Recommender Systems

9

9.1 Introduction

In this chapter, we will see what are recommender systems, how they work, and how they can be implemented. We will also see the different paradigms of recommender systems based on the information they use, as well as the output they produce. We will consider typical questions that companies like Netflix or Amazon include in their products: Which movie should I rent? Which TV should I buy? and we will give some insights in order to deal with more complex questions: Which is the best place for me and my family to travel to?

So, the first question we should answer: What is a recommender system? It can be defined as a tool designed to interact with large and complex information spaces, and to provide information or items that are likely to be of interest to the user, in an automated fashion. We refer to complex information space to the set of items, and its characteristics, which the system recommends to the user, i.e., books, movies, or city trips.

Nowadays, recommender systems are extremely common, and are applied in a large variety of applications. Perhaps one of the most popular types are the movie recommender systems in applications used by companies such as Netflix, and the music recommenders in Pandora or Spotify, as well as any kind of product recommendation from Amazon.com. However, the truth is that recommender systems are present in a huge variety of applications, such as movies, music, news, books, research papers, search queries, social tags, and products in general, but they are also present in more sophisticated products where personalization is critical, like recommender systems for restaurants, financial services, life assurance, online dating, and Twitter followers.

Why and When Do We Need a Recommender System?

In this new era, where the quantity of information is huge, recommender systems are extremely useful in several domains. People are not able to be experts in all

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_9

these domains in which they are users, and they do not have enough time to spend looking for the perfect TV or book to buy. Particularly, recommender systems are really interesting when dealing with the following issues:

- solutions for large amounts of good data;
- reduction of cognitive load on the user;
- allowing new items to be revealed to users.

9.2 How Do Recommender Systems Work?

There are several different ways to build a recommender system. However, most of them take one of two basic approaches: *content-based filtering* (CBF) or *collaborative filtering* (CF).

9.2.1 Content-Based Filtering

CBF methods are constructed behind the following paradigm: "Show me more of the same what I've liked". So, this approach will recommend items which are similar to those the user liked before and the recommendations are based on descriptions of items and a profile of the user's preferences. The computation of the similarity between items is the most important part of these methods and it is based on the content of the items themselves. As the content of the item can be very diverse, and it usually depends on the kind of items the system recommends, a range of sophisticated algorithms are usually used to abstract features from items. When dealing with textual information such as books or news, a widely used algorithm is *tf–idf* representation. The term tf–idf refers to frequency–inverse document frequency, it is a numerical statistic that measures how important a word is to a document in a collection or corpus.

An interesting content-based filtering system is Pandora.[1] This music recommender system uses up to 400 songs and artist properties in order to find similar songs to recommend to the original seed. These properties are a subset of the features studied by musicologists in The Music Genome Project who describe a song in terms of its melody, harmony, rhythm, and instrumentation as well as its form and the vocal performance.

[1] http://www.pandora.com/.

9.2.2 Collaborative Filtering

CF methods are constructed behind the following paradigm: "Tell me what's popular among my like-minded users". This is really intuitive paradigm since it is really similar of what people use to do: ask or look at the preferences of the people they trust. An important working hypothesis behind these kind of recommenders is that similar users tend to like similar items. In order to do so, these approaches are based on collecting and analyzing a large number of data related to the behavior, activities, or tastes of users, and predicting what users will like based on their similarity to other users. One of the main advantages of this type of system is that it does not need to "understand" what the item it recommends is.

Nowadays, these methods are extremely popular because of the simplicity and the large amount of data available from users. The main drawbacks of this kind of method is the need for a user community, as well as the *cold-start* effect for new users in the community. The cold-start problem appears when the system cannot draw any, or an optimal, inference or recommendation for the users (or items) since it has not yet obtained the sufficient information of them.

CF can be of two types: *user-based* or *item-based*.

- User-based CF works like this: Find similar users to me and recommend what they liked. In this method, given a user, U, we first find a set of other users, D, whose ratings are similar to the ratings of U and then we calculate a prediction for U.
- Item-based CF works like this: Find similar items to those that I previously liked. In item-based CF, we first build an item–item matrix that determines relationships between pairs of items; then using this matrix and data on the current user U, we infer the user's taste. Typically, this approach is used in the domain: people who buy x also buy y. This is a really popular approach used by companies like Amazon. Moreover, one of the advantages of this approach is that items usually do not change much, so its similarities can be computed offline.

9.2.3 Hybrid Recommenders

Hybrid approaches can be implemented in several ways: by making content-based and collaborative predictions separately and then combining them; by adding content-based capabilities to a collaborative approach (and vice versa); or by unifying the approaches into one model.

9.3 Modeling User Preferences

Both, CBF and CF recommender systems, require to understand the user preferences. Understanding how to model the user preference is a critical step due to the variety of sources. It is not the same when we deal with applications like the movie

recommender from Netflix, where the users rank the movies with 1 to 5 stars; or as dealing with any product recommender system from Amazon, where usually the tracking information of the purchases is used. In this case, three values can be used: 0 - not bought; 1 - viewed; 2 - bought.

The most common types of labels used to estimate the user preferences are:

- Boolean expressions (is bought?; is viewed?)
- Numerical expressions (e.g., star ranking)
- Up-Down expressions (e.g., like, neutral, or dislike)
- Weighted value expressions (e.g., number of reproductions or clicks)

In the following sections of this chapter, we only consider the numerical expression described as stars on the scale of 1 to 5.

9.4 Evaluating Recommenders

The evaluation of the recommender systems is another important step in order to assess the effectiveness of the method. When dealing with numerical labels, as the 5-star ratings, the most common way to validate a recommender system is based on their prediction value, i.e., the capacity to predict the user's choices. Standard functions such as *root mean square error* (RMSE), *precision*, *recall*, or *ROC/cost* curves have been extensively used.

However, there are several other ways to evaluate the systems. It is because metrics are entirely relevant to point of view of the person who has to evaluate it. Imagine the following three persons: (a) a marketing guy; (b) a technical system designer; and (c) a final user. It is clear that what is relevant for all of them is not the same. For a marketing guy, what is usually important is how the system helps to push the product, for the technical system designer is how efficient is the algorithm, and for the final user is if the system gives him good, or mostly cool, results. In the literature we can see two main typologies: *offline* and *online* evaluation.

We refer to evaluation as offline when a set of labeled data is obtained and then divided into two sets: a *training set* and a *test set*. The training set is used to create the model and adjust all the parameters; while the test set is used to determine selected evaluation metrics. As mentioned above, standard metrics such as RMSE, precision, and recall are extensively used, but recently other indirect functions have also started to be widely considered. Examples of these: diversity, novelty, coverage, cold-start, or serendipity, the latter is a quite popular metric that evaluates how surprising the recommendations are. For further discussion of this field, the reader is referred to [1].

We refer to evaluation as online when a set of tools is used that allows us to look at the interactions of users with the system. The most common online technique is called *A-B testing* and has the benefit of allowing evaluation of the system at the same time as users are learning, buying, or playing with the recommender system. This brings the evaluation closer to the actual working of the system and makes it really effective when the purpose of the system is to change or influence the behavior of users. In order to evaluate the test, we are interested in measuring how user behavior changes when the user is interacting with different recommender systems. Let us give an example: imagine we want to develop a music recommender system like Pandora, where your final goal is none other than for users to love your intelligent music station and spend more time listening to it. In such a situation, offline metrics like RMSE are not good enough. In this case, we are particularly interested in evaluation of the global goal of the recommender system as it is the long-term profit or user retention.

9.5 Practical Case

In this section, we will play with a real dataset to implement a movie recommender system. We will work with a user-based collaborative system with the *MovieLens* dataset.

9.5.1 MovieLens Dataset

MovieLens datasets are a collection of movie ratings produced by hundreds of users collected by the GroupLens Research Project at the University of Minnesota and released into the public domain. Several versions of this dataset can be found at the GroupLens site.[2] Figure 9.1 shows a capture of this website.

Although performance on bigger dataset is expected to be better, we will work with the smallest dataset: *MovieLens 100K Dataset*. Working with this lite version has the benefit of less computational costs, while we will also get the basic skills required on user-based recommender systems.

Once you have downloaded and unzipped the file into a directory, you can create a Pandas DataFrame with the following code:

[2] http://grouplens.org/datasets/movielens/.

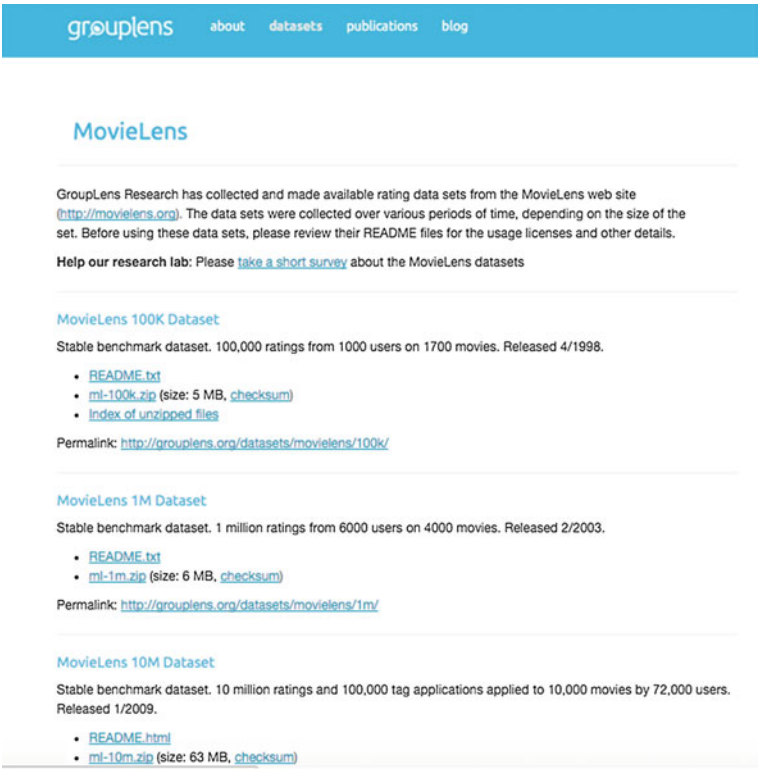


Fig. 9.1 Grouplens website

In [1]:

```
# Load user data
u_cols = [
    'user_id', 'age', 'sex',
    'occupation', 'zip_code'
    ]
users = pd.read_csv('files/ch09/ml-100k/u.user',
                    sep='|',
                    names=u_cols)

# Load movie data
r_cols = [
    'user_id', 'movie_id',
    'rating', 'unix_timestamp'
    ]
ratings = pd.read_csv('files/ch09/ml-100k/u.data',
                      sep='\t',
                      names=r_cols)

# The movie file contains columns indicating the genres of
    the movie
# We will only load the first three columns of the file with
     usecols
```

In [1]:
```
m_cols = [
    'movie_id', 'title',
    'release_date'
    ]
movies = pd.read_csv('files/ch09/ml-100k/u.item',
                     sep='|',
                     names=m_cols,
                     usecols=range(3))

# Create a DataFrame  using only the fields required
data = pd.merge(pd.merge(ratings, users), movies)
data = data[['user_id', 'title', 'movie_id', 'rating']]


print "The BD has "+ str(data.shape[0]) +" ratings"
print "The BD has ", data.user_id.nunique()," users"
print "The BD has ", data.movie_id.nunique(), " items"
print data.head()
```

Out[1]:
```
The DB has 100000 ratings
The DB has 943 different users
The DB has 1682 different items

  user_id    title    movie_id rating
0   196   Kolya (1996)    242      3
1   305   Kolya (1996)    242      5
2    6    Kolya (1996)    242      4
3   234   Kolya (1996)    242      4
4    63   Kolya (1996)    242      3
```

If you explore the dataset in detail, you will see that it consists of:

- 100,000 ratings from 943 users of 1682 movies. Ratings are from 1 to 5.
- Each user has rated at least 20 movies.
- Simple demographic info for the users (age, gender, occupation, zip).

9.5.2 User-Based Collaborative Filtering

In order to create a user-based collaborative recommender system we must define: (1) a prediction function, (2) a user similarity function, and (3) an evaluation function.

Prediction Function

The prediction function behind the user-based CF will be based on the movie ratings from similar users. So, in order to recommend a movie, p, from a set of movies, P, to a given user, a, we first need to see the set of users, B, who have already seen p. Then, we need to see the taste similarity between these users in B and user a. The most simple prediction function for a user a and movie p can be defined as follows:

$$pred(a, p) = \frac{\sum_{b \in B} sim(a, b)(r_{b,p})}{\sum_{b \in B} sim(a, b)} \tag{9.1}$$

Table 9.1 Recommender System

Critic	sim(a,b)	Rating movie1: r_{b,p_1}	$sim(a,b)(r_{b,p_1})$
Paul	0.99	3	2.97
Alice	0.38	3	1.14
Marc	0.89	4.5	4.0
Anne	0.92	3	2.77
$\sum_{b \in N} sim(a,b)(r_{b,p})$			10.87
$\sum_{b \in N} sim(a,b)$			3.18
$pred(a,p)$			3.41

where $sim(a,b)$ is the similarity between user a and user b, B is the set of users in the dataset that have already seen p and $r_{b,p}$ is the rating of p by b.

Let us give an example (see Table 9.1). Imagine the system can only recommend one movie, since the rest have already been seen by the user. So, we only want to estimate the score corresponding to that movie. The movie has been seen by Paul, Alice, Marc, and Anne and scored 3, 3, 4, and 3, respectively. Similarity between user a and Paul, Alice, Marc, and Anne has been computed "somehow" (we will see later how we can compute it) and the values are 0.99, 0.38, 0.89, and 0.92, respectively. If we follow the previous equation, the estimated score is 3.41, as seen in Table 9.1.

User Similarity Function

The computation of the similarity between users is one of the most critical steps in the CF algorithms. The basic idea behind the similarity computation between two users a and b is that we can first isolate the set P of items rated by both users, and then apply a similarity computation technique to determine the similarity.

The set of `common_movies` can be obtained with the following code:

In [2]:

```
# dataframe with the data from user 1
df_usr1 = data_train[data_train.user_id == 1]

# dataframe with the data from user 2
df_usr2 = data_train[data_train.user_id == 6]

# We first compute the set of common movies
common_mov = set(df_usr1.movie_id).intersection(
    df_usr2.movie_id)

print "\nNumber of common movies",
      len(common_mov)
```

In [2]:

```
# Sub-dataframe with only the common movies
mask = (data_user_1.movie_id.isin(common_movies))
data_user_1 = data_user_1[mask]
print data_user_1[['title', 'rating']].head()

mask = (data_user_2.movie_id.isin(common_movies))
data_user_2 = data_user_2[mask]
print data_user_2[['title', 'rating']].head()
```

Out[2]:

```
Number of common movies 11
Movies User 1
                     title                 rating
14                Kolya (1996)                5
417         Shall We Dance? (1996)            4
1306 Truth About Cats & Dogs, The (1996)      5
1618         Birdcage, The (1996)             4
3479          Men in Black (1997)             4
Movies User 2
                     title                 rating
32                Kolya (1996)                5
424         Shall We Dance? (1996)            5
1336 Truth About Cats & Dogs, The (1996)      4
1648         Birdcage, The (1996)             4
3510          Men in Black (1997)             4
```

Once the set of ratings for all movies common to the two users has been obtained, we can compute the user similarity. Some of the most common similarity functions used in CF methods are as follows:

Euclidean distance:

$$sim(a,b) = \frac{1}{1+\sqrt{\sum_{p\in P}(r_{a,p}-r_{b,p})^2}} \tag{9.2}$$

Pearson correlation:

$$sim(a,b) = \frac{\sum_{p\in P}(r_{a,p}-\bar{r_a})(r_{b,p}-\bar{r_b})}{\sqrt{\sum_{p\in P}(r_{a,p}-\bar{r_a})}\sqrt{\sum_{p\in P}(r_{b,p}-\bar{r_b})}} \tag{9.3}$$

where $\bar{r_a}$ and $\bar{r_b}$ are the mean ratings of users a and b.

Cosine distance:

$$sim(a,b) = \frac{a\cdot b}{|a|\cdot|b|} \tag{9.4}$$

Now, the question: Which function should we use? The answer is that there is no fixed recipe; but there are some issues we can take into account when choosing the proper similarity function. On the one hand, Pearson correlation usually works better than Euclidean distance since it is based more on the ranking than on the values. So, two users who usually like more the same set of items, although their rating is on different scales, will come out as similar users with Pearson correlation but not with Euclidean distance. On the other hand, when dealing with binary/unary data, i.e.,

like versus not like or buy versus not buy, instead of scalar or real data like ratings, cosine distance is usually used.

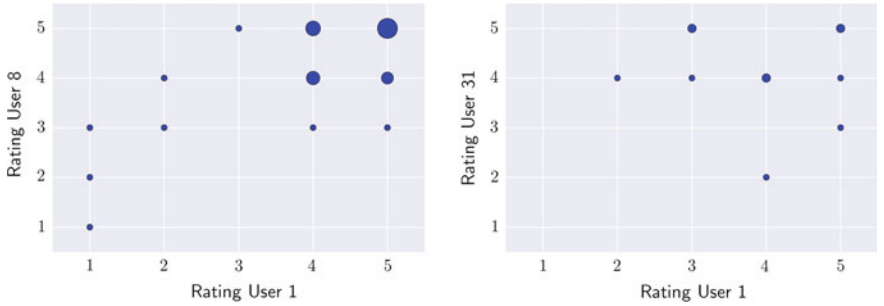
Let us define the Euclidean and Pearson functions:

In [3]:
```
from scipy.spatial.distance import Euclidean

# Similarity based on Euclidean distance for users 1-2
def SimEuclid(df,User1,User2,min_common_items=10):
    # GET MOVIES OF USER1
    mov_u1 = df[df['user_id'] == User1 ]
    # GET MOVIES OF USER2
    mov_u2 = df[df['user_id'] == User2 ]

    # FIND SHARED FILMS
    rep = pd.merge(mov_u1, mov_u2, on = 'movie_id')
    if len(rep) == 0:
        return 0
    if(len(rep) < min_common_items):
        return 0
    return 1.0 / (1.0+euclidean(rep['rating_x'],
                                rep['rating_y']))
```

In [4]:
```
from scipy.stats import pearsonr

# Similarity based on Pearson correlation for user 1-2
def SimPearson(df, User1, User2, min_common_items = 10):
    # GET MOVIES OF USER1
    mov_u1 = df[df['user_id'] == User1 ]
    # GET MOVIES OF USER2
    mov_u2 = df[df['user_id'] == User2 ]

    # FIND SHARED FILMS
    rep = pd.merge(mov_u1, mov_u2, on = 'movie_id')
    if len(rep)==0:
        return 0
    if(len(rep) < min_common_items):
        return 0
    return pearsonr(rep['rating_x'], rep['rating_y']) [0]
```

Figure 9.2 shows the correlation plots for user 1 versus user 8 and user 1 versus user 31. Each point in the plots corresponds to a different set of ratings from the two users of the same movies. The bigger the dot, the larger the set of movies rated with the corresponding values. We can observe in these plots that ratings from user 1 are more correlated with ratings from user 8 than from the user 31. However, as we can observe in the following outputs, the Euclidean similarity between user 1 and user 31 is closer than between user 1 and user 8.

In [5]:
```
print "Euclidean similarity",SimEuclid(data_train, 1, 8)
print "Pearson similarity",SimPearson(data_train, 1, 8)

print "Euclidean similarity",SimEuclid(data_train, 1, 31)
print "Pearson similarity",SimPearson(data_train, 1, 31)
```



(a) User 1 vs. 8 (b) User 1 vs. 31

Fig. 9.2 Similarity between users

```
Out[5]: Euclidean similarity 0.195194101601
        Pearson similarity 0.773097845465

        Euclidean similarity 0.240253073352
        Pearson similarity 0.272165526976
```

Evaluation

In order to validate the system, we will divide the dataset into two different sets: one called `X_train` containing 80% of the data from each user; and another called `X_test`, with the remaining 20% of the data from each user. In the following code we create a function `assign_to_set` that creates a new column in the DataFrame indicating which sample it belongs to.

In [6]:
```
def assign_to_set(df):
    sampled_ids = np.random.choice(
        df.index,
        size = np.int64(np.ceil(df.index.size * 0.2)),
        replace=False)
    df.ix[sampled_ids, 'for_testing'] = True
    return df

data['for_testing'] = False
grouped = data.groupby('user_id', group_keys = False)
              .apply(assign_to_set)
X_train = data[grouped.for_testing == False]
X_test  = data[grouped.for_testing == True]
```

The resulting `X_train` and `X_test` sets have 79619 and 20381 ratings, respectively.

Once the data is divided in these sets, we can build a model with the training set and evaluate its performance using the test set. In our case, the evaluation will be performed using the standard RMSE:

$$RMSE = \sqrt{\left(\frac{\sum(\hat{y} - y)^2}{n}\right)} \tag{9.5}$$

where y is the real rating and $\hat{y}$ is the predicted rating.

In [7]:
```
def compute_rmse(y_pred, y_true):
    """ Compute Root Mean Squared Error. """
    return np.sqrt(np.mean(np.power(y_pred - y_true, 2)))
```

Collaborative Filtering Class

We can define our recommender system with a Python class. This class consists of a constructor and two methods: `fit` and `predict`. In the `fit` method the user's similarities are computed and stored into a Python dictionary. This is a really simple method but quite expensive in terms of computation when dealing with a large dataset. We decided to show one of the most basic schemes in order to implement it. More complex algorithms can be used in order to improve the computations cost. Moreover, online strategies can be used when dealing with a really dynamic problems. In the `predict` the score for a movie and a user is estimated.

In [8]:
```
class CollaborativeFiltering:
    """ CF using a custom sim(u,u'). """
    def __init__(self, df, similarity = SimPearson):
        """ Constructor """
        self.sim_method = similarity
        self.df = df
        self.sim = pd.DataFrame(
        np.sum([0]), columns = df.user_id.unique(),
        index = df.user_id.unique())

    def fit(self):
        """ Prepare data structures for estimation.
            Similarity matrix for users """
        allUsers = set(self.df['user_id'])
        self.sim = {}
        for person1 in allUsers:
            self.sim.setdefault(person1, {})
            a = self.df[
                self.df['user_id'] == person1][['movie_id']
                ]
            data_reduced = pd.merge(self.df, a,
                                    on = 'movie_id')
            for person2 in allUsers:
                # Avoid our-self
                if person1 == person2: continue
                self.sim.setdefault(person2, {})
                if(self.sim[person2].has_key(person1)):
                    continue # since symmetric matrix
                sim = self.sim_method(data_reduced,
                                      person1,
                                      person2)
                if(sim < 0):
                    self.sim[person1][person2] = 0
                    self.sim[person2][person1] = 0
                else:
                    self.sim[person1][person2] = sim
                    self.sim[person2][person1] = sim

    def predict(self, user_id, movie_id):
        totals = {}
        users = self.df[self.df['movie_id'] == movie_id]
```

In [11]:
```
rating_num, rating_den = 0.0, 0.0
        allUsers = set(users['user_id'])
        for other in allUsers:
            if user_id == other: continue
            rating_num +=
                self.sim[user_id][other] * float(users[users
                    ['user_id'] == other]['rating'])
           rating_den += self.sim[user_id][other]
        if rating_den == 0:
            if self.df.rating[self.df['movie_id'] ==
                movie_id].mean() > 0:
                # Mean movie rating if there is no similar
                    for the computation
               return self.df.rating[self.df['movie_id'] ==
                   movie_id].mean()
            else:
                # else mean user rating
                return self.df.rating[self.df['user_id'] ==
                    user_id].mean()
        return rating_num/rating_den
```

For the evaluation of the system we define a function called `evaluate`. This function estimates the score for all items in the test set (`X_test`) and compares them with the real values using the RMSE.

In [9]:
```
def evaluate(fit_f,train,test):
    """ RMSE-based predictive performance evaluation with
        pandas. """
    ids_to_estimate = zip(test.user_id, test.movie_id)
    estimated = np.array([fit_f(u, i)
                          if u
                          in train.user_id
                          else 3
                          for (u, i)
                          in ids_to_estimate])
    real = test.rating.values
    return compute_rmse(estimated, real)
```

Now, the system can be executed with the following lines:

In [10]:
```
print 'RMSE for Collaborative Recommender:',
print ' %s' % evaluate(reco.fit, data_train, data_test)
```

Out[10]:
```
RMSE for Collaborative Recommender: 1.00468945461
```

As we can see, the obtained $RMSE$ for this first basic recommender system is 1.004. Sure, that this result could be improved with a bigger dataset, but let us think of how we can improve it with just few tricks:

Trick 1: Since humans do not usually act the same as critics, i.e., some people usually rank movies higher or lower than others, this prediction function can be easily improved by taking into account the user mean as follows:

$$pred(a, p) = \bar{r_a} + \frac{\sum_{b \in B} sim(a, b) * (r_{b,p} - \bar{r_b})}{\sum_{b \in B} sim(a, b)} \tag{9.6}$$

where $\bar{r_a}$ and $\bar{r_b}$ are the mean rating of user a and b.

Table 9.2 Recommender system using mean user ratings

Critic	sim(a,b)	Mean ratings: $\bar{r_b}$	Rating movie1: r_{b,p_1}	$sim(a,b) * (r_{b,p_1})$
Paul	0.99	4.3	3	−1.28
Alice	0.38	2.73	3	0.1
Marc	0.89	3.12	4.5	1.22
Anne	0.92	3.98	3	−0.9
$\sum_{b \in N} sim(a,b) * (r_{b,p} - \bar{r_b})$				−1.13
$\sum_{b \in N} sim(a,b)$				3.18
$pred(a,p)$				3.14

Let us see an example: Prediction for the user "a" with $\bar{r_a} = 3.5$ (Table 9.2)

If we modify the recommender system using Eq. (9.6), the RMSE obtained is the following:

```
Out[11]: RMSE for Collaborative Recommender: 0.950086206741
```

Trick 2: One of the most critical steps with this kind of recommender system is the user similarity computation. If two users have very few items in common, let us imagine that there is only one, and the rating is the same, the user similarity will be really high; however, the confidence is really small. In order to solve this problem we can modify the similarity function as follows:

$$new\_sim(a,b) = sim(a,b) * \frac{min(K, |P_{ab}|)}{K} \tag{9.7}$$

where $|P_{ab}|$ is the number of common items shared by user a and user b, and K is the minimum number of common items in order not to penalize the similarity function.

In the next code, we define an update version of the similarity function called `simPersonCorrected` that follows the Eq. 9.7.

In [12]:
```
def SimPearsonCorrected(df, User1, User2,
                        min_common_items = 1,
                        pref_common_items = 20):
    """ RMSE-based predictive performance evaluation with
        pandas. """
    # GET MOVIES OF USER1
    m_user1 = df[df['user_id'] == User1 ]
    # GET MOVIES OF USER2
    m_user2 = df[df['user_id'] == User2 ]

    # FIND SHARED FILMS
    rep = pd.merge(m_user1, m_user2, on = 'movie_id')
    if len(rep) == 0:
        return 0
    if(len(rep) < min_common_items):
        return 0
```

In [12]:
```
        res = pearsonr(rep['rating_x'], rep['rating_y'])[0]
        res = res * min(pref_common_items, len(rep))
        res = res / pref_common_items
        if(isnan(res)):
            return 0
        return res
    reco4 = CollaborativeFiltering3(
        data_train,
        similarity = SimPearsonCorrected)
    reco4.learn()

    print 'RMSE for Collaborative Recommender:',
    print ' %s' % evaluate(reco4.fit, data_train, data_test)
```

Out[12]:
```
RMSE for Collaborative Recommender: 0.930811091922
```

As it can be seen, with this small modification the RMSE error has decreased from 1.0 to 0.93.

9.6 Conclusions

In this chapter, we have introduced what are recommender systems, how they work, and how they can be implemented in Python. We have seen that there are different types of recommender systems based on the information they use, as well as the output they produce. We have introduced content-based recommender systems and collaborative recommender systems; and we have seen the importance of defining the similarity function between items and users.

We have learned how recommender system can be implemented in Python in order to answer questions such as which movie should I see? We have also discussed how recommender system should be evaluated, and several online and offline metrics.

Finally, we have worked with a publicly available dataset from GroupLens in order to implement and evaluate a collaborative recommendation system for movie recommendations.

Acknowledgements This chapter was co-written by Santi Seguí and Eloi Puertas

References

1. G. Shani, A. Gunawardana, A survey of accuracy evaluation metrics of recommendation tasks. in *J. Mach. Learn. Res.*, 10:2935–2962, 2009
2. F. Ricci, L. Rokach, B. Schapira, in *Recommender Systems Handbook* (Springer, 2015).