarity(mu[1],1);

mu[0].getSimilarity(mu[2],0);

mu[1].getSimilarity(mu[2],0);     <-- Similarity is symmetrical
mu[2].getSimilarity(mu[1],0);

mu[0].getSimilarity(mu[0],0);     <-- Similarity of a user with itself
mu[0].getSimilarity(mu[0],1);
```

We've provided two definitions of similarity, which are invoked by providing a different value in the second argument of the `getSimilarity` method of the `MusicUser` class. We'll describe the detailed implementation of that code shortly, but first look at figure 3.1, which shows the results that we get for the comparisons between the three users.

According to our calculations, shown in figure 3.1, Frank's preferences in songs are more similar to Constantine's than they are to Catherine's. The similarity between

```
bsh % MusicUser[] mu = MusicData.loadExample();

bsh % mu[0].getSimilarity(mu[1],0);

 User Similarity between Frank and Constantine is equal to
0.3911406349860862

bsh % mu[0].getSimilarity(mu[1],1);

 User Similarity between Frank and Constantine is equal to
0.22350893427776353

bsh % mu[0].getSimilarity(mu[2],0);

 User Similarity between Frank and Catherine is equal to 0.
004197074413470947

bsh % mu[1].getSimilarity(mu[2],0);

 User Similarity between Constantine and Catherine is equal to
0.0023790682635077554

bsh % mu[2].getSimilarity(mu[1],0);

 User Similarity between Catherine and Constantine is equal to
0.0023790682635077554

bsh % mu[0].getSimilarity(mu[0],0);

 User Similarity between Frank and Frank is equal to 1.0

bsh % mu[0].getSimilarity(mu[0],1);

User Similarity between Frank and Frank is equal to 1.0
```

Figure 3.1 Calculating the similarity of users for the data that are shown in table 3.1. It's clear that Frank and Constantine agree more than Frank and Catherine (see also table 3.1).

two users doesn't depend on the order in which we pass the arguments in the `getSimilarity` method. The similarity of Frank with himself is equal to 1.0, which we take to be the maximum value of similarity between any two entities. These properties stem from the fact that many similarity measures are based on distances, like the geometric distance between two points on a plane that we learned in high school.

In general, mathematical distances have the following four important properties:

- All distances are greater than or equal to zero. In most cases, as with the `MusicUser`, we constrain the similarities to be nonnegative like distances. In fact, we constrain the similarities within the interval [0,1].

- The distance between any two points, say A and B, is zero if and only if A is the same point as B. In our example, and based on our implementation of similarity, this property is reflected in the fact that when two users have exactly the same ratings, the similarity between them will be equal to 1.0. That's true in figure 3.1, where we used the same user twice to show that the similarity is 1.0. Of course, you can create a fourth user and prove that the similarity will be equal to 1, provided that the users have listened to the same songs.
- The third property of distances is *symmetry*—the distance between A and B is exactly the same as the distance between B and A. This means that if Catherine's musical taste is similar to the musical taste of Constantine, the reverse will also be true by exactly the same amount. So, quite often we want the measure of similarity to preserve the symmetric property of distances, with respect to its arguments.
- The fourth property of mathematical distances is the *triangle inequality* because it relates the distances between three points. In mathematical terms, if *d(A,B)* denotes the distance between points *A* and *B*, then the triangle inequality states that *d(A,B)* <= *d(A,C)* + *d(C,B)*, for any third point C. In figure 3.1, Frank is similar to Constantine by 0.391 and Constantine is similar to Catherine by 0.002, while Frank is similar to Catherine by 0.004, which is less than the sum of the first two similarities. Nevertheless, that property doesn't hold, in general, for our similarities.

Relaxing the fourth fundamental property of distances when we pass on to similarities is fine; there's no imperative to carry over the properties of distances to similarities. We should always be cautious to ensure that the mathematics involved is in agreement with what we consider to be reasonable. There's a century-old counterexample to the triangle inequality, when it comes to similarities, that's attributed to William James:[1] "A flame is similar to the moon because they are both luminous, and the moon is similar to a ball because they are both round, but in contradiction to the triangle inequality, a flame is not similar to a ball." For an interesting account of similarities in relation to cognition, we recommend *Classification and Cognition* by W.K. Estes.

At the top of figure 3.2, we show a visual representation of the similarity between Frank and Constantine by plotting their ratings for the songs they both rated. The closer the lines of the ratings, the more similar the users are; the further apart the lines, the less the similarity. On the bottom plot of figure 3.2, where we show the ratings of Frank versus those of Catherine, the lines diverge and are far apart, which is in accordance with the low similarity value that we got during our calculation.

The lines for Frank and Constantine are close, depicting the similarity between them. If you look at the code in the `plot` method of `MusicUser`, you'll see that we sort these ratings in order of increasing difference. If you have a lot of these ratings, you'll see the difference between the two lines increase as you look at the plot from left to right.

---

[1] Source: ScholarPedia (http://www.scholarpedia.org/article/Similarity_measures)

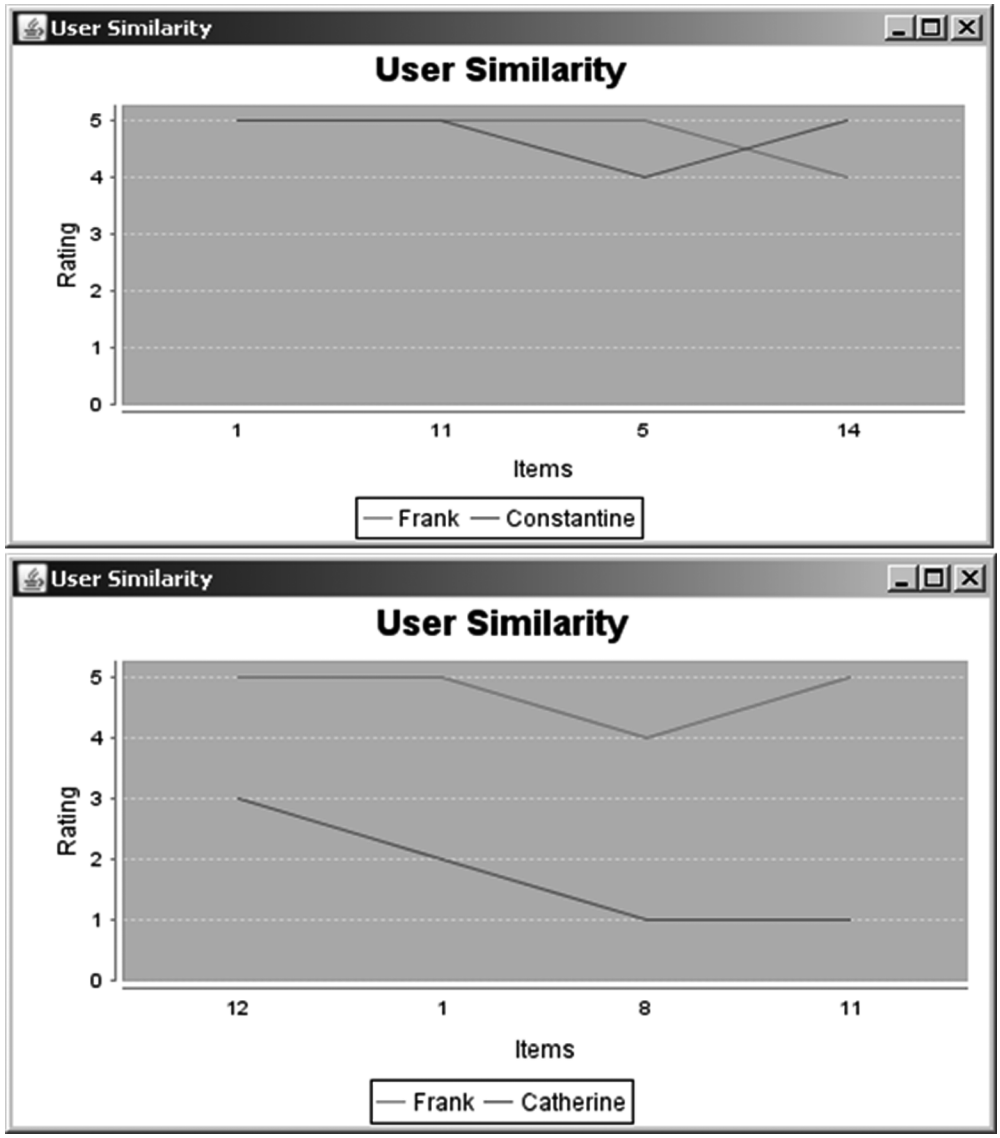

Figure 3.2 The similarity between two users can be measured by evaluating the extent of overlap between the two lines in plots like this. Thus, Frank and Constantine (top) are more similar than Frank and Catherine (bottom).

The plots of the ratings in figure 3.2 clearly display the somewhat reciprocal nature of distance and similarity. The greater the distance between the two curves, the smaller the similarity between the two users; the smaller the distance between the two curves, the greater the similarity between the two users. As we'll see in the next section, the evaluation of similarity often involves the evaluation of some kind of distance; although

that's not necessary. The concept of distance is more familiar. The concept of distance and the concept of similarity are special cases of the general concept of a metric.

3.1.2 A closer look at the calculation of similarity

Now, let's examine the code that helped us find the similarity between the users and look closely at how we can calculate similarity. The code in listing 3.2 shows the details of the `getSimilarity` method, which accepts two arguments. The first provides a reference to another user, the second specifies the kind of similarity that we want to use.

Listing 3.2 Two similarity measures in `getSimilarity` of `MusicUser`

```
public double getSimilarity(MusicUser u, int simType) {

  double sim=0.0d;
  int commonItems=0;

  switch(simType) {

  case 0:
    for (Rating r : this.ratingsByItemId.values()) {

        for (Rating r2 : u.ratingsByItemId.values()) {

        //Find the same item
        if ( r.getItemId() == r2.getItemId() ) {
         commonItems++;
         sim += Math.pow((r.getRating()-r2.getRating()),2);
        }
      }
    }

// If there are no common items, we cannot tell whether
// the users are similar or not. So, we let it return 0.
if (commonItems > 0) {

   sim = Math.sqrt(sim/(double)commonItems);

   // Similarity should be between 0 and 1
   // For the value 0, the two users are as dissimilar as they come
   // For the value 1, their preferences are identical.
   //
   sim = 1.0d - Math.tanh(sim);
}

break;

case 1:
  for (Rating r : this.ratingsByItemId.values()) {
    for (Rating r2 : u.ratingsByItemId.values()) {

        //Find the same item
        if ( r.getItemId() == r2.getItemId() ) {
         commonItems++;
         sim += Math.pow((r.getRating()-r2.getRating()),2);
        }
      }
    }
```

Identify all common items

Square differences of ratings and sum them

Identify all common items

Square differences of ratings and sum them

```
// If there are no common items, we cannot tell whether
// or not the users are similar. So, we let it return 0.
if (commonItems > 0) {

   sim = Math.sqrt(sim/(double)commonItems);

   // Similarity should be between 0 and 1
   // For the value 0, the two users are as dissimilar as they come
   // For the value 1, their preferences are identical.
   //
   sim = 1.0d - Math.tanh(sim);

   // Find the max number of items that the two users can have in common
   int maxCommonItems =
➥ Math.min(this.ratingsByItemId.size(), u.ratingsByItemId.size());

   // Adjust similarity to account for the importance of common terms
   // through the ratio of common items over all possible common items

   sim = sim * ((double)commonItems/(double)maxCommonItems);
}
break;

} //switch block ends

//Let us know what it is
System.out.print("\n"); //Just for pretty printing in the Shell
System.out.print(" User Similarity between");
System.out.print(" "+this.getName());
System.out.print(" and "+u.getName());
System.out.println(" is equal to "+sim);
System.out.print("\n"); //Just for pretty printing in the Shell

return sim;
}
```

We included two similarity formulas in the code to show that the notion of similarity is fairly flexible and extensible. Let's examine the basic steps in the calculation of these similarity formulas. First we take the differences between all the ratings of songs that the users have in common, square them, and add them together. The square root of that value is called the *Euclidean distance* and, as it stands, it's not sufficient to provide a measure of similarity. As we mentioned earlier, the concept of distance and similarity are somewhat reciprocal, in the sense that the smaller the value of the Euclidean distance, the more similar the two users. We can argue that the ordering incompatibility with the concept of similarity is easy to rectify. For instance, we could say that we'll add the value 1 to the Euclidean score and invert it.

At first sight, it appears that inverting the distance (after adding the constant value 1) might work. But this seemingly innocuous modification suffers from shortcomings. If two users have listened to only one song and one of them rated the song with 1 and the other rated the song with 4, the sum of their differences squared is 9. In that case, the naïve similarity, based on the Euclidean distance, would result in a similarity value of 0.25. The same similarity value can be obtained in other cases. If the two users listened to three songs and among these three songs, their ratings differed by 1 (for each song), their similarity would also be 0.25, according to the naïve

similarity metric. Intuitively we expect these two users to be more similar than those who listened to a single song and their opinions differed by 3 units (out of 5!).

The naïve similarity "squeezes" the similarity values for small distances (because we add 1) while leaving large distances (values of the distance much larger than 1) unaffected. What if we add another value? The general form of the naïve similarity is *y = beta / (beta + x)*, where beta is our free parameter and *x* is the Euclidean distance. Figure 3.3 shows what the naïve similarity would look like for various values, between 1 and 2, of the parameter `beta`.

Keeping in mind the shortcomings of the naïve similarity, let's look at the first similarity definition between two users as shown in listing 3.2, in the `case 0` block. If the users have some songs in common we divide the sum of their squared differences by the number of common songs, take the positive square root, and pass on that value to a special function. We've seen that function before: it's the hyperbolic tangent function. We subtract the value of the hyperbolic tangent from 1, so that our final value of similarity ranges between 0 and 1, with zero implying dissimilarity and 1 implying the highest similarity. Voilà! We've arrived at our first definition of similarity of users based on their ratings.

The second similarity definition that we present in listing 3.2, in the `case 1` block, improves on the first similarity by taking into account the ratio of the common items versus the number of all possible common items. That's a heuristic that intuitively makes sense. If I've listened to 30 songs and you've listened to 20, we could have up to 20 common songs. Let's say that we have only 5 songs in common and we agree fairly well on

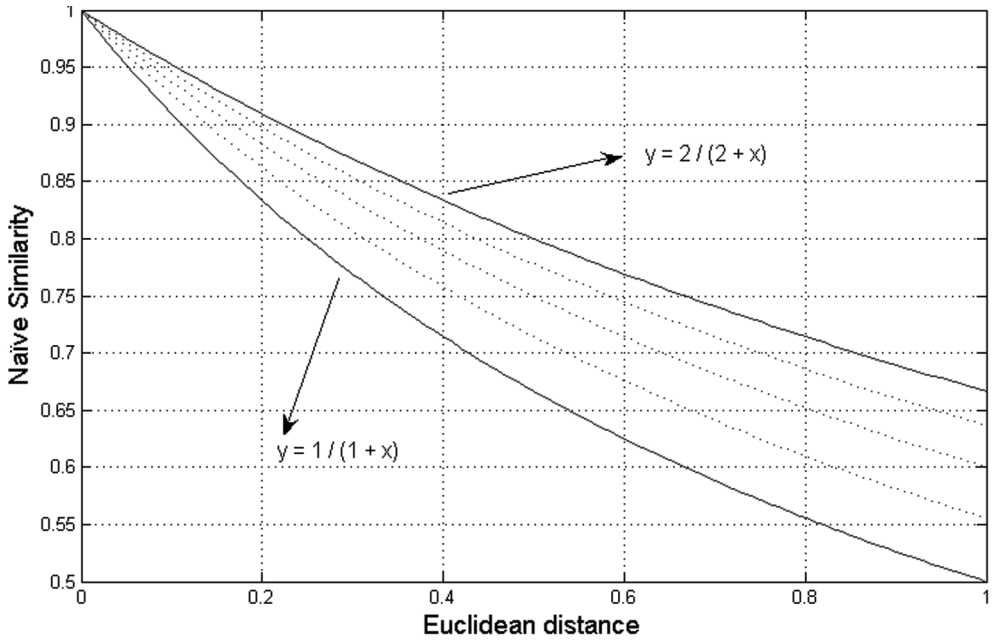

Figure 3.3 Naïve similarity curves as functions of the Euclidean distance

these songs, which is nice, but why don't we have more songs in common? Shouldn't that somehow be reflected in our similarity? This is exactly the aspect of the problem that we're trying to capture in the second similarity formula. In other words, the extent to which we listen to the same songs should somehow affect the degree of our similarity as music listeners.

3.1.3 Which is the best similarity formula?

It may be clear by now that there are many formulas you can use to establish the similarity between two users, or between two items for that matter. In addition to the two similarities that we introduced in the `MusicUser` class, we could've used a metric formula known as the *Jaccard similarity* between users, which is defined by the ratio of the intersection over the union of their item sets—or, in the case of item similarity, the ratio of the intersection over the union of the user sets. In other words, the Jaccard similarity between two sets, A and B, is defined by the following pseudocode: *Jaccard = intersection(A,B) / union(A,B)*. We'll use the Jaccard similarity in the next sections and will also present a few more similarity formulas in our "To do" section at the end of this chapter.

Of course, you may naturally wonder: "which similarity formula is more appropriate?" The answer, as always, is it depends. In this case, it depends on your data. In one of the few large-scale comparisons of similarity metrics (conducted by Spertus, Sahami, and Buyukkokten), the simple Euclidean distance-based similarity showed the best empirical results among seven similarity metrics, despite the fact that other formulas were more elaborate and intuitively expected to perform better. Their measurements were based on 1,279,266 clicks on related community recommendations from September 22, 2004, through October 21, 2004, on the social networking website Orkut (http://www.orkut.com); for more details, see the related reference.

We don't advise that you choose randomly your similarity metric, but if you're in a hurry, use a formula similar to the two that we included in the `MusicUser` class—the Euclidean or the Jaccard similarity. It should give you decent results. You should try to understand the nature of your data and what it means for two users or two items to be similar. If you don't understand the reasons why a particular similarity metric (formula) is good or bad, you're setting yourself up for trouble. To stress this point, think of the common misconception that "the shortest path between two points is a straight line that joins them." That statement is true only for what we call "flat" geometries, such as the area of a football field. To convince yourself, compare the distance of going over a tall but not wide hill versus going around the hill's base. The "straight" line will not be the shortest path for a wide range of hill sizes.

In summary, one of the cornerstones of recommendations is the ability to measure the similarity between any two users and the similarity between any two items. We've provided a number of similarity measures that you can use off-the-shelf, and the music store exemplified the typical structure of the data that you'd deal with in order to create recommendations. We'll now pass on to examine the types of recommendation engines and how they work.

3.2 How do recommendation engines work?

Armed with a good understanding of what similarity between two users or two items means, we can proceed with our description of recommendation engines. Generally speaking, there are two categories of recommendation engines. The first goes under the label *collaborative filtering (CF)*. The first incarnation of CF appeared in an experimental mail system (circa 1992) developed at the Xerox Palo Alto Research Center (PARC) by Goldberg et al. CF relies on the breadcrumbs that a user leaves behind through the interaction with a software system. Typically, these breadcrumbs are the user's ratings, such as the song ratings that we described in the previous section. Collaborative filtering isn't limited to one-dimensional or only discrete variables; its main characteristic is that it depends on the user's past behavior rather than the content of each item in the collection of interest. CF requires neither domain knowledge nor preliminary gathering and analysis work to produce recommendations.

The second broad category of recommendation engines is based on the analysis of the content—associated with the items or the users, or both. The main characteristic of this content-based approach is the accumulation and analysis of information related to both users and items. That information may be provided either by the software system or through external sources. The system can collect information about the users *explicitly* through their response to solicited questionnaires or *implicitly* through the mining of the user's profile or news reading habits, emails, blogs, and so on.

In the category of CF, we'll describe recommendations based on the similarity of users and of items. We'll also describe the category of content-based recommendations, thus covering all known recommendation engine systems.

3.2.1 Recommendations based on similar users

There's an ancient Greek proverb (with similar variants in nearly every culture of the world) that states: "Show me your friends and I'll tell you who you are." Collaborative filtering based on neighborhoods of similar users is more or less an algorithmic incarnation of that proverb. In order to evaluate the rating of a particular user for a given item, we look for the ratings of similar users (neighbors or friends, if you prefer) on the same item. Then, we multiply the rating of each friend by a weight and add them up. Yes, it's that simple, in principle!

Listing 3.3 shows a series of steps that demonstrate the creation and usage of a recommendation engine, which we called Delphi. First, we need to build data to work with. We create a sample of data by assigning ratings to songs for all users. For each user, we randomly pick a set of songs that corresponds to 80% of all the songs in our online music store. For each song assigned to a user, we assign a random rating that's either 4 or 5 if the username starts with the letters A through D (inclusive), and 1, 2, or 3 otherwise.

Thus, we establish two large groups of users with similar preferences; this allows us to quickly assess the results of our engine.

Listing 3.3 Creating recommendations based on similar users

```
BaseDataset ds = MusicData.createDataset();     <-- Create music dataset
ds.save("C:/iWeb2/deploy/data/ch3_2_dataset.ser");     <-- Save it for later
Delphi delphi = new Delphi(ds,RecommendationType.USER_BASED);
delphi.setVerbose(true);                                  <-- Create recommendation engine
MusicUser mu1 = ds.pickUser("Babis");
delphi.findSimilarUsers(mu1);
MusicUser mu2 = ds.pickUser("Lukas");                     <-- Find similar users
delphi.findSimilarUsers(mu2);
delphi.recommend(mu1);     <-- Recommend a few songs
```

The first line creates the dataset of our users and the ratings for the songs, in the way we described earlier. The code is straightforward and you can modify the data in the `MusicData` class as you see fit. In the second line, we store the dataset that we use in our example so we can refer to it later on. The third line creates an instance of our `Delphi` recommendation engine, and the fourth line sets it to verbose mode so that we can see the details of the results. Note that the constructors of `Delphi` use the interface `Dataset` rather than our example classes. You can use it with your own implementation straight out of the box—or more precisely out of the Java Archive (JAR).

Figure 3.4 shows the results of our script for the `findSimilarUsers` method. In the first case, the username starts with the letter B, and all the friends that are selected have names that start with the letters A through D. In the second case, the username starts with the letter J, and all the friends that are selected have names that start with the letters E through Z. In both cases, we obtain results that are in agreement with what we expected.

So, it seems that our recommendation engine is working well! Note also that the similarities between the friends of the first case are higher than the similarities of the group that corresponds to the second case because the ratings were distributed between only two values (4 and 5) in the first case, but in the second case were distributed among three values (1, 2, and 3). These kinds of sanity checks are useful, and you should always be alert of what an intelligent algorithm returns; it wouldn't be very intelligent if it didn't meet common sense criteria, would it?

In addition, figure 3.4 shows the results of the song recommendations for one of the users, as well as the predicted ratings for each recommendation. Note that although the ratings of the users are integers, the recommendation engine uses a `double` for its prediction. That's because the prediction expresses only a degree of belief about the rating rather than an actual rating. You may wonder why websites don't allow you to give a rating that's not an integer, or equally liberating, offer a rating between larger ranges of values, such as between 1 and 10 or even 1 and 100. We'll revisit this point in one of our to-do items at the end of the chapter.

Observe that the recommendation engine is correctly assigning values between 4 and 5, since the users whose letters start with the letters A through D have all given ratings that are either 4 or 5.

```
bsh % MusicUser mu1 = ds.pickUser("Bob");
bsh % delphi.findSimilarUsers(mu1);

Top Friends for user Bob:

name: Babis                                , similarity: 0.692308
name: Alexandra                            , similarity: 0.666667
name: Bill                                 , similarity: 0.636364
name: Aurora                               , similarity: 0.583333
name: Charlie                              , similarity: 0.583333

bsh % MusicUser mu2 = ds.pickUser("John");
bsh % delphi.findSimilarUsers(mu2);

Top Friends for user John:

name: George                               , similarity: 0.545455
name: Jack                                 , similarity: 0.500000
name: Elena                                , similarity: 0.461538
name: Lukas                                , similarity: 0.454545
name: Frank                                , similarity: 0.416667

bsh % delphi.recommend(mu1);

Recommendations for user Bob:

Item: I Love Rock And Roll          , predicted rating: 4.922400
Item: La Bamba                      , predicted rating: 4.758600
Item: Wind Beneath My Wings         , predicted rating: 4.540900
Item: Sunday, Bloody Sunday         , predicted rating: 4.526800
```

Figure 3.4 Discovering friends and providing recommendations with Delphi based on user similarity

How did the Delphi class arrive at these conclusions? How can it find the similar users (friends) for any given user? How can it recommend songs from the list of songs that a user never listened to? Let's go through the basic steps to understand what happens. Recommendation engines that are based on collaborative filtering proceed in two steps. First, they calculate the similarity between either users or items. Then, they use a weighted average to calculate the rating that a user would give to a yet-unseen item.

CALCULATING THE USER SIMILARITIES

Since we're dealing with recommendations that are based on user similarity, the first thing that `Delphi` does for us is to calculate the similarity between the users. This is shown in listing 3.4, where we show the code from the method `calculate` of the class `UserBasedSimilarity`, an auxiliary class that's used in `Delphi`. Note that the double loop has been optimized to account for the symmetry of the similarity matrix; we discuss this and one more optimization after the code listing.

Listing 3.4 `UserBasedSimilarity`: calculating the user similarity

```
protected void calculate(Dataset dataSet) {

  int nUsers = dataSet.getUserCount();          <-- Defines size of similarity matrix

  int nRatingValues = 5;                        <-- Defines size of
                                                    rating count matrix
  similarityValues = new double[nUsers][nUsers];

  if( keepRatingCountMatrix ) {
     ratingCountMatrix = new RatingCountMatrix[nUsers][nUsers];
  }

  // if mapping from userId to index then generate index for every userId
  if( useObjIdToIndexMapping ) {

     for(User u : dataSet.getUsers() ) {
        idMapping.getIndex(String.valueOf(u.getId()));
     }
  }

  for (int u = 0; u < nUsers; u++ ) {

     int userAId = getObjIdFromIndex(u);
     User userA = dataSet.getUser(userAId);

     for (int v = u + 1; v < nUsers; v++) {     <-- (1) Similarity matrix

        int userBId = getObjIdFromIndex(v);
        User userB = dataSet.getUser(userBId);

        RatingCountMatrix rcm =                 <-- Agreement of
     ➥  new RatingCountMatrix(userA, userB, nRatingValues);    ratings between
                                                                two users
        int totalCount = rcm.getTotalCount();
        int agreementCount = rcm.getAgreementCount();

        if (agreementCount > 0) {               <-- Calculate
                                                    similarity or
          similarityValues[u][v] =                  set it to zero
     ➥  (double) agreementCount / (double) totalCount;

        } else {
           similarityValues[u][v] = 0.0;
        }

        // For large datasets
        if( keepRatingCountMatrix ) {
           ratingCountMatrix[u][v] = rcm;
        }
     }

     // for u == v assign 1.
     // RatingCountMatrix wasn't created for this case
     similarityValues[u][u] = 1.0;              <-- (1) Similarity matrix
  }
}
```

(1) Here is the optimization that we mentioned earlier. You'd expect the first loop to select the first user and the second loop to select all other users. But in the listing, the

second loop uses the fact that the similarity matrix is symmetrical. This simply means that if user *A* is similar to user *B* with a similarity value *X* then user *B* will be similar to user *A* with a similarity value equal to *X*. The code avoids evaluating the similarity of a user object with itself, because that should always be equal to 1. These two code optimizations are simply a reflection of the fundamental properties that every similarity measure should obey, as stated in section 3.1.1.

As you can see, the definition of similarity is given by the Jaccard metric, where the agreement on the ratings represents the intersection between the two sets of ratings, and the total count of ratings represents the union of the two sets of ratings. Similarity values are held in a two-dimensional array of type `double`. But similarity is a symmetrical property, which simply means that if I'm similar to you then you're similar to me, regardless of how similarity was defined. So clearly, we can use the similarity values much more efficiently by either using sparse matrices or by using some other structure that's designed to store only half the number of values; the latter structure is technically known as the *upper triangular form* of the matrix. From a computational perspective, we're already leveraging that fact in the code of listing 3.4. Once again, note that the second loop doesn't run over all users, but starts with the user that follows the outer loop user in our list.

The calculation of similarity for each pair of users relies on an auxiliary class that we called `RatingCountMatrix`. The purpose of the class is to store the rating of one user with respect to another in a nice tabular format and allow us to calculate the final similarity value easily and transparently. Listing 3.5 contains the code for `RatingCountMatrix`.

Listing 3.5 Storing the agreement distribution of two users in a tabular form

```
public class RatingCountMatrix implements Serializable {

  private int matrix[][] = null;

  public RatingCountMatrix(Item itemA, Item itemB,
    int nRatingValues) {

      init(nRatingValues);

      calculate(itemA, itemB);
  }

  public RatingCountMatrix(User userA, User userB,
    int nRatingValues) {

      init(nRatingValues);

      calculate(userA, userB);
  }

  private void init(int nSize) {
      // starting point - all elements are zero
      matrix = new int[nSize][nSize];
  }
```

Calculate item-based similarity

Initialize rating count matrix

Calculate user-based similarity

```
private void calculate(Item itemA, Item itemB) {

    for (Rating ratingForA : itemA.getAllRatings()) {

       // check if the same user rated itemB
       Rating ratingForB =
➥  itemB.getUserRating(ratingForA.getUserId());

       if (ratingForB != null) {

          int i = ratingForA.getRating() - 1;
          int j = ratingForB.getRating() - 1;

          matrix[i][j]++;
       }
    }
 }

 private void calculate(User userA, User userB) {

    for (Rating ratingByA : userA.getAllRatings()) {

       Rating ratingByB =
➥  userB.getItemRating(ratingByA.getItemId());

       if (ratingByB != null) {

          int i = ratingByA.getRating() - 1;
          int j = ratingByB.getRating() - 1;

          matrix[i][j]++;
       }
    }
 }

public int getTotalCount() {

    int ratingCount = 0;
    int n = matrix.length;

    for (int i = 0; i < n; i++) {
       for (int j = 0; j < n; j++) {
          ratingCount += matrix[i][j];
       }
    }

    return ratingCount;
}

public int getAgreementCount() {

    int ratingCount = 0;
    for (int i = 0, n = matrix.length; i < n; i++) {
       ratingCount += matrix[i][i];
    }

    return ratingCount;
 }

 public int getBandCount(int bandId) {
    int bandCount = 0;
    for (int i = 0, n = matrix.length; (i + bandId) < n; i++) {
       bandCount += matrix[i][i + bandId];
```

Calculate item-based similarity

Calculate user-based similarity

Auxiliary methods for various counters

```
            bandCount += matrix[i + bandId][i];
        }
        return bandCount;
    }
}
```

The heart of that class is the two-dimensional `int` array (5-by-5, in this case) that stores the agreement rate of two users based on their ratings. Let's say that user *A* and user *B* both listened to 10 songs, and agreed on 6 and disagreed on the rest. The matrix is initialized to zero for all its elements; for every agreement, we add the value 1 in the row and column that corresponds to the rating. So, if three of the agreements were for a rating with value 4, and another three were for the rating 5, then the `matrix[3][3]` and the `matrix[4][4]` elements will both be equal to 3. In general, if you add the diagonal elements of the `matrix` array, you'll find the number of times that the two users agreed on their ratings.

This way of storing the ratings of your users has several advantages. First, you can treat ratings that are from 1 to 10 (or 100 for that matter) in exactly the same way that you treat ratings that are from 1 to 5. Second, as we'll see later, it gives you the opportunity to derive more elaborate similarity measures that account not only for the number of times that two users agreed on their ratings but also for the number of times and the extent to which they disagreed. Third, it's possible to generalize this matrix form into a more general object that may not be a simple two-dimensional array but a more complicated structure; this may be desirable in a situation where your assessment relies on more than a simple rating.

THE INNER WORKINGS OF DELPHI

Now, the code in listing 3.4 has been fully explained. The similarity value between user *A* and user *B*, in this case, is simply the ratio of the number of times that user *A* agreed with number *B* divided by the total number of times that both users rated a particular item. Thus, we're one step away from creating our recommendations.

Listing 3.6 `Delphi`: creating recommendations based on user similarity

```
public List<PredictedItemRating> recommend(User user, int topN) {

 List<PredictedItemRating> recommendations =
➥ new ArrayList<PredictedItemRating>();

 for (Item item : dataSet.getItems()) {     <-- Loop through all items

    // only consider items that the user hasn't rated yet
    if (user.getItemRating(item.getId()) == null) {

      double predictedRating = predictRating(user, item);     <-- Predict ratings for this user

  if (!Double.isNaN(predictedRating)) {
    recommendations.add(new PredictedItemRating(user.getId(),
                    item.getId(), predictedRating));
  }
    }                                                  Add prediction as
  }                                                    candidate recommendation
```

```
 Collections.sort(recommendations);
 Collections.reverse(recommendations);

 List<PredictedItemRating> topRecommendations =
➥ new ArrayList<PredictedItemRating>();

 for(PredictedItemRating r : recommendations) {
    if( topRecommendations.size() >= topN ) {
           // had enough recommendations.
           break;
    }
     topRecommendations.add(r);
  }

  return recommendations;
}
```

Sort candidate recommendations

Select top N recommendations

Listing 3.6 shows the high-level method `recommend` of `Delphi`, which is invoked for providing recommendations, as we've seen in listing 3.3. This method omits from consideration the items that a user has already rated. This may or may not be desirable; consider your own requirements before using the code as-is. If you had to change it, you could change the behavior in this method; for example, you could provide an `else` clause in the first `if` statement.

The `recommend` method delegates the rating prediction of a `user` (the first argument) to the method `predictRating(user, item)` for each `item`, which in turn delegates the calculation of the weighted average to the method `estimateUserBasedRating`. Listing 3.7 presents the method `predictRating(user, item)`. The purpose of that method is to create a façade that hides all the possible implementations of evaluating similarity, such as user-based similarity, item-based similarity and so on. Some cases are suggested but not implemented, so that you can work on them!

Listing 3.7 Predicting the rating of an item for a user

```
public double predictRating(User user, Item item) {

   switch (type) {

      case USER_BASED:
         return estimateUserBasedRating(user, item);

      case ITEM_BASED:
         return estimateItemBasedRating(user, item);

      case USER_CONTENT_BASED:
         throw new IllegalStateException(
➥ "Not implemented similarity type:" + type);

      case ITEM_CONTENT_BASED:
         throw new IllegalStateException(
➥ "Not implemented similarity type:" + type);

      case USER_ITEM_CONTENT_BASED:
         return MAX_RATING * similarityMatrix
➥ .getValue(user.getId(), item.getId());
```

```
    }

    throw new RuntimeException("Unknown type:" + type);
  }
```

The method `estimateUserBasedRating` is the user-based implementation for predicting the rating of a user. If we know the rating of a user there's no reason for any calculation. This isn't possible in the execution flow that we described in listing 3.6 because we invoke the method call only for those items that the user hasn't yet rated. But the code was written in a way that handles independent calls to this method as well.

Listing 3.8 Evaluating user-based similarities

```
private double estimateUserBasedRating(User user, Item item) {

  double estimatedRating = Double.NaN;

  int itemId = item.getId();
  int userId = user.getId();

  double similaritySum = 0.0;

  double weightedRatingSum = 0.0;

  // check if user has already rated this item
  Rating existingRatingByUser = user.getItemRating(item.getId());

  if (existingRatingByUser != null) {

   estimatedRating = existingRatingByUser.getRating();

  } else {

     for (User anotherUser : dataSet.getUsers()) {

        Rating itemRating = anotherUser.getItemRating(itemId);

        // only consider users that rated this book
        if (itemRating != null) {

          double similarityBetweenUsers =

            similarityMatrix.getValue(userId, anotherUser.getId());

          double ratingByNeighbor = itemRating.getRating();

          double weightedRating =
➥ similarityBetweenUsers * ratingByNeighbor;

          weightedRatingSum += weightedRating;

          similaritySum += similarityBetweenUsers;
        }
      }

      if (similaritySum > 0.0) {
         estimatedRating = weightedRatingSum / similaritySum;
      }
  }

  return estimatedRating;
}
```

Get rating for same item

Loop over all other users

Get similarity between two users

Scale rating according to similarity

Estimate rating as ratio of direct and scaled sum

In the more interesting case where the user hasn't yet rated a specific item, we loop over all users and identify those who've rated the specific `item`. Each one of these users contributes to the weighted average rating in direct proportion to his similarity with our reference `user`. The `similaritySum` variable is introduced for normalization purposes—the weights must add up to 1.

As you can see in listings 3.4 through 3.6, this way of creating recommendations can become extremely difficult if the number of users in your system becomes large, which is often the case in large online stores. Opportunities for optimizing this code abound. We already mentioned storage optimization, but we can also implement another structural change that will result in both space and time efficiency during runtime. While calculating the similarity between users, we can store the top *N* similar users and create our weighted rating (prediction) based on the ratings of these users alone rather than taking into account the ratings of all users that have rated a given item; that's the version known as *kNN*, where *NN* stands for nearest neighbors and *k* denotes how many of them we should consider. Creating recommendations based on user similarity is a reliable technique, but it may not be efficient for large number of users; in this case, the use of item-based similarity is preferred.

3.2.2 Recommendations based on similar items

Collaborative filtering based on similar items works in much the same way as CF based on similar users, except that the similarity between users is replaced by the similarity between items. Let's configure `Delphi` to work based on the similarity between the items (music songs) and see what we get. Listing 3.9 shows the script that we use for that purpose. We load the data that we saved in listing 3.3 and request recommendations for the same user in order to compare the results. We also request the list of similar items for the song "La Bamba," which appears on both lists.

Listing 3.9 Creating recommendations based on similar items

```
BaseDataset ds = BaseDataset
➥ .load("C:/iWeb2/deploy/data/ch3_2_dataset.ser");

Delphi delphi = new Delphi(ds,RecommendationType.ITEM_BASED);
delphi.setVerbose(true);

MusicUser mu1 = ds.pickUser("Bob");
delphi.recommend(mu1);

MusicItem mi = ds.pickItem("La Bamba");
delphi.findSimilarItems(mi);
```

Load same data as in listing 3.3

Create item-based recommendation engine

Recommend a few items to Bob

Find items similar to La Bamba

Figure 3.5 shows the results of execution for listing 3.9. If you compare these results with the results shown in figure 3.4, you'll see that the recommendations are the same but the order has changed. There's no guarantee that the recommendations based on user similarity will be identical to those based on item similarity. In addition, the scores will almost certainly be different. The interesting part in the specific example of our artificially generated data is that the ordering of the recommendations has been

```
bsh % MusicUser mu1 = ds.pickUser("Bob");
bsh % delphi.recommend(mu1);

Recommendations for user Bob:

    Item: Sunday, Bloody Sunday  , predicted rating: 4.483900
    Item: La Bamba               , predicted rating: 4.396600
    Item: I Love Rock And Roll   , predicted rating: 4.000000
    Item: Wind Beneath My Wings  , predicted rating: 4.000000

bsh % MusicItem mi = ds.pickItem("La Bamba");
bsh % delphi.findSimilarItems(mi);

Items like item La Bamba:

    name: Yesterday                   , similarity: 0.615385
name: Fiddler On The Roof             , similarity: 0.588235
name: Vivaldi: Four Seasons           , similarity: 0.555556
name: Singing In The Rain             , similarity: 0.529412
name: You've Lost That Lovin' Feelin' , similarity: 0.529412
```

Figure 3.5 Discovering similar items and providing recommendations with Delphi based on item similarity

inverted. That's not a general result; it just happened in this case. In other cases, and particularly in real datasets, the results can have any other ordering; run the scripts a few times to see how the results vary each time you generate a different dataset.

The code for creating recommendations based on item similarity is much the same, with the exception that we use items instead of users, of course. The calculation takes place in the method `calculate` of the class `ItemBasedSimilarity`.

Listing 3.10 Calculating the item-based similarity

```
protected void calculate(Dataset dataSet) {

  int nItems = dataSet.getItemCount();

  int nRatingValues = 5;

  similarityValues = new double[nItems][nItems];

  if( keepRatingCountMatrix ) {
     ratingCountMatrix = new RatingCountMatrix[nItems][nItems];
  }

  // if mapping from itemId to index then generate index for every itemId
  if( useObjIdToIndexMapping ) {
    for(Item item : dataSet.getItems() ) {
       idMapping.getIndex(String.valueOf(item.getId()));
    }
  }

  for (int u = 0; u < nItems; u++) {

    int itemAId = getObjIdFromIndex(u);
```

Defines size of similarity matrix

Defines size of rating count matrix

```
    Item itemA = dataSet.getItem(itemAId);

    // we only need to calculate elements above the main diagonal.
    for (int v = u + 1; v < nItems; v++) {                          (1)

      int itemBId = getObjIdFromIndex(v);
      Item itemB = dataSet.getItem(itemBId);

      RatingCountMatrix rcm =
    ➥ new RatingCountMatrix(itemA, itemB, nRatingValues);

      int totalCount    = rcm.getTotalCount();
      int agreementCount = rcm.getAgreementCount();

      if (agreementCount > 0) {

        similarityValues[u][v] =
    ➥   (double) agreementCount / (double) totalCount;

      } else {

        similarityValues[u][v] = 0.0;
      }

      if( keepRatingCountMatrix ) {
        ratingCountMatrix[u][v] = rcm;
      }
    }

    // for u == v assign 1
    similarityValues[u][u] = 1.0;                                   (1)
  }
}
```

Agreement of ratings between two items

Calculate similarity or set to zero

This is the same code optimization (1) that we've seen for the user-based similarity evaluation in listing 3.4.

The `RatingCountMatrix` class is used once again to keep track of the agreement versus disagreement in the ratings, although now, the agreement/disagreement is between the ratings of two different items rather than two different users. The code iterates through all the possible pairs of items and assigns similarity values based on the Jaccard metric. The code in the `Delphi` class for item-based recommendations closely follows the corresponding code for user-based recommendations. In listing 3.11, we show the evaluation of the similarity for item-based recommendations; compare it with the code in listing 3.8. The code in listings 3.6 and 3.7 is identical for all types of similarity evaluation.

Listing 3.11 `Delphi`: creating recommendations based on item similarity

```
private double estimateItemBasedRating(User user, Item item) {

  double estimatedRating = Double.NaN;

  int itemId = item.getId();
  int userId = user.getId();

  double similaritySum = 0.0;
  double weightedRatingSum = 0.0;
```

```
// check if the user has already rated the item
Rating existingRatingByUser = user.getItemRating(item.getId());

if (existingRatingByUser != null) {

  estimatedRating = existingRatingByUser.getRating();

} else {

  double similarityBetweenItems = 0;
  double weightedRating = 0;

  for (Item anotherItem : dataSet.getItems()) {

    // only consider items that were rated by the user
    Rating anotherItemRating = anotherItem.getUserRating(userId);

    if (anotherItemRating != null) {

      similarityBetweenItems =
➥ similarityMatrix.getValue(itemId, anotherItem.getId());

      if (similarityBetweenItems > similarityThreshold) {

        weightedRating =
➥ similarityBetweenItems * anotherItemRating.getRating();

        weightedRatingSum += weightedRating;

        similaritySum += similarityBetweenItems;
      }
    }
  }

  if (similaritySum > 0.0) {

    estimatedRating = weightedRatingSum / similaritySum;
  }
}
return estimatedRating;
}
```

Get rating for same user

Loop over all other items

Get similarity between two items

Scale rating according to similarity

Estimate rating as ratio of direct and scaled sum

These listings complete our initial coverage of collaborative filtering, or creating recommendations based on users and items. Typically, CF based on item similarity is preferred because the number of customers is large (millions or even tens of millions), but sometimes in the pursuit of better recommendations, the two CF methods are combined. In the following sections, we'll present the examples of customizing a site like Amazon.com (http://www.amazon.com), which employs an item-to-item collaborative approach, and providing recommendations on a site like Netflix.com (http://www.netflix.com), which will demonstrate the combination of the two methods.

3.2.3 Recommendations based on content

Creating recommendations based on content relies on the similarity of content between users, between items, or between users and items. Instead of ratings, we now have a measure of how "close" two documents are. The notion of distance between documents is a generalization of the relevance score between a query and a document, something that we discussed in chapter 2. You can always think of one document as the

query and the other document as reference. Of course, you'd have to compare only the significant parts of each document; otherwise the information that each document carries may be lost by obfuscation.

CASE STUDY SETUP

We'll use the documents from chapter 2 as sources of content and assign a number of these web pages to each user, in a way that resembles the assignment of songs to users in our earlier example. For each user, we'll randomly pick a set of pages that corresponds to 80% of all the eligible pages from our collection. Eligible documents for each user are introduced with a strong bias as follows:

- If the username starts with the letters A through D (inclusive), we assign 80% of the documents that belong to either the Business or the Sports category.
- Otherwise, we assign 80% of the documents that belong to either the USA or the World category

Thus, we establish two large groups of users with similar (although somewhat artificial) preferences, which will allow us to quickly assess our results. Let's see the steps of creating content-based instances of our `Delphi` recommender. Listing 3.12 shows the code that prepares the data and then identifies similar users and similar items. We also provide the recommendation of items based on a hybrid user-item content-based similarity.

Listing 3.12 Creating recommendations based on content similarities

```
BaseDataset ds = NewsData.createDataset();

Delphi delphiUC = new Delphi(ds,RecommendationType.USER_CONTENT_BASED);
delphiUC.setVerbose(true);
                                                    Create user-content-
NewsUser nu1 = ds.pickUser("Bob");                      based engine
delphiUC.findSimilarUsers(nu1);

NewsUser nu2 = ds.pickUser("John");
delphiUC.findSimilarUsers(nu2);

Delphi delphiIC = new Delphi(ds,RecommendationType.ITEM_CONTENT_BASED);
delphiIC.setVerbose(true);
                                                    Create item-content-
ContentItem i = ds.pickContentItem("biz-05.html");      based engine
delphiIC.findSimilarItems(i);

Delphi delphiUIC =
   new Delphi(ds,RecommendationType.USER_ITEM_CONTENT_BASED);
delphiUIC.setVerbose(true);
                                                    Create user-item-
delphiUIC.recommend(nu1);                           content-based engine
```

The first line of the script creates the dataset in the way that we described earlier. Once we get the dataset, we create a `Delphi` instance that's based on a user-to-user similarity matrix that we calculate in the class `UserContentBasedSimilarity`. Since each user has more than one document, we must compare each document of each user with each document of every other user. There are many ways to do this. In our code, as shown in listing 3.13, for each user-pair combination—user *A* and user *B*—we

loop over each document of *A* and find the document of *B* with the highest similarity. Then we average the best similarities for each document of *A* and assign the average value as the similarity between *A* and *B*.

Listing 3.13 Calculating the similarity of users based on their content

```
protected void calculate(Dataset dataSet) {

  int nUsers = dataSet.getUserCount();

  similarityValues = new double[nUsers][nUsers];

  // if mapping from userId to index then generate index for every userId
  if( useObjIdToIndexMapping ) {
    for(User u : dataSet.getUsers() ) {
       idMapping.getIndex(String.valueOf(u.getId()));
    }
  }

  CosineSimilarityMeasure cosineMeasure =
➥ new CosineSimilarityMeasure();

  for (int u = 0; u < nUsers; u++ ) {

    int userAId = getObjIdFromIndex(u);
    User userA = dataSet.getUser(userAId);

    for (int v = u + 1; v < nUsers; v++) {

      int userBId = getObjIdFromIndex(v);
      User userB = dataSet.getUser(userBId);

      double similarity = 0.0;

      for(Content userAContent : userA.getUserContent() ) {

        double bestCosineSimValue = 0.0;

        for(Content userBContent : userB.getUserContent() ) {

          double cosineSimValue = cosineMeasure
➥ .calculate(userAContent.getTFMap(), userBContent.getTFMap());

          bestCosineSimValue =
➥ Math.max(bestCosineSimValue, cosineSimValue);
        }

        similarity += bestCosineSimValue;
      }

      similarityValues[u][v] = similarity /
➥ userA.getUserContent().size();
    }

    // for u == v assign 1.
    similarityValues[u][u] = 1.0;
  }
}
```

❶

Create cosine similarity measure

Iterate over all rated items of user A

Iterate over all rated items of user B

Aggregate best similarities from all documents

Calculate similarity as simple average

This is the same code optimization ❶ that we've seen for the user-based similarity evaluation in listing 3.4.

THE KEY IDEAS BEHIND CONTENT-BASED SIMILARITIES

The key element to all content-based methods is representing the textual information as a numerical quantity. An easy way to achieve this is to identify the *N* most frequent terms in each document and use the set of most frequent terms across all documents as a coordinate space. We can take advantage of Lucene's `StandardAnalyzer` class to eliminate stop words and stem the terms to their roots, thus amplifying the importance of the meaningful terms while reducing the noise significantly. For that purpose, we've created a `CustomAnalyzer` class, which extends the `StandardAnalyzer`, in order to remove some words that are common and, if present, would add a significant level of noise to our vectors.

Let's digress for awhile here to make these important ideas more concrete. For argument's sake, let's say that $N = 4$ and that you have three documents and the following (high frequency) terms:

- D1 = {Google, shares, advertisement, president}
- D2 = {Google, advertisement, stock, expansion}
- D3 = {NVidia, stock, semiconductor, graphics}

Each of these documents can be represented mathematically by a nine-dimensional vector that reflects whether a specific document contains one of the nine unique terms—{Google, shares, advertisement, president, stock, expansion, Nvidia, semiconductor, graphics}. So, these three documents would be represented by the following three vectors:

- D1 = {1,1,1,1,0,0,0,0,0}
- D2 = {1,0,1,0,1,1,0,0,0}
- D3 = {0,0,0,0,1,0,1,1,1}

Voilà! We constructed three purely mathematical quantities that we can use to compare our documents quantitatively. The similarity that we're going to use is called the *cosine similarity*. We've seen many similarity formulas so far, and this isn't much different. Instead of bothering you with a mathematical formula, we'll list the class that encapsulates its definition. Listing 3.14 shows the code from the `CosineSimilarityMeasure` class.

Listing 3.14 Calculating the cosine similarity between term vectors

```
public class CosineSimilarityMeasure {

   public double calculate(double[] v1, double[] v2) {

      double a = getDotProduct(v1, v2);

      double b = getNorm(v1) * getNorm(v2);

      return a / b;
   }

   private double getDotProduct(double[] v1, double[] v2) {
```

Get cosine similarity

Normalize two vectors and calculate product

Find dot product

```
        double sum = 0.0;

        for(int i = 0, n = v1.length; i < n; i++) {
            sum += v1[i] * v2[i];
        }

        return sum;
    }
    private double getNorm(double[] v) {
        double sum = 0.0;

        for( int i = 0, n = v.length; i < n; i++) {
            sum += v[i] * v[i];
        }

        return Math.sqrt(sum);
    }
}
```

Calculate Euclidean norm of a vector

As you can see, first we form what's called the *dot (inner) product* between the two vectors—the `double` variable `a`. Then we calculate the norm (magnitude) of each vector and store their product in the `double` variable `b`. The cosine similarity is simply the ratio *a/b*. If we denote the cosine similarity between document *X* and document *Y* as CosSim(X,Y), for our simple example, we have the following similarities:

- *CosSim(D1,D2) = 2 / (2*2) = 0.5*
- *CosSim(D1,D3) = 0 / (2*2) = 0*
- *CosSim(D2,D3) = 1 / (2*2) = 0.25*

The technique of representing documents based on their terms is fundamental in information retrieval. We should point out that identifying the terms is a crucial step, and it's difficult to get it right for a general corpus of documents. For example, modify our code to use the `StandardAnalyzer` instead of our own `CustomAnalyzer`. What do you observe? The results can be altered significantly, even though at first sight, there's not much in our custom class. This small experiment should convince you that the content-based approach is very sensitive to the lexical analysis stage.

THREE TYPES OF CONTENT-BASED RECOMMENDATIONS

Coming back to our example, let's have a look at the results. Figure 3.6 shows a part of the results from executing the code in listing 3.12, which is responsible for finding similar users.

The algorithm is successful because it correctly identifies the two distinct groups as similar—users whose names start with *A* through *D* and users whose names start with *E* through *Z*. Note that the values of similarity don't vary much. The content-based approach doesn't seem to produce a good separation between the users when they're compared with each other. Figure 3.7 shows the execution of the code that's responsible for finding similar items. As you can see, a number of relevant items have been identified, but so were a number of items that a human user wouldn't find very similar.

```
bsh % BaseDataset ds = NewsData.createDataset();
bsh % Delphi delphiUC =
new Delphi(ds,RecommendationType.USER_CONTENT_BASED);

bsh % delphiUC.setVerbose(true);
bsh % NewsUser nu1 = ds.pickUser("Bob");
bsh % delphiUC.findSimilarUsers(nu1);

Top Friends for user Bob:

    name: Albert        , similarity: 0.950000
    name: Catherine     , similarity: 0.937500
    name: Carl          , similarity: 0.937500
    name: Alexandra     , similarity: 0.925000
    name: Constantine   , similarity: 0.925000

bsh % NewsUser nu2 = ds.pickUser("John");
bsh % delphiUC.findSimilarUsers(nu2);

Top Friends for user John:

    name: George        , similarity: 0.928571
    name: Lukas         , similarity: 0.914286
    name: Eric          , similarity: 0.900000
    name: Nick          , similarity: 0.900000
    name: Frank         , similarity: 0.900000
```

Figure 3.6 Users who are similar to Bob have names that start with the letters A through D. The algorithm identified the two groups of similar users successfully!

Once again, you can see that the similarity values don't vary much; it would be difficult for the algorithm to provide excellent recommendations. The reason for that lack of disambiguation lies in the paucity of our lexical analysis. *Natural language processing* (NLP)) is a rich and difficult field. Nevertheless, much progress has been made in the last two decades; although we won't go in-depth on that fascinating subject in this book, we'll summarize the various components of a NLP system in appendix D.

In figure 3.8 we present recommendations based on user-item similarity. Although CF usually deals with user-user or item-item similarities, a content-based approach is advantageous for building recommendations on user-item similarities. Nevertheless, the problems of lexical analysis remain, and without tedious and specific work based on NLP, the results won't be satisfactory. If you enlarge the dataset and run the script several times for different users, a large number of the recommendations will have identical ratings and the predicted ratings won't vary significantly.

In summary, recommendation systems are built around user-user, item-item, and content-based similarities. Creating recommendations based on user similarity is a reliable technique but may not be efficient for a large number of users. In the latter case, collaborative filtering based on item similarity is preferred because the number of customers (millions or even tens of millions) is orders of magnitude larger than the

```
bsh % Delphi delphiIC =
new Delphi(ds,RecommendationType.ITEM_CONTENT_BASED);

bsh % delphiIC.setVerbose(true);
bsh % ContentItem biz1 = ds.pickContentItem("biz-01.html");
bsh % delphiIC.findSimilarItems(biz1);

Items like item biz-01.html:

    name: biz-03.html    , similarity: 0.600000
    name: biz-02.html    , similarity: 0.600000
    name: biz-04.html    , similarity: 0.100000
    name: biz-07.html    , similarity: 0.100000

bsh % ContentItem usa1 = ds.pickContentItem("usa-01.html");
bsh % delphiIC.findSimilarItems(usa1);

Items like item usa-01.html:

    name: usa-02.html    , similarity: 0.300000
    name: usa-03.html    , similarity: 0.300000
    name: world-03.html  , similarity: 0.100000
    name: world-05.html  , similarity: 0.100000
    name: usa-04.html    , similarity: 0.100000

bsh % ContentItem sport1 = ds.pickContentItem("sport-01.html");
bsh % delphiIC.findSimilarItems(sport1);

Items like item sport-01.html:

    name: sport-03.html  , similarity: 0.400000
    name: sport-02.html  , similarity: 0.300000
```

Figure 3.7 Items that belong in the same category as the query item are correctly identified as similar.

```
bsh % Delphi delphiUIC = new Delphi(
➥  ds,RecommendationType.USER_ITEM_CONTENT_BASED);
bsh % delphiUIC.setVerbose(true);
bsh % delphiUIC.recommend(nu1);

Recommendations for user Bob:

    Item: biz-06.html      , predicted rating: 2.500000
    Item: biz-04.html      , predicted rating: 1.500000
    Item: usa-02.html      , predicted rating: 0.500000
    Item: world-03.html    , predicted rating: 0.500000
    Item: world-05.html    , predicted rating: 0.500000
```

Figure 3.8 We obtain item recommendations based on the content that's associated with the user Bob.

number of items. The content-based approach isn't widely used, but it does have certain advantages and can be used in combination with collaborative filtering to

improve the quality of the recommendations. Usually, production systems employ a combination of these techniques. Let's look at the concept of combining recommendation engines.

3.3 Recommending friends, articles, and news stories

In this section, we present a more realistic example that'll help us illustrate combining the techniques that we've discussed so far. We'll work with a hypothetical website whose purpose is to identify individuals with similar opinions, articles with similar comments, and news stories with similar content. Let's call our website MyDiggSpace.com. As the name suggests, the site would use the Digg API to retrieve the articles that you submitted through your Digg account (information about your Digg account could be provided upon registration). Then it would identify and present to you stories similar to the ones that you "dug." In addition, it would allow you to rate the stories that you read, so that in the future the system can sharpen its selection of recommended stories based on your feedback. As if that weren't enough, the site would present you with groups of common interest that you can join if you'd like, thus facilitating social interaction with similar minded individuals.

3.3.1 Introducing MyDiggSpace.com

Let's take the steps of building such a site one by one. True to our promise in the introduction, we won't address issues such as the design of the UI, persistence, and other important engineering components. To keep things interesting, we'll use the Digg API to retrieve data and make our example more realistic. First, we need to explain that Digg is a website (http://digg.com/) where users share content that they've discovered anywhere on the Web. The idea is that content isn't aggregated by editors who know what's best for you (or not), but from the users themselves. Whether the item that you want to talk about comes from a high-profile commercial news outlet or an obscure blog, Digg will let you post your selections and let the best content be revealed through the votes of the participating users.

The Digg API allows third parties to interact programmatically with Digg. Most of the data that lives in the Digg website is available through the API. You can get lists of stories based on popularity, time, or category (topic of discussion). We've written a set of wrapper classes that use the Digg API, and you can later extend them for your own purposes.

We'll build the dataset of MyDiggSpace.com by executing several simple steps. First, we'll collect the top stories from each category in Digg. This will create a list of users and a list of stories (items) for each user.

For each story of each user, we'll identify 10 stories that were submitted by other users, based on the content similarity between the stories. In other words, we'll create a content-based item-item recommendation engine and we'll find the top 10 similar stories.

To complete our dataset, we pretend that the users provide ratings for these stories and therefore we assign a random rating for each story. The assigned rating follows the same convention that we used in our earlier examples—the users whose names

start with the letters *A* through *D* assign ratings that are equal to either 4 or 5; the rest of the users assign ratings that are equal to 1, 2, or 3.

The purpose of this example is to introduce you to the concept of combining the results of different recommendation engines in order to get better results than any one engine alone could give you. This appears to be a wise practice across an area of applications that's much wider than recommendation engines. Later in the book, we'll talk about combining the results of classification engines. This example is the prelude to a broad and promising field. It also contains a bigger message that we want to convey in this book—the importance of the synergy of various elements of intelligence in delivering high-quality results for real applications.

3.3.2 Finding friends

Let's run our script, as shown in listing 3.15, and get in action with the hypothetical MyDiggSpace.com data.[2]

Listing 3.15 MyDiggSpace.com: an example of combining recommendation engines

```
BaseDataset ds = DiggData
➥ .loadDataFromDigg("C:/iWeb2/data/ch03/digg_stories.csv");   <-- Save data from Digg

// BaseDataset ds = DiggData
[CA].loadData("C:/iWeb2/data/ch03/digg_stories.csv");   <-- Or load local data

iweb2.ch3.collaborative.model.User user = ds.getUser(1);   <-- Pick user

DiggDelphi delphi = new DiggDelphi(ds);   <-- Create instance of recommender

delphi.findSimilarUsers(user);   <-- Find similar users

delphi.recommend(user);   <-- Recommend stories
```

Similar users could be presented on a side panel, for example, as the user is reviewing her stories. The recommended stories could also be presented in a special panel and, in order to improve our recommendations for each user, we could use a click-based approach similar to the one described in chapter 2. We could also offer the ability to rate each recommended story in order to achieve an even higher level of confidence in the user's preferences. We'll discuss these improvements in a bit, but first, let's look at the results that our script produced while we were writing the book.

We collected 146 items (stories) from 7 categories, for 33 users; you can control the number and the content of categories in the class `iweb2.ch3.content.digg.DiggCategory`. For these users, we've assigned 811 item ratings. For each user, the selection of items and the ratings are random, except that we follow the same convention that we used before in terms of clustering the ratings based on the initial letter of the username. The minimum number of ratings that a user has made on that set is 7, the maximum is 31, and the median is 26.

[2] Disclaimer: The data that the script enables you to collect is publicly available. Obviously, we can't be responsible for the content that may be retrieved when you run our example. Our goal is to provide a working example of using the Digg API and demonstrate how you can do something useful with it.

THE TRIANGULATION EFFECT

Figure 3.9 presents the set of similar users for the first user (adamfishercox) on our list, then the similar users for his most similar user (adrian67), then the similar users for a user who's similar to adrian67 (although not the most similar), whose username is DetroitThang1. An interesting observation can be made about the data in figure 3.9, which may or may not be obvious. User amipress is in the top five similar users of adamfishercox but isn't in the top five similar users of adrian67. And yet, amipress is in the top five similar users of DetroitThang1 with a similarity score 0.7, which is almost equal to the similarity score that we found between amipress and adamfishercox. Interesting, isn't it? We call this the *triangulation effect* and it shows us that there are *second-order effects* that can be leveraged and improve the accuracy—and thereby effectiveness—of our recommendations.

Let's further clarify this point by using the data from figure 3.9. The user adamfishercox is related to adrian67 by rank 1 and a similarity score equal to 1; the user amipress is related to adamfishercox by rank 2 and a similarity score (approximately)

```
bsh % delphi.findSimilarUsers(user);
Top Friends for user adamfishercox:

    name: adrian67       , similarity: 1.000000
    name: amipress       , similarity: 0.666667
    name: dvallone       , similarity: 0.500000
    name: cosmikdebris   , similarity: 0.500000
    name: cruelsommer    , similarity: 0.500000

bsh % iweb2.ch3.collaborative.model.User u2 =
ds.findUserByName("adrian67");

bsh % delphi.findSimilarUsers(u2);

Top Friends for user adrian67:

    name: adamfishercox  , similarity: 1.000000
    name: dvallone       , similarity: 1.000000
    name: ambermacbook   , similarity: 1.000000
    name: DetroitThang1  , similarity: 0.800000
    name: cruelsommer    , similarity: 0.750000

bsh % iweb2.ch3.collaborative.model.User u3 =
ds.findUserByName("DetroitThang1");

bsh % delphi.findSimilarUsers(u3);

Top Friends for user DetroitThang1:

    name: adrian67       , similarity: 0.800000
    name: cosmikdebris   , similarity: 0.750000
    name: amipress       , similarity: 0.700000
```

Figure 3.9 Finding similar users and the triangulation effect on a random Digg dataset

equal to 0.67. The rank of user amipress in relation to adrian67 is 7 and their similarity is equal to 0.57. We show these relationships graphically in figure 3.10, where adamfishercox is User 1, amipress is User 2, and adrian67 is User 3.

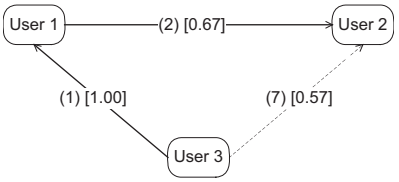

Figure 3.10 The triangulation effect and the opportunity for improvement of the relative ranking

The number inside the parentheses is the relative ranking, and the number inside the brackets is our similarity score; the base of the arrow refers to the user for whom we seek to find similar users. The arrow that connects User 3 with User 2 has a dotted line to depict the relationship that we can improve based on the information of the other relationships (arrows drawn with solid lines).

3.3.3 The inner workings of DiggDelphi

Now, let's look at the code that created these recommendations. Listing 3.16 presents the code from the class `DiggDelphi`.

Listing 3.16 Combining recommendation systems for the MyDiggSpace.com site

```
public class DiggDelphi {

 private Dataset ds;

 private Delphi delphiUC;
 private Delphi delphiUIC;
 private Delphi delphiUR;
 private Delphi delphiIR;

 private boolean verbose = true;

 public DiggDelphi(Dataset ds) {
    this.ds = ds;

    delphiUC  =
➥   new Delphi(ds,RecommendationType.USER_CONTENT_BASED);

    delphiUIC =
➥  new Delphi(ds,RecommendationType.USER_ITEM_CONTENT_BASED);

    delphiUR  = new Delphi(ds,RecommendationType.USER_BASED);

    delphiIR  = new Delphi(ds,RecommendationType.ITEM_BASED);
 }

public SimilarUser[] findSimilarUsers(User user, int topN) {      ❶

     List<SimilarUser> similarUsers =
➥   new ArrayList<SimilarUser>();

     similarUsers.addAll(
➥  Arrays.asList(delphiUC.findSimilarUsers(user, topN)));

     similarUsers.addAll(
➥   Arrays.asList(delphiUR.findSimilarUsers(user, topN)));
```

Initialize various recommendation engines

```
        return SimilarUser.getTopNFriends(similarUsers, topN);
    }

    public List<PredictedItemRating> recommend(User user, int topN) {      ❷
        List<PredictedItemRating> recommendations =
        new ArrayList<PredictedItemRating>();

        recommendations.addAll(delphiUIC.recommend(user, topN));
        recommendations.addAll(delphiUR.recommend(user, topN));
        recommendations.addAll(delphiIR.recommend(user, topN));

        return PredictedItemRating
         .getTopNRecommendations(recommendations, topN);
    }
}
```

We want to find similar users based on user-based and user-content-based similarities ❶ and recommend stories based on user-item-content-based, user-based, and item-based similarities ❷.

As you can see, in the method `findSimilarUsers`, we take the simplest approach of combining the lists of similar users—we add all the results in a list and sort the entries based on their similarity score (that happens inside the `getTopNFriends` method). We use the content-based approach, through the `delphiUC` instance, and the user-to-user similarity based on rankings approach (collaborative filtering), through the `delphiUR` instance. Note that the similarities between these two recommendation engines aren't in any way normalized. This means that the results will be a bit mixed up, even though we ordered them.

To understand this point better, think of a list that's made up of 20 bank accounts. If 10 of the accounts are in U.S. dollars and the other 10 are in euros, sorting a list that contains both of them based on their total amount won't make perfect sense unless we express them all in U.S. dollars or in euros. Nevertheless, the accounts that contain little money would still be at the bottom of the list, while the accounts that contain a lot of money would be at the top; the ordering just won't be exact.

Our analogy with the currencies, although illuminating, oversimplifies a major difference between the two cases. The normalization between currencies is well understood and straightforward. If I want to convert 100 U.S. dollars into 100 euros then I'd use the exchange rate between these two currencies to get the nominal value of 100 U.S. dollars into euros. In reality, if you want to get euros in your hands (or in your bank account), you have to pay the bank a commission fee, but your normalization formula is still extremely easy. Unfortunately, user similarities and recommendation scores aren't as easily susceptible to normalization. Combining recommendation engine scores is as much an art as it is a science. Ingenious heuristics are often used, and machine learning algorithms play an important role in creating an information processing layer on top of the initial recommendations.

Figure 3.11 shows the results of naïvely combining the recommendations from three different approaches, for the three users that we've examined so far. As shown in the method `recommend` of listing 3.16, we create a list that contains recommendations that

```
bsh % delphi.recommend(user);

Recommendations for user adamfishercox:

Item: Lumeneo Smera: French Concept of Car and MotorCycle,
predicted rating: 5.0
Item: Bill Gates to Congress: Let us hire more foreigners -
CNET N, predicted rating: 5.0
Item: The Best Tools for Visualization, predicted rating: 5.0
Item: Coolest Cubicle Contest, Part Three, predicted rating: 5.0
Item: Bush: Telecoms Should Be Thanked For Their Patriotic
Service, predicted rating: 5.0

bsh % delphi.recommend(u2);

Recommendations for user adrian67:

Item: Can women parallel park on Mars?, predicted rating: 5.0
Item: Coast Guard loses a few flares and ..., predicted rating:
5.0
Item: 10.5.2 released, predicted rating: 5.0
Item: They are all hot!, predicted rating: 5.0
Item: 11 Greatest Basketball Commercials Ever Made, predicted
rating: 5.0

bsh % delphi.recommend(u3);

Recommendations for user DetroitThang1:

Item: The Best Tools for Visualization, predicted rating: 5.0
Item: Coolest Cubicle Contest, Part Three, predicted rating:
5.000000
Item: Stink Films comes correct with 3 Adidas Original Films,
predicted rating: 5.0
Item: The Power Rangers Meet The Teenage Mutant Ninja Turtles,
predicted rating: 5.0
```

Figure 3.11 A sample of the results from the combination of three different recommendation engines

stem from a user-item content-based recommender, a user-user collaborative filtering recommender, and an item-item collaborative filtering recommender.

These are good results, in the sense that the recommended ratings are all fives as we'd expect due to our artificial bias on the ratings—the users whose names start with letters *A* through *D* always give a rating of 5 or 4. Remember that we said it's possible that the lack of normalization among the similarities is favoring one recommender over the others. We need a mechanism that will allow us to consider the recommendations of the various engines on an equal footing.

Look at the implementation of the `recommend` method shown in listing 3.17, which takes these concerns into consideration. The first step is to normalize all the predicted ratings, taking as reference the maximum predicted rating for the user across all recommendation engines. We also introduce an ad hoc threshold that eliminates

recommendations whose predicted ratings are below a certain value. Let this be your first exposure to the interesting subject of accounting for the cost of bad recommendations. In other words, our threshold value (however artificial) sets a barrier for the predicted ratings that our recommendations must exceed before they're seriously taken into consideration.

The last part of that implementation consists of averaging all the predicted ratings for a particular item in order to get a single predicted rating. This is a valid approach because we've normalized the ratings; without normalization, the averaging wouldn't make much sense. If a particular recommendation engine doesn't rate a particular item then the value of the rating would be zero, and therefore the particular item would be pushed further down in the list of recommendations. In other words, our approach combines averaging and voting between the predicted ratings of the recommenders. Once the combined score has been computed, the recommendations are added in a list and the results are sorted on the basis of the new predicted rating.

Listing 3.17 Improved implementation of recommending by combining recommenders

```
public List<PredictedItemRating> recommend(User user, int topN) {

    List<PredictedItemRating> recommendations =
➥  new ArrayList<PredictedItemRating>();

 double maxR=-1.0d;

 double maxRatingDelphiUIC =
➥ delphiUIC.getMaxPredictedRating(user.getId());

 double maxRatingDelphiUR  =
➥ delphiUR.getMaxPredictedRating(user.getId());

 double maxRatingDelphiIR  =
➥ delphiIR.getMaxPredictedRating(user.getId());

 double[] sortedMaxR =
➥ {maxRatingDelphiUIC, maxRatingDelphiUR, maxRatingDelphiIR};

 Arrays.sort(sortedMaxR);

 maxR = sortedMaxR[2];

 // auxiliary variable
 double scaledRating = 1.0d;

 // Recommender 1 -- User-to-Item content based
 double scaling = maxR/maxRatingDelphiUIC;

 //Set an ad hoc threshold and scale it
 double scaledThreshold = 0.5 * scaling;

 List<PredictedItemRating> uicList =
➥ new ArrayList<PredictedItemRating>(topN);

 uicList = delphiUIC.recommend(user, topN);

 for (PredictedItemRating pR : uicList) {

  scaledRating = pR.getRating(6) * scaling;
```

Max predicted ratings by recommender

Max predicted rating across recommenders

maxR is max predicted rating

Create scaling factor for each engine

Get recommendations from each engine

```
    if (scaledRating < scaledThreshold) {
      uicList.remove(pR);
    } else {
      pR.setRating(scaledRating);
    }
  }

  // Recommender 2 -- User based collaborative filtering
  scaling = maxR/maxRatingDelphiUR;

  scaledThreshold = 0.5 * scaling;

  List<PredictedItemRating> urList =
➥ new ArrayList<PredictedItemRating>(topN);

urList = delphiUR.recommend(user, topN);

  for (PredictedItemRating pR : urList) {

    scaledRating = pR.getRating(6) * scaling;

          if (scaledRating < scaledThreshold) {
             urList.remove(pR);
          } else {
             pR.setRating(scaledRating);
          }
    }

    // Recommender 3 -- Item based collaborative filtering
    scaling = maxR/maxRatingDelphiIR;

    scaledThreshold = 0.5 * scaling;

    List<PredictedItemRating> irList =
➥ new ArrayList<PredictedItemRating>(topN);

    irList = delphiIR.recommend(user, topN);

    for (PredictedItemRating pR : irList) {

         scaledRating = pR.getRating(6) * scaling;

         if (scaledRating < scaledThreshold) {
             irList.remove(pR);
         } else {
             pR.setRating(scaledRating);
         }
    }

    double urRating=0;
    double irRating=0;
    double vote=0;

    for (PredictedItemRating uic : uicList) {

        //Initialize
        urRating=0; irRating=0; vote=0;

       for (PredictedItemRating ur : urList) {
         if (uic.getItemId() == ur.getItemId()) {
           urRating = ur.getRating(6);
         }
```

Scaled rating should be above threshold

Create scaling factor for each engine

Get recommendations from each engine

Get average value and scale properly

```
      }

      for (PredictedItemRating ir : irList) {
        if (uic.getItemId() == ir.getItemId()) {
         irRating = ir.getRating(6);
        }
      }

      vote = (uic.getRating(6)+urRating+irRating)/3.0d;

     recommendations.add(
➥ new PredictedItemRating(user.getId(), uic.getItemId(), vote));
    }

    rescale(recommendations,maxR);

    return PredictedItemRating
➥ .getTopNRecommendations(recommendations, topN);
}
```

You can further improve your recommendations by targeting the preferences of each individual user on MyDiggSpace.com by combining the results obtained in the `DiggDelphi` class and the `NaiveBayes` classifier that we encountered in chapter 2. For more details on this approach, see the to-do list at the end of this chapter. Any learning mechanism (a number of them are presented in chapter 5) as well as optimization techniques can be employed to enhance the results of the base recommenders. This approach of combining techniques with an encapsulating learning layer is gaining popularity and support from both industry leaders and academics (see also chapter 6).

You should, by now, have a good idea about combining recommendation systems and the interplay of their capabilities in identifying friends and interesting articles for the users of your web application. The next section will focus on a different example: the recommendation of movies on a site such as Netflix. The main characteristic of such examples is the large size of their datasets.

3.4 Recommending movies on a site such as Netflix.com

In the introduction, we talked about Netflix, Inc., the world's largest online movie rental service, offering more than 7 million subscribers access to 90,000 DVD titles plus a growing library of more than 5,000 full-length movies and television episodes available for instant watching on their PCs. If you recall, part of Netflix's online success is its ability to provide users with an easy way to choose movies from an expansive selection of titles. At the core of that ability is a recommendation system called Cinematch. Its job is to predict whether someone will enjoy a movie based on how much he liked or disliked other movies.

3.4.1 An introduction of movie datasets and recommenders

In this section, we'll describe a recommendation system whose goal is the same as that of Cinematch. We'll work with publicly available data from the MovieLens project. The MovieLens project is a free service provided by the GroupLens research lab at the University of Minnesota. The project hosts a website that offers movie recommendations.

You can try it out at http://www.movielens.org/quickpick. There are two MovieLens datasets available on the website of the GroupLens lab.

The first dataset[3] consists of 100,000 ratings by 943 users for 1,682 movies. The second dataset[4] has one million ratings by 6,040 users for 3,900 movies. The first dataset is provided with the distribution of this book; please make sure that you read the license and terms of use. The format of the data is different between the two datasets. We find the format of the second (1M ratings) dataset more appropriate and convenient; it contains just three files, movies.dat, ratings.dat, and users.dat. However, we want to use the smaller dataset for efficiency. So, we've transformed the original format of the small dataset (100K ratings) into the format of the larger dataset, for convenience. The original data and the large dataset can be retrieved from the GroupLens website. You should extract the data inside the `C:/iWeb2/data/ch03/MovieLens/` directory; if you don't then, in listing 3.18, you should alter the `createDataset` method so that it takes the path of the data directory as an argument.

Large recommendation systems such as those of Netflix and Amazon.com rely heavily on item-based collaborative filtering (see Linden, Smith, and York). This approach, which we described in sections 3.2.1 and 3.2.2, is improved by three major components.

The first is *data normalization*. This is a fancy term for something that's intuitively easy to grasp. If a user tends to rate all movies with a high score (a rating pattern that we adopted for our artificial rating of items in the earlier sections) it makes sense to consider the relative ratings of the user as opposed to their absolute values.

The second major component is the *neighbor selection*. In collaborative filtering, we identify a set of items (or users) whose ratings we'll use to infer the rating of nonrated items. So naturally, two questions arise from this mandate: how many neighbors do we need? How do we choose the "best" neighbors—the neighbors that will provide the most accurate prediction of a rating?

The third major component of collaborative filtering is determining the *neighbor weights*—how important is the rating of each neighbor? Bell and Koren showed that data normalization and neighbor weight selection are the two most important components in improving the accuracy of the collaborative filtering approach.

Let's begin by describing our Bean Shell script for this example. Listing 3.18 demonstrates how to load the data, create an instance of our recommender (called `MovieLensDelphi`), pick users, and get recommendations for each one of them.

Listing 3.18 MovieLensDelphi: Recommendations for the MovieLens datasets

```
MovieLensDataset ds = MovieLensData.createDataset();      <-- Load MovieLens dataset

MovieLensDelphi delphi = new MovieLensDelphi(ds);         <-- Create recommender

iweb2.ch3.collaborative.model.User u1 = ds.getUser(1);    <-- Pick users and create
delphi.recommend(u1);                                         recommendations
```

[3] The URL for the original data is http://www.grouplens.org/system/files/ml-data.tar__0.gz

[4] The URL for the original data is http://www.grouplens.org/system/files/million-ml-data.tar__0.gz

```
iweb2.ch3.collaborative.model.User u155 = ds.getUser(155);
delphi.recommend(u155);

iweb2.ch3.collaborative.model.User u876 = ds.getUser(876);
delphi.recommend(u876);
```

The first user could've been any user, so we picked the user whose ID is equal to 1. The other two users were identified by executing the command `Delphi.findSimilarUsers(u1);`. We did this so that we can quickly check whether our recommendations make sense. It's reasonable to expect that if two users are similar and neither has seen a movie, then if a movie is recommended to one of them, there's a good chance that it'll be recommended to the other user too. Figure 3.12 shows the results that we get when we run the script and corroborates this sanity check.

These datasets aren't as large as the ones that can be found in the Amazon.com or the Netflix applications, but they're certainly much larger than everything else that

```
bsh % iweb2.ch3.collaborative.model.User u1 = ds.getUser(1);
bsh % delphi.recommend(u1);

Recommendations for user 1:

Item: Yojimbo (1961)              , predicted rating: 5.000000
Item: Loves of Carmen, The (1948) , predicted rating: 4.303400
Item: Voyage to
the Beginning of the World (1997) , predicted rating: 4.303400
Item: Baby, The (1973)            , predicted rating: 4.303400
Item: Cat from Outer Space,
The (1978)                        , predicted rating: 4.123200

bsh % iweb2.ch3.collaborative.model.User u155 = ds.getUser(155);
bsh % delphi.recommend(u155);

Recommendations for user 155:

Item: Persuasion (1995)              , predicted rating: 5.000000
Item: Close Shave, A (1995)          , predicted rating: 4.373000
Item: Notorious (1946)               , predicted rating: 4.181900
Item: Shadow of a Doubt (1943)       , predicted rating: 4.101800
Item: Crimes and Misdemeanors (1989) , predicted rating: 4.061700

bsh % iweb2.ch3.collaborative.model.User u876 = ds.getUser(876);
bsh % delphi.recommend(u876);

Recommendations for user 876:

Item: Third Man, The (1949)      , predicted rating: 5.000000
Item: Bicycle Thief,
The (Ladri di biciclette)(1948)  , predicted rating: 4.841200
Item: Thin Blue Line, The (1988) , predicted rating: 4.685600
Item: Loves of Carmen, The (1948), predicted rating: 4.600200
Item: Heaven's Burning (1997)    , predicted rating: 4.600200
```

Figure 3.12 Recommendations from the MovieLensDelphi recommender based on the MovieLens dataset

we've presented so far, and large enough to be realistic. Running the script for the small MovieLens dataset (100K ratings) will take anywhere between 30 seconds to a minute simply to create the recommender. During that time, the recommender does a lot of processing, as we'll see. The recommendations themselves are relatively fast, typically under one second.

3.4.2 Data normalization and correlation coefficients

As promised, in the example for this section, we enriched our collaborative filtering approach by introducing two new tools. The first is data normalization and the second a new similarity measure for capturing the correlation between items. The new similarity measure is called the *linear correlation coefficient* (also known as the *product-moment correlation coefficient*, or *Pearson's r*). Calculating that coefficient for two arrays *x* and *y* is fairly straightforward. Listing 3.19 shows the three methods responsible for that calculation.

Listing 3.19 The calculation of the linear correlation coefficient (Pearson's r)

```
public double calculate() {

   if( n == 0) {
      return 0.0;
   }
   double rho=0.0d;
   double avgX = getAverage(x);
   double avgY = getAverage(y);

   double sX = getStdDev(avgX,x);
   double sY = getStdDev(avgY,y);

   double xy=0;

   for (int i=0; i < n; i++) {

      xy += (x[i]-avgX)*(y[i]-avgY);        ❶
   }

    if( sX == ZERO || sY == ZERO) {         ❷

      double indX = ZERO;
      double indY = ZERO;

    for (int i=1; i < n; i++) {

      indX += (x[0]-x[i]);
      indY += (y[0]-y[i]);
    }

       if (indX == ZERO && indY == ZERO) {
      // All points refer to the same value
      // This is a degenerate case of correlation
      return 1.0;
    } else {
      //Either the values of the X vary or the values of Y
      if (sX == ZERO) {
         sX = sY;
```

Calculate average values for each vector

Calculate standard deviations for each vector

```
            } else {
            sY = sX;
            }
          }
        }
        rho = xy / ((double)n*(sX*sY));
        return rho;
    }
    private double getAverage(double[] v) {
       double avg=0;
       for (double xi : v ) {
          avg += xi;
       }
       return (avg/(double)v.length);
    }
    private double getStdDev(double m, double[] v) {
       double sigma=0;
       for (double xi : v ) {
            sigma += (xi - m)*(xi - m);
       }
       return Math.sqrt(sigma / (double)v.length);
    }
```

The value of Pearson's r

❶ is the cross product calculation of the pointwise deviations from the mean value. ❷ is a special (singular) case, where all the points have the exact same values for either *X* or *Y*, or both. This case must be treated separately because it leads to division by zero.

The method `getAverage` is self-explanatory; it calculates the average of the vector that's provided as an argument. The `getStdDev` method calculates the *standard deviation* for the data of the vector that's passed as the second argument; the first argument of the method ought to be the average. There's a smarter way to do this that avoids a plague of numerical calculations called the *roundoff error*; read the article on the corrected two-pass algorithm by Chan, Golub, and LeVeque.

Calculating similarity based on Pearson's correlation is a widely used metric that has the following properties:

- Whenever it's equal to zero, the two items are (statistically) *uncorrelated*.
- Whenever it's equal to 1, the ratings of the two items fit exactly onto a straight line with positive slope; for example, (1,2), (3,4), (4,5), (4,5), where the first number in parentheses denotes the rating of the first item while the second number denotes the rating of the second item. This is called *complete positive correlation*. In other words, if we know the ratings of one item, we can infer the ratings of the other with high probability.
- Whenever it's equal to -1, the ratings of the two items fit exactly onto a straight line but with negative slope; for example (1,5), (2,4), (3,3), (4,2). This is called *complete negative correlation*. In this case too, we can infer the ratings of one item based on those of the other item, but now whenever the ratings for the first item increase, the ratings for the second item will decrease.

If the items are correlated linearly, the linear correlation coefficient is a good measure for the strength of that correlation. In fact, if you fit a straight line to your dataset then the linear correlation coefficient reflects the extent to which your ratings lie away from that line. But not everything fits that rosy picture. Unfortunately, this metric is a rather poor measure of correlation if no correlation exists! Say what? Yes, that's right.

A celebrated counterexample is known as the *Anscombe's quartet*. Figure 3.13 depicts Anscombe's quartet for four different pairs of values; this plot is available on Wikipedia, in SVG format, at http://en.wikipedia.org/wiki/Image:Anscombe.svg.

In plain terms, if you plot the ratings between two items against each other, and the plot is similar to the upper-left graph of figure 3.13, the linear correlation coefficient is a meaningful metric. In the other graphs, Pearson's correlation has the same value but its significance is questionable; the datasets are carefully crafted so that they also have the same mean, the same standard deviation, and the same linear fit ($y = 3 + 0.5*x$). This inability to determine the significance of the linear (Pearson) correlation coefficient led people to a different kind of similarity metric called *nonparametric correlation*. There are two popular nonparametric correlation coefficients: the Spearman rank-order correlation coefficient (rs) and the Kendall's tau (τ). These metrics trade some loss of information for the assurance that a detected correlation is truly present in the data when the values of the metrics indicate so. We discuss nonparametric correlation in the to-do

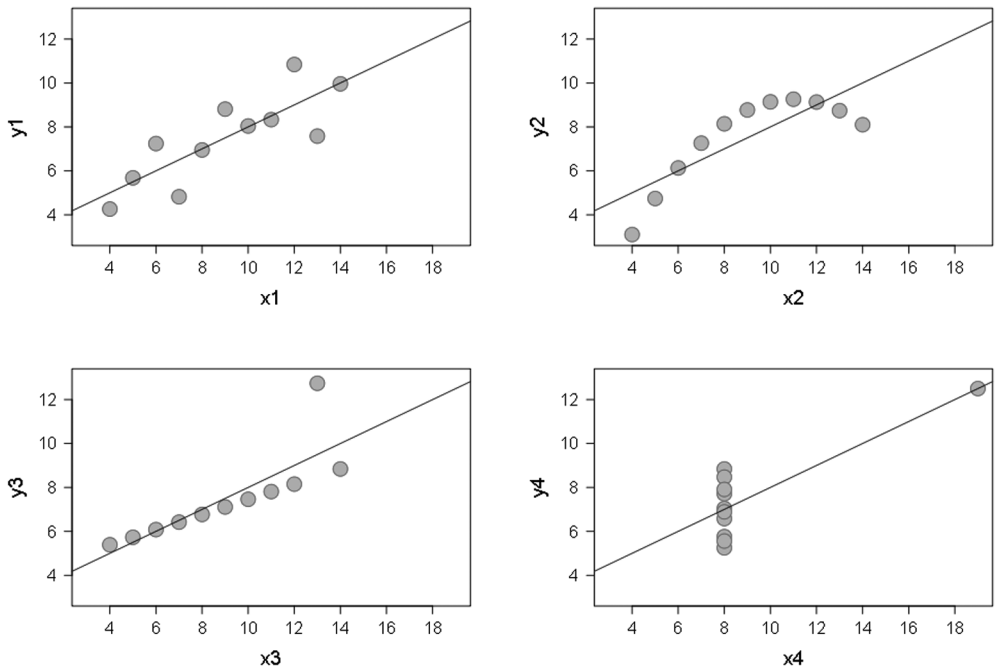

Figure 3.13 Anscombe's quartet: Four datasets that have the same Pearson's correlation but different distributions

section because in many cases the distribution of ratings will look like the graph in the lower-right corner. Nevertheless, from now on, we'll assume that whenever the item ratings are correlated, they're linearly correlated and we can safely use Pearson's correlation. You can find more information about the nonparametric correlations in the references section.

Having discussed the new possibilities that the linear coefficient (Pearson's r) and the nonparametric correlations offer for evaluating similarities, we'll proceed by showing you one way of achieving data normalization. Listing 3.20 shows code that does just that; it's one of the constructors for the class `PearsonCorrelation`. The first argument provides a reference to the original dataset, and the other two are references to the items whose correlation we want to calculate. As you can see, the arrays that are constructed for calculating the Pearson correlation don't refer to the ratings of each user, as they were recorded, but rather to a new set of data in which we've subtracted the average rating of an item from the user's ratings. Clearly, this isn't the only way of achieving data normalization. Bell and Koren describe sophisticated data normalization techniques as applied to the Netflix prize dataset.

Listing 3.20 Data normalization around the average rating of items

```
public PearsonCorrelation(Dataset ds, Item iA, Item iB) {

  double aAvgR = iA.getAverageRating();
  double bAvgR = iB.getAverageRating();

  Integer[] uid = Item.getSharedUserIds(iA, iB);

  n = uid.length;

  x = new double[n];
  y = new double[n];

  User u;

  double urA=0;
  double urB=0;

  for (int i=0; i<n; i++) {

    u = ds.getUser(uid[i]);

    urA = (double) u.getItemRating(iA.getId()).getRating();
    urB = (double) u.getItemRating(iB.getId()).getRating();

    x[i] = urA - aAvgR;
    y[i] = urB - bAvgR;
  }
}
```

Data normalization and the use of Pearson's correlation are incorporated in the `PearsonCorrelation` class, and their use is encapsulated by the `MovieLensItemSimilarity` class. For that reason, the `MovieLensDelphi` class is slightly different from the other `Delphi`-type classes. The code in listing 3.21 highlights these differences.

Listing 3.21 Calculation of a rating involves data renormalization and rescaling

```
private double estimateItemBasedRating(User user, Item item) {

 double itemRating = item.getAverageRating();

 int itemId = item.getId();
 int userId = user.getId();

 double itemAvgRating = item.getAverageRating();
 double weightedDeltaSum = 0.0;

 int sumN=0;

 // check if the user has already rated the item
 Rating existingRatingByUser = user.getItemRating(item.getId());

 if (existingRatingByUser != null) {

  itemRating = existingRatingByUser.getRating();

 } else {

       double similarityBetweenItems = 0;

       double weightedDelta = 0;
       double delta = 0;

       for (Item anotherItem : dataSet.getItems()) {

    // only consider items that were rated by the user
      Rating anotherItemRating =
➥  anotherItem.getUserRating(userId);

      if (anotherItemRating != null) {

         delta = itemAvgRating - anotherItemRating.getRating();

       similarityBetweenItems =
➥  itemSimilarityMatrix.getValue(itemId, anotherItem.getId());

        if (Math.abs(similarityBetweenItems) >
➥  similarityThreshold) {                                    ❶

         weightedDelta = similarityBetweenItems * delta;

         weightedDeltaSum += weightedDelta;

         sumN++;
         }
      }
     }

     if (sumN > 0) {
      itemRating = itemAvgRating -
➥  (weightedDeltaSum/(double) sumN)                          ❷
     }
  }

  return itemRating;
}

public List<PredictedItemRating> getTopNRecommendations(
➥ List<PredictedItemRating> recommendations, int topN) {
```

Iterate through all items

Perform data renormalization

Get similarity between two items

```
  PredictedItemRating.sort(recommendations);

  double maxR = recommendations.get(0).getRating();
  double scaledR;

  List<PredictedItemRating> topRecommendations =
➥ new ArrayList<PredictedItemRating>();

   for(PredictedItemRating r : recommendations) {

    if( topRecommendations.size() >= topN ) {
     // have enough recommendations.
     break;
    }

    scaledR = r.getRating() * (5/maxR);
    r.setRating(scaledR);

    topRecommendations.add(r);
  }

  return topRecommendations;
}
```

We weigh the deviation ❶ from the mean value based on the similarity of the two items and assign ❷ a rating based on the item's mean value and the sum of weighted deviations.

Data renormalization refers to the fact that our similarities were built around the item's average rating, so in order to calculate the predicted item rating, we need to renormalize from differences (delta) to actual ratings. One drawback of this kind of data normalization is that the maximum value of the predicted rating can fall outside the range of the acceptable values. Thus, a rescaling of the predicted ratings is required, as shown inside the method `getTopNRecommendations`.

3.5 Large-scale implementation and evaluation issues

Commercial recommendation systems operate under demanding conditions. The number of users is typically on the order of millions, and the number of items on the order of hundreds of thousands. An additional requirement is the capability to provide recommendations in real-time (typically, subsecond response times) without sacrificing the quality of the recommendations. As we've seen, by accumulating ratings from each user, it's possible to enhance the accuracy of our predictions over time. But in real life, it's imperative that we give excellent recommendations to new users for which, by definition, we don't have a lot of ratings. Another stringent requirement for state-of-the-art recommendation systems is the ability to update their predictions based on incoming ratings. In large commercial sites, there may be thousands of ratings and purchases that take place in a few hours, and perhaps tens of thousands in the course of a single day. The ability to update the recommendation system with that additional information is important and must happen online—without downtime.

Let's say that you wrote a recommender and you're satisfied with its speed and the amount of data that it can handle. Is this a good recommender? It's not useful to

have a fast and scalable recommender that produces bad recommendations! So, let's talk about evaluating the accuracy of a recommendation system. If you search the related literature, you'll find that there are dozens of quantitative metrics and several qualitative methods for evaluating the results of recommendation systems. The plethora of metrics and methods reflects the challenges of conducting a meaningful, fair, and accurate evaluation for recommendations. The review article by Herlocker, Konstan, Terveen, and Riedl contains a wealth of information if you're interested in this topic.

We've written a class that evaluates our recommendations on the MovieLens data by calculating the *root mean square error (RMSE)* of the predicted ratings. The RMSE is a simple but robust technique of evaluating the accuracy of your recommendations. This metric has two main features: (1) it always increases (you don't get kudos for predicting a rating accurately) and (2) by taking the square of the differences, the large differences (>1) are amplified, and it doesn't matter if you undershoot or you overshoot the rating.

We can argue that the RMSE is probably too naïve. Let's consider two cases. In the first case, we recommend to a user a movie with four stars and he really doesn't like it (he'd rate it two stars); in the second case, we recommend a movie with three stars but the user loves it (he'd rate it five stars). In both cases, the contribution to the RMSE is the same, but it's likely that the user's dissatisfaction would probably be larger in the first case than in the second case; we know that our dissatisfaction would be!

You can find the code that calculates the RMSE in the class `RMSEEstimator`. Listing 3.22 shows you how you can evaluate the accuracy of our `MovieLensDelphi` recommender.

Listing 3.22 Calculating the root mean squared error for a recommender

```
MovieLensDataset ds = MovieLensData.createDataset(100000);

MovieLensDelphi delphi = new MovieLensDelphi(ds);

RMSEEstimator rmseEstimator = new RMSEEstimator();

rmseEstimator.calculateRMSE(delphi);
```

❶ **Create the dataset but reserve 100,000 ratings for testing**

We create a dataset that excludes 100K ratings from the one million ratings that are available in the large MovieLens dataset ❶. The recommender will train on the remaining 900K ratings and be evaluated on the 100K ratings; the rest of the script is self-explanatory. If you run this with the code that we've described in this section then your RMSE should be equal to 1.0256. This isn't a bad RMSE but it's not very good either. We highly recommend that you improve on that result and set as your goal an RMSE that's below 1. As a relative measure of success, we should mention that the best teams that compete for the Netflix prize have an RMSE that is between 0.86 and 0.88. So, even though the dataset is different, don't be disappointed if your improvements bring your RMSE to be approximately equal to 0.9—it would be a great success for you and for us!

3.6 Summary

In this chapter, you've learned about the concepts of distance and similarity between users and items. We've seen that one size doesn't fit all, and we need to be careful in our selection of a similarity metric. Throughout the chapter we encountered several metrics: the Jaccard metric, the Pearson correlation, and variants of these metrics that we introduced. Similarity formulas must produce results that are consistent with a few basic rules, but otherwise we're free to choose the ones that produce the best results for our purposes.

We discussed the two broad categories of techniques for creating recommendations—collaborative filtering and the content-based approach. We walked through the construction of an online music store that demonstrated the underlying principles, in detail but with clarity. In the process of building these examples, we've created the infrastructure that you need for writing a general recommendation system for your own application.

Finally, we tackled two more general examples. The first example was a hypothetical website that used the Digg API and retrieved the content of our users for further analysis of similarity between them, and in order to provide unseen article recommendations to them. In this example, we pointed out the existence of second-order effects, and by extension of *higher-order effects*, and we suggested a way to leverage them in order to improve the accuracy of our recommendations. Our second example dealt with movie recommendations and introduced the concept of data normalization, as well as the popular linear (Pearson) correlation coefficient. In the latter context, we also introduced a class that evaluates the accuracy of our recommendations based on the root mean squared error.

In both examples, we demonstrated that as the complexity and the size of the problem increase, it becomes imperative to leverage the combination of techniques for improving the efficiency and quality of our recommendations. Thus, we discussed the possibility of reusing what you learning from user clicks in the example of MyDiggSpace.com. This is a theme that we'll encounter throughout this book—the combination of techniques that capture different aspects of our problem can, and often does, result in recommendations of higher accuracy.

In the next chapter, we'll encounter another family of intelligent algorithms: clustering algorithms. Nevertheless, if you haven't worked on the to-do topics yet then you might want to have a look at them now, while all the recommendation related material still reverberates in your mind.

3.7 To Do

1. *Similarity metrics.* Implement the Jaccard similarity for the `MusicUsers`. What differences do you observe? A variation of the Jaccard metric is the *Tanimoto metric*, which is more appropriate for continuous values. The Tanimoto metric is equal to the ratio of the intersection of two sets ($Ni = |X \cap Y|$) over the union ($Nu = |X| + |Y|$) minus the intersection—$T = Ni/(Nu-Ni)$.

For example, if *X* = {baseball, basketball, volleyball, tennis, golf} and *Y* = {baseball, basketball, cricket, running} then the Tanimoto metric has a value equal to *2/((5+4)–2)*, which is approximately equal to 0.2857. Work out the formula in the case of vectors (Java arrays `double[] x` and `double[] y`). Hint: the intersection corresponds to the inner product of the two vectors and the union to the sum of their magnitudes.

Another interesting similarity measure is the *city block* metric. Its name stems from the fact that the values of the vectors, *X* and *Y*, are assumed to be coordinates on a multidimensional orthogonal grid. When the vectors are two-dimensional, it resembles the way that a taxi driver would give you instructions in a city: "the Empire State Building is two blocks south and three blocks east from here." If you like that metric or want to study the cases where it's most applicable, *Taxicab Geometry: An Adventure in Non-Euclidean Geometry* by Eugene F. Krause provides a detailed exposition.

2 *Varying the range of prediction.* Did you ever wonder why various websites want you to rate movies, songs, and other products by assigning one integer value between 1 and 5 (inclusive)? Why not pick a value between 1 and 10? Or even between 1 and 100? Wouldn't that give you more flexibility to express the degree of your satisfaction with the product? To take this one step further, why not rate different aspects about a product? In the case of a movie, we could rate the plot, the performance of the actors, the soundtrack, and the visual effects. You can extend the code that we presented in this chapter and experiment along these lines. Can you identify any potential issues?

3 *Improving recommendations through ensemble methods.* A technique that's becoming increasingly popular consists of combining independent techniques in order to improve the combined recommendation accuracy. There are many good theoretical reasons for pursuing ensemble methods; if you're interested in that topic, you could read the article by Dietterich. In addition to theory, there's empirical evidence that ensemble methods may produce better results than individual techniques. Bell and Korren are leading the Netflix prize competition (at the time of this writing), and their assessment was the following: "We found no perfect model. Instead, our best results came from combining predictions of models that complemented each other."

How about combining some of the recommenders that we've given you in this chapter, as well as those that you may invent, and comparing their results to the results of each individual recommender? If the results are better, your "soup" worked! If not, investigate what recommenders you used and to what extent they capture a different aspect of the problem.

4 *Minimizing the roundoff error.* As you may know, the typical numerical types in Java and most other languages store the values with finite precision. The representation of an `integer` or `long` number is exact, even though the range of their values is finite and determined by the number of bits associated with each type.

But enter floating-point arithmetic (`float` and `double`) and a number of issues crop up due to the inexactness of the numerical representations. At best, you don't have to worry about them, and at worst, you can use `double` throughout.

Nevertheless, in intelligent applications, the heavy use of numerical calculations requires that you be aware of the implications that the finite precision of real numbers has on the result of computations, especially the results that are produced as a result of accumulations or multiplications with very small or large numbers. Let's consider the roundoff error mentioned in the evaluation of the standard deviation of the class `PearsonCorrelation`. The smallest floating-point number that gives a result other than 1.0, when added to 1.0, is called the *machine accuracy* (ϵ). Nearly every arithmetic operation between floating numbers introduces a fractional error on the order of magnitude of ϵ. That error is called the *roundoff error*.

Read the article on the corrected two-pass algorithm of Chan, Golub, and LeVeque, and implement the computation of the standard deviation accordingly. You can also find a brief description of this algorithm in the monumental *Numerical Recipes: The Art of Scientific Computing*. Do you see a perceptible difference in the outcome? What do you think will happen if you use sets that are even larger than the ones considered in this book? Note that the main points of the algorithm apply equally well in the computation of the RMSE that we conducted for evaluating the accuracy of our recommendations.

5 *Nonparametric or rank correlation.* Correlations that belong in this category are useful if you have reason to question the validity of the linearity assumption underlying the Pearson correlation metric. You can create new similarity classes based on this type of metric, which trade off some information about the data for an assurance about the presence of a true correlation between two sets of data—in our case, two sets of ratings. The main idea behind nonparametric correlation is substituting the values of a variable with the rank of that value in the dataset. The best-known nonparametric correlation coefficients are the *Spearman rank-order correlation coefficient (rs)* and the *Kendall's tau* (τ). You can read all about these coefficients in the masterly written book *Numerical Recipes: The Art of Scientific Computing*.

In the case of movie ratings from 1 to 5, you'll get a lot of conflicts in the rank of values; for example, there will be a lot of movies whose value will be exactly 4. But this presents an opportunity to be creative about using these correlations. What if you use the time of the rating to break the tie of the values? Implement such an approach and compare with the results that you get from using the plain vanilla Pearson's correlation.

3.8 References

Bell, R.M., and Y. Koren. "Scalable Collaborative Filtering with Jointly Derived Neighborhood Interpolation Weights." IEEE International Conference on Data Mining (ICDM'07), 2007. http://www.research.att.com/~yehuda/pubs/BellKorIcdm07.pdf.

Chan, T.F., G.H. Golub, and R.J. LeVeque. "Algorithms for computing the sample variance: Algorithms and recommendations." *American Statistician*, vol. 37, pp. 242-247, 1983.

Dietterich, T.G., "Ensemble methods in machine learning." *Multiple Classifier Systems*, (Editors: J. Kittler and F.Roli) volume 1857 of *Lecture Notes in Computer Science*, Cagliari, Italy. Springer, pp.1-15, 2000. http://citeseer.ist.psu.edu/dietterich00ensemble.html.

Estes, W.K. *Classification and Cognition*. Oxford University Press, 1996.

Herlocker, J.L., J.A. Konstan, L.G. Terveen, and J.T. Riedl (2004). "Evaluating Collaborative Filtering Recommender Systems." *ACM Transactions on Information Systems*, Vol 22, 5-53. ACM Press, 2004. http://web.engr.oregonstate.edu/~herlock/papers/eval_tois.pdf.

James, W. *The Principles of Psychology*. Henry Holt and Company, 1918.

Krause, E.F. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, Inc. 1986.

Linden, G., B. Smith, and J. York. "Amazon.com recommendations: Item-to-item collaborative filtering." *IEEE Internet Computing*, January-February 2003, pp.76-80.

Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes: The Art of Scientific Computing* (3rd Edition). Cambridge University Press, 1997.

JAVA/WEB

Algorithms *of the* Intelligent Web

Haralambos Marmanis • Dmitry Babenko

An algorithm is a sequence of steps that solves a problem. *Algorithms of the Intelligent Web* provides exactly that—explicit, clearly organized patterns to implement valuable web application features like recommendation engines, smart searching, content organizers, and much more. With these techniques you'll capture vital raw information about your users and profitably transform it into action.

Algorithms of the Intelligent Web is a handbook for web developers who want to exploit relationships in user data that can't be discovered manually. The book presents crystal-clear explanations of techniques you can apply immediately. It is based on the authors' practical experience as web developers and their deep expertise in the science of machine learning. With a wealth of detailed, Java-based examples this book shows you how to build applications that behave intelligently and learn from your users' actions.

What's Inside

- Create recommendations like Netflix or Amazon
- Implement Google's PageRank algorithm
- Discover matches on social-networking sites
- Organize your news group discussions
- Select topics of interest from shared bookmarks
- Filter spam and categorize emails based on content

Dr. Haralambos (Babis) Marmanis is a pioneer in the adoption of machine learning techniques for industrial solutions, and also a world expert in supply management. **Dmitry Babenko** has designed applications and infrastructure for banking, insurance, supply-chain management, and business intelligence companies.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/AlgorithmsoftheIntelligentWeb

"Unequivocally outstanding—this is the best technical book I have read all year."
—Robert Hanson
Quality Technology Services

"You don't need a PhD to build an intelligent website—pick up this book instead."
—Ajay Bhandari, FoodieBytes.com

"Very useful ... will bring you up to speed quickly."
—Sumit Pal, LeapFrogrx

"Excellent ... a perfect blend of theory and practice."
—Carlton Gibson
Noumenal Software

"Unlock the future of the web by analyzing what we know today!"
—Eric Swanson, AAA

ISBN 13: 978-1-933988-66-5
ISBN 10: 1-933988-66-5
54499
9 781933 988665

MANNING $44.99 / Can $56.99 [INCLUDING eBOOK]