

CHAPTER 7

Recommender Systems

A common trend that can be observed in brick and mortar stores, is that we have salespeople guiding and recommending us relevant products while shopping on the other hand, with online retail platforms, there are zillions of different products available, and we have to navigate ourselves to find the right product. The situation is that users have too many options and choices available, yet they don't like to invest a lot of time going through the entire catalogue of items. Hence, the role of Recommender Systems (RS) becomes critical for recommending relevant items and driving customer conversion.

Traditional physical stores use planograms to arrange the items in such a way that can increase the visibility of high-selling items and increase revenue whereas online retail stores need to keep it dynamic based on preferences of each individual customer rather than keeping it the same for everyone.

Recommender systems are mainly used for auto-suggesting the right content or product to the right users in a personalized manner to enhance the overall experience. Recommender systems are really powerful in terms of using huge amounts of data and learning to understand the preferences of specific users. Recommendations help users to easily navigate through millions of products or tons of content (articles/videos/movies) and show them the right item/information that they might like or buy. So, in simple

terms, RS help discover information on behalf of the users. Now, it depends on the users to decide if RS did a good job at recommendations or not, and they can choose to either select the product/content or discard and move on. Each of the decisions of users (Positive or Negative) helps to retrain the RS on the latest data to be able to give even better recommendations. In this chapter, we will go over how RS work and the different types of techniques used under the hood for making these recommendations. We will also build a recommender system using PySpark.

Recommendations

Recommender systems can be used for multiple purposes in the sense of recommending various things to users. For example, some of them might fall in the categories below:

1. Retail Products
2. Jobs
3. Connections/Friends
4. Movies/Music/Videos/Books/Articles
5. Ads

The “What to Recommend” part totally depends on the context in which RS are used and can help the business to increase revenues by providing the most likely items that users can buy or increasing the engagement by showcasing relevant content at the right time. RS take care of the critical aspect that the product or content that is being recommended should either be something which users might like but would not have discovered on their own. Along with that, RS also need an element of varied recommendations to keep it interesting enough. A few examples of heavy usage of RS by businesses today such as Amazon products, Facebook’s friend suggestions, LinkedIn’s “People you may know,” Netflix’s movie, YouTube’s videos, Spotify’s music, and Coursera’s courses.

The impact of these recommendations is proving to be immense from a business standpoint, and hence more time is being spent in making these RS more efficient and relevant. Some of the immediate benefits that RS offer in retail settings are:

1. Increased Revenue
2. Positive Reviews and Ratings by Users
3. Increased Engagement

For the other verticals such as ads recommendations and other content recommendation, RS help immensely to help them find the right thing for users and hence increases adoption and subscriptions. Without RS, recommending online content to millions of users in a personalized manner or offering generic content to each user can be incredibly off target and lead to negative impacts on users.

Now that we know the usage and features of RS, we can take a look at different types of RS. There are mainly five types of RS that can be built:

1. Popularity Based RS
2. Content Based RS
3. Collaborative Filtering based RS
4. Hybrid RS
5. Association Rule Mining based RS

We will briefly go over each one of these except for the last item, that is, Association Rule Mining based RS as it's out of the scope of this book.

Popularity Based RS

This is the most basic and simplest RS that can be used to recommend products or content to the users. It recommends items/content based on bought/viewed/liked/downloaded by most of the users. While it is

easy and simple to implement, it doesn't produce relevant results as the recommendations stay the same for every user, but it sometimes outperforms some of the more sophisticated RS. The way this RS is implemented is by simply ranking the items on various parameters and recommending the top-ranked items in the list. As already mentioned, items or content can be ranked by the following:

1. No. of times downloaded
2. No. of times bought
3. No. of times viewed
4. Highest rated
5. No. of times shared
6. No. of times liked

This kind of RS directly recommends the best-selling or most watched/bought items to the customers and hence increases the chances of customer conversion. The limitation of this RS is that it is not hyper-personalized.

Content Based RS

This type of RS recommends similar items to the users that the user has liked in the past. So, the whole idea is to calculate a similarity score between any two items and recommended to the user based upon the profile of the user's interests. We start with creating item profiles for each of the items. Now these item profiles can be created in multiple ways, but the most common approach is to include information regarding the details or attributes of the item. For an example, the item profile of a Movie can have values on various attributes such as Horror, Art, Comedy, Action, Drama, and Commercial as shown below.

Movie ID	Horror	Art	Comedy	Action	Drama	Commercial
2310	0.01	0.3	0.8	0.0	0.5	0.9

Above is the example of an item profile, and each of the items would have a similar vector representing its attributes. Now, let's assume the user has watched 10 such movies and really liked them. So, for that particular user, we end up with the item matrix shown in Table 7-1.

Table 7-1. *Movie Data*

Movie ID	Horror	Art	Comedy	Action	Drama	Commercial
2310	0.01	0.3	0.8	0.0	0.5	0.9
2631	0.0	0.45	0.8	0.0	0.5	0.65
2444	0.2	0.0	0.8	0.0	0.5	0.7
2974	0.6	0.3	0.0	0.6	0.5	0.3
2151	0.9	0.2	0.0	0.7	0.5	0.9
2876	0.0	0.3	0.8	0.0	0.5	0.9
2345	0.0	0.3	0.8	0.0	0.5	0.9
2309	0.7	0.0	0.0	0.8	0.4	0.5
2366	0.1	0.15	0.8	0.0	0.5	0.6
2388	0.0	0.3	0.85	0.0	0.8	0.9

User Profile

The other component in content based RC is the User Profile that is created using item profiles that the user has liked or rated. Assuming that the user has liked the movie in Table 7-1, the user profile might look like a single vector, which is simply the mean of item vectors. The user profile might look something like that below.

User ID	Horror	Art	Comedy	Action	Drama	Commercial
1A92	0.251	0.23	0.565	0.21	0.52	0.725

This approach to create the user profile is one of the most baseline ones, and there are other sophisticated ways to create more enriched user profiles such as normalized values, weighted values, etc. The next step is to recommend the items (movies) that this user might like based on the earlier preferences. So, the similarity score between the user profile and item profile is calculated and ranked accordingly. The more the similarity score, the higher the chances of liking the movie by the user. There are a couple of ways by which the similarity score can be calculated.

Euclidean Distance

The user profile and item profile both are high-dimensional vectors and hence to calculate the similarity between the two, we need to calculate the distance between both vectors. The Euclidean distance can be easily calculated for an n-dimensional vector using the formula below:

$$d(x,y)=\sqrt{(x_1-y_n)^2+\dots+(x_n-y_n)^2}$$

The higher the distance value, the less similar are the two vectors. Therefore, the distance between the user profile and all other items are calculated and ranked in decreasing order. The top few items are recommended to the user in this manner.

Cosine Similarity

Another way to calculate a similarity score between the user and item profile is cosine similarity. Instead of distance, it measures the angle between two vectors (user profile vector and item profile vector). The

smaller the angle between both vectors, the more similar they are to each other. The cosine similarity can be found out using the formula below:

$$\text{sim}(x,y)=\cos(\theta)=x*y / |x|*|y|$$

Let's go over some of the pros and cons of Content based RS.

Advantages:

1. Content based RC works independently of other users' data and hence can be applied to an individual's historical data.
2. The rationale behind RC can be easily understood as the recommendations are based on the similarity score between the User Profile and Item Profile.
3. New and unknown items can also be recommended to users just based on historical interests and preferences of users.

Disadvantages:

1. Item profile can be biased and might not reflect exact attribute values and might lead to incorrect recommendations.
2. Recommendations entirely depend on the history of the user and can only recommend items that are like the historically watched/liked items and do not take into consideration the new interests or liking of the visitor.

Collaborative Filtering Based RS

CF based RS doesn't require the item attributes or description for recommendations; instead it works on user item interactions. These interactions can be measured in various ways such as ratings, item bought,

time spent, shared on another platform, etc. Before diving deep in CF, let's take a step back and reflect on how we make certain decisions on a day-to-day basis – decisions such as the following:

1. Which movie to watch
2. Which book to read
3. Which restaurant to go to
4. Which place to travel to

We ask our friends, right! We ask for recommendations from people who are similar to us in some ways and have same tastes and likings as ours. Our interests match in some areas and so we trust their recommendations. These people can be our family members, friends, colleagues, relatives, or community members. In real life, it's easy to know who are the people falling in this circle, but when it comes to online recommendations, the key task in collaborative filtering is to find the users who are most similar to you. Each user can be represented by a vector that contains the feedback value of a user item interaction. Let's understand the user item matrix first to understand the CF approach.

User Item Matrix

The user item matrix is exactly what the name suggests. In the rows, we have all the unique users; and along the columns, we have all the unique items. The values are filled with feedback or interaction scores to highlight the liking or disliking of the user for that product. A simple User Item matrix might look something like shown in Table 7-2.

Table 7-2. *User Item Matrix*

User ID	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
14SD	1	4			5	
26BB		3	3			1
24DG	1	4	1		5	2
59YU		2			5	
21HT	3	2	1	2	5	
68BC		1				5
26DF	1	4		3	3	
25TR	1	4			5	
33XF	5	5	5	1	5	5
73QS	1		3			1

As you can observe, the user item matrix is generally very sparse as there are millions of items, and each user doesn't interact with every item; so the matrix contains a lot of null values. The values in the matrix are generally feedback values deduced based upon the interaction of the user with that particular item. There are two types of feedback that can be considered in the UI matrix.

Explicit Feedback

This sort of feedback is generally when the user gives ratings to the item after the interaction and has been experiencing the item features. Ratings can be of multiple types.

1. Rating on 1-5 scale
2. Simple rating item on recommending to others
(Yes or No or never)
3. Liked the Item (Yes or No)

The Explicit feedback data contains very limited amounts of data points as a very small percentage of users take out the time to give ratings even after buying or using the item. A perfect example can be of a movie, as very few users give the ratings even after watching it. Hence, building RS solely on explicit feedback data can put us in a tricky situation, although the data itself is less noisy but sometimes not enough to build RS.

Implicit Feedback

This kind of feedback is not direct and mostly inferred from the activities of the user on the online platform and is based on interactions with items. For example, if user has bought the item, added it to the cart, viewed, and spent a great deal of time on looking at the information about the item, this indicates that the user has a higher amount of interest in the item. Implicit feedback values are easy to collect, and plenty of data points are available for each user as they navigate their way through the online platform. The challenges with implicit feedback are that it contains a lot of noisy data and therefore doesn't add too much value in the recommendations.

Now that we understand the UI matrix and types of values that go into that matrix, we can see the different types of collaborative filtering (CF). There are mainly two kinds of CF:

1. Nearest Neighbors based CF
2. Latent Factor based CF

Nearest Neighbors Based CF

This CF works by finding out the k-nearest neighbors of users by finding the most similar users who also like or dislike the same items as the active user (for user we are trying to recommend). There are two steps involved in the nearest neighbor's collaborative filtering. The first step is to find k-nearest neighbors, and the second step is to predict the rating or likelihood of the active user liking a particular item. The k-nearest

neighbors can be found out using some of the earlier techniques we have discussed in the chapter. Metrics such as cosine similarity or Euclidean distance can help us to find most similar users to active users out of the total number of users based on the common items that both groups have liked or disliked. One of the other metrics that can also be used is Jaccard similarity. Let's look at an example to understand this metric – going back to the earlier user item matrix and taking just five users' data as shown in Table 7-3.

Table 7-3. *User Item Matrix*

User ID	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
14SD	1	4			5	
26BB		3	3			1
24DG	1	4	1		5	2
59YU		2			5	
26DF	1	4		3	3	

Let's say we have in total five users and we want to find the two nearest neighbors to the active user (14SD). The Jaccard similarity can be found out using

$$\text{sim}(x,y)=|R_x \cap R_y| / |R_x \cup R_y|$$

So, this is the number of items that any two users have rated in common divided by the total number of items that both users have rated:

$\text{sim}(\text{user1}, \text{user2}) = 1 / 5 = 0.2$ since they have rated only Item 2 in common).

The similarity score for the rest of the four users with active users would then look something like that shown in Table 7-4.

Table 7-4. *User Similarity Score*

User ID	Similarity Score
14SD	1
26BB	0.2
24DG	0.6
59YU	0.677
26DF	0.75

So, according to Jaccard similarity the top two nearest neighbors are the fourth and fifth users. There is a major issue with this approach, though, because the Jaccard similarity doesn't consider the feedback value while calculating the similarity score and only considers the common items rated. So, there could be a possibility that users might have rated many items in common, but one might have rated them high and the other might have rated them low. The Jaccard similarity score still might end up with a high score for both users, which is counterintuitive. In the above example, it is clearly evident that the active user is most similar to the third user (24DG) as they have the exact same ratings for three common items whereas the third user doesn't even appear in the top two nearest neighbors. Hence, we can opt for other metrics to calculate the k-nearest neighbors.

Missing Values

The user item matrix would contain lot of missing values for the simple reason that there are lot of items and not every user interacts with each item. There are a couple of ways to deal with missing values in the UI matrix.

- 1. Replace the missing value with 0s.
- 2. Replace the missing values with average ratings of the user.

The more similar the ratings on common items, the nearer the neighbor is to the active user. There are, again, two categories of Nearest Neighbors based CF

1. User based CF
2. Item based CF

The only difference between both RS is that in user based we find k-nearest users, and in item based CF we find k-nearest items to be recommended to users. We will see how recommendations work in user based RS.

As the name suggests, in user based CF, the whole idea is to find the most similar user to the active user and recommend the items that the similar user has bought/rated highly to the active user, which he hasn't seen/bought/tried yet. The assumption that this kind of RS makes is that if two or more users have the same opinion about a bunch of items, then they are likely to have the same opinion about other items as well. Let's look at an example to understand the user based collaborative filtering: there are three users, out of which we want to recommend a new item to the active user. The rest of the two users are the top two nearest neighbors in terms of likes and dislikes of items with the active user as shown in Figure 7-1.

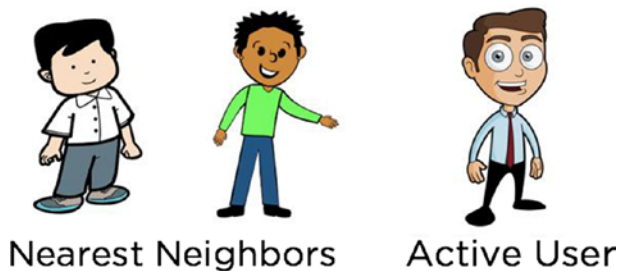


Figure 7-1. Active User and Nearest Neighbors

All three users have rated a particular camera brand very highly, and the first two users are the most similar users to the active user based on a similarity score as shown in Figure 7-2.

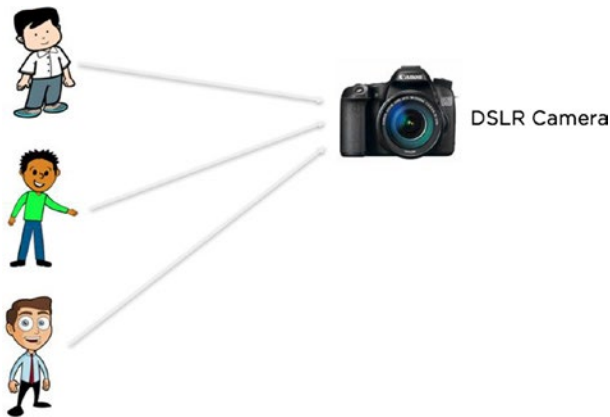


Figure 7-2. All users like a item

Now, the first two users have also rated another item (Xbox 360) very highly, which the third user is yet to interact with and has also not seen as shown in Figure 7-3. Using this information, we try to predict the rating that the active user would give to the new item (Xbox 360), which again is the weighted average of ratings of the nearest neighbors for that particular item (XBOX 360).

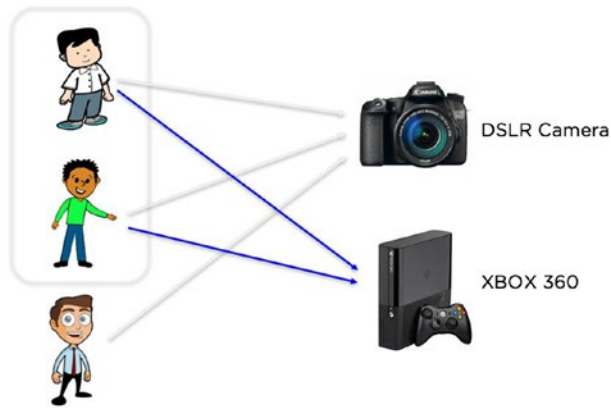


Figure 7-3. *Nearest Neighbors also like the other item*

The user based CF then recommends the other item (XBOX 360) to the active user since he is most likely to rate this item higher as the nearest neighbors have also rated this item highly as shown in Figure 7-4.

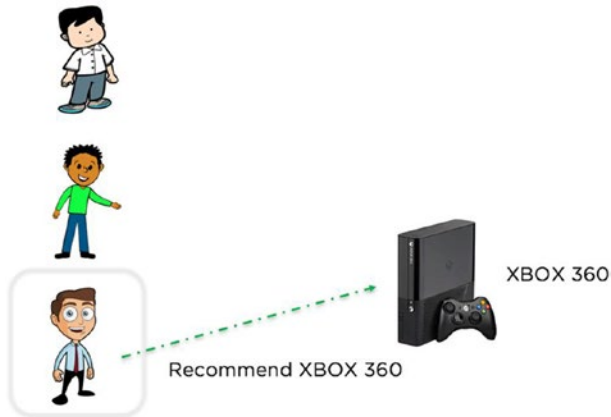


Figure 7-4. *Active User Recommendation*

Latent Factor Based CF

This kind of collaborative filtering also uses the user item matrix but instead of finding the nearest neighbors and predicting ratings, it tries to decompose the UI matrix into two latent factor matrices. The latent factors are derived values from original values. They are intrinsically related to the observed variables. These new matrices are much lower in terms of rank and contain latent factors. This is also known as matrix factorization. Let's take an example to understand the matrix factorization process. We can decompose an $m \times n$ size matrix 'A' of rank r into two smaller rank matrices X, Y such that the dot product of X and Y results in the original A matrix. If we have a matrix A shown in Table 7-5,

Table 7-5. Latent Factor Calculation

1	2	3	5
2	4	8	12
3	6	7	13

then we can write all the column values as linear combinations of the first and third columns (A1 and A3).

$$A1 = 1 * A1 + 0 * A3$$

$$A2 = 2 * A1 + 0 * A3$$

$$A3 = 0 * A1 + 1 * A3$$

$$A4 = 2 * A1 + 1 * A3$$

Now we can create the two small rank matrices in such a way that the product between those two would return the original matrix A.

$$X = \begin{array}{c|cc} & \mathbf{1} & \mathbf{3} \\ \hline 2 & 2 & 8 \\ 3 & 3 & 7 \end{array}$$

$$Y = \begin{array}{c|cccc} & \mathbf{1} & \mathbf{2} & \mathbf{0} & \mathbf{2} \\ \hline 0 & 0 & 0 & 1 & 1 \end{array}$$

X contains columns values of A1 and A3 and Y contains the coefficients of linear combinations.

The dot product between X and Y results back into matrix 'A' (original matrix)

Considering the same user item matrix as shown in Table 7-2, we factorize or decompose it into two smaller rank matrices.

1. Users latent factor matrix
2. Items latent factor matrix

CHAPTER 7 RECOMMENDER SYSTEMS

User ID	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
14SD	1	4			5	
26BB		3	3			1
24DG	1	4	1		5	2
59YU		2			5	
21HT	3	2	1	2	5	
68BC		1				5
26DF	1	4		3	3	
25TR	1	4			5	
33XF	5	5	5	1	5	5
73QS	1		3			1



	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
ITF1	0.3	0.23			0.9	
ITF2		0.1	0.14			0.02
ITF3	0.25	0.8	0.09		0.9	0.33

Item latent Factor Matrix

User ID	USF1	USF2	USF3
14SD	0.02	0.97	
26BB		0.24	0.65
24DG	0.03	0.86	0.07
59YU		0.45	
21HT	0.65	0.38	0.05
68BC		0.03	
26DF	0.02	0.78	
25TR	0.01	0.84	
33XF	0.95	0.98	0.93
73QS	0.03		0.48

User latent Factor Matrix


0.23
0.1
0.8

The user latent factor matrix contains all the users mapped to these latent factors, and similarly the item latent factor matrix contains all items in columns mapped to each of the latent factors. The process of finding these latent factors is done using machine learning optimization

techniques such as Alternating Least squares. The user item matrix is decomposed into latent factor matrices in such a way that the user rating for any item is the product between a user's latent factor value and the item latent factor value. The main objective is to minimize the total sum of squared errors over the entire user item matrix ratings and predicted item ratings. For example, the predicted rating of the second user (26BB) for Item 2 would be

Rating (user2, item2) =

	0.24	0.65
--	------	------



There would be some amount of error on each of the predicted ratings, and hence the cost function becomes the overall sum of squared errors between predicted ratings and actual ratings. Training the recommendation model includes learning these latent factors in such a way that it minimizes the SSE for overall ratings. We can use the ALS method to find the lowest SSE. The way ALS works is that it fixes first the user latent factor values and tries to vary the item latent factor values such that the overall SSE reduces. In the next step, the item latent factor values are kept fixed, and user latent factor values are updated to further reduce the SSE. This keeps alternating between the user matrix and item matrix until there can be no more reduction in SSE.

Advantages:

1. Content information of the item is not required, and recommendations can be made based on valuable user item interactions.
2. Personalizing experience based on other users.

Limitations:

1. Cold Start Problem: If the user has no historical data of item interactions, then RC cannot predict the k-nearest neighbors for the new user and cannot make recommendations.
2. Missing values: Since the items are huge in number and very few users interact with all the items, some items are never rated by users and can't be recommended.
3. Cannot recommend new or unrated items: If the item is new and yet to be seen by the user, it can't be recommended to existing users until other users interact with it.
4. Poor Accuracy: It doesn't perform that well as many components keep changing such as interests of users, limited shelf life of items, and very few ratings of items.

Hybrid Recommender Systems

As the name suggests, the hybrid RS include inputs from multiple recommender systems, making it more powerful and relevant in terms of meaningful recommendations to the users. As we have seen, there are a few limitations in using individual RS, but in combination they overcome few of those and hence are able to recommend items or information that users find more useful and personalized. The hybrid RS can be built in specific ways to suit the requirement of the business. One of the approaches is to build individual RS and combine the recommendations from multiple RS output before recommending them to the user as shown in Figure 7-5.

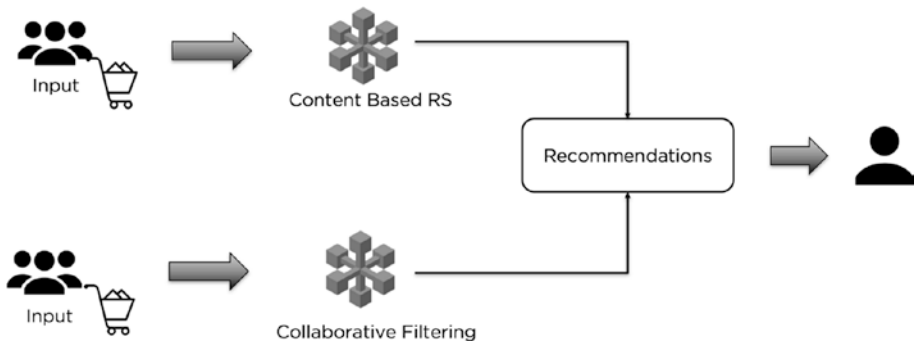


Figure 7-5. *Combining Recommendations*

The other approach is by leveraging content based recommender strengths and using them as input for collaborative filtering based recommendations to provide better recommendations to the users. This approach can also be reversed, and collaborative filtering can be used as input for content based recommendations as shown in Figure 7-6.

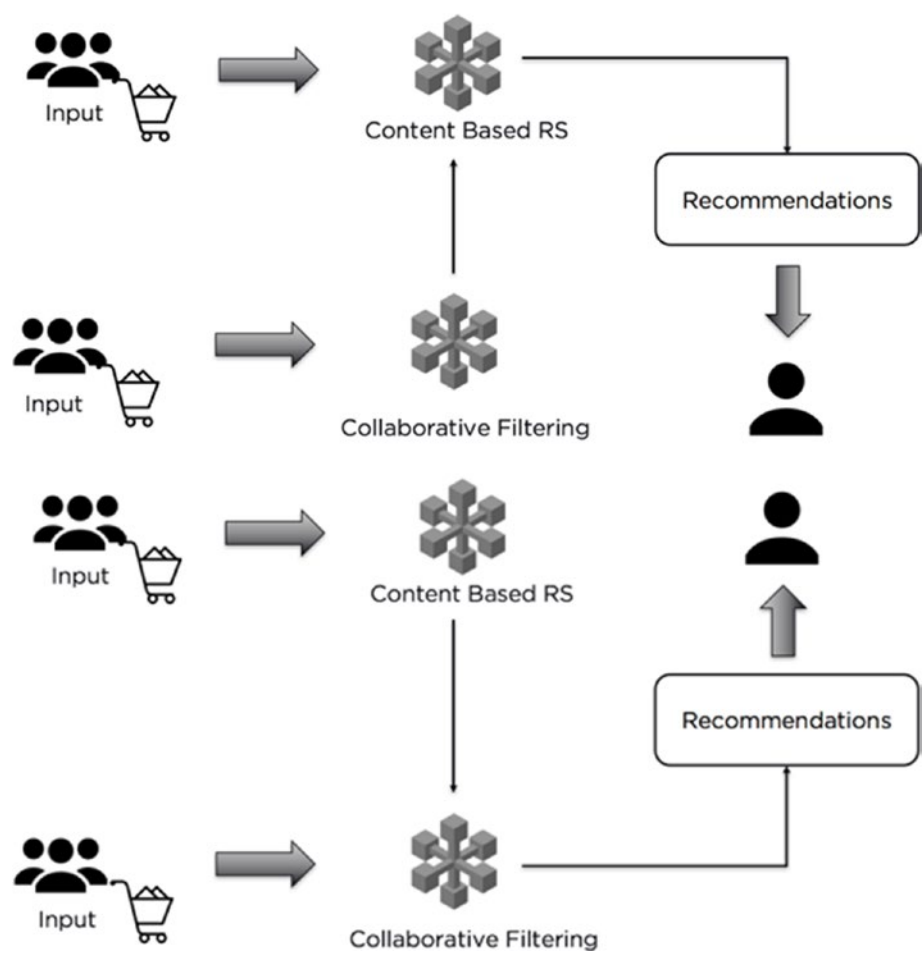


Figure 7-6. *Hybrid Recommendations*

Hybrid recommendations also include using other types of recommendations such as demographic based and knowledge based to enhance the performance of its recommendations. Hybrid RS have become integral parts of various businesses to help their users consume the right content, therefore deriving a lot of value.

Code

This section of the chapter focuses on building an RS from scratch using the ALS method in PySpark and Jupyter Notebook.

Note The complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.0 and higher versions.

Let's build a recommender model using Spark's MLlib library and predict the rating of an item for any given user.

Data Info

The dataset that we are going to use for this chapter is a subset from a famous open sourced movie lens dataset and contains a total of 0.1 million records with three columns (User_Id,title,rating). We will train our recommender model using 75% of the data and test it on the rest of the 25% user ratings.

Step 1: Create the SparkSession Object

We start the Jupyter Notebook and import SparkSession and create a new SparkSession object to use Spark:

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('lin_reg').
      getOrCreate()
```

Step 2: Read the Dataset

We then load and read the dataset within Spark using a dataframe. We have to make sure we have opened the PySpark from the same directory folder where the dataset is available or else we have to mention the directory path of the data folder.

```
[In]:  
df=spark.read.csv('movie_ratings_df.csv',inferSchema=True,  
header=True)
```

Step 3: Exploratory Data Analysis

In this section, we explore the dataset by viewing the dataset, validating the shape of the dataset, and getting a count of the number of movies rated and the number of movies that each user rated.

```
[In]: print((df.count(), len(df.columns)))  
[Out]: (100000,3)
```

So, the above output confirms the size of our dataset and we can then validate the datatypes of the input values to check if we need to change/cast any columns' datatypes.

```
[In]: df.printSchema()  
[Out]: root  
|-- userId: integer (nullable = true)  
|-- title: string (nullable = true)  
|-- rating: integer (nullable = true)
```

There is a total of three columns out of which two are numerical and the title is categorical. The critical thing with using PySpark for building RS is that we need to have user_id and item_id in numerical form. Hence, we will convert the movie title to numerical values later. We now view a few rows of the dataframe using the rand function to shuffle the records in random order.


```
[In]: df.orderBy(rand()).show(10,False)
```

```
[Out]:
```

userId	title	rating
13	Liar Liar (1997)	2
741	Cape Fear (1991)	4
916	Return of the Jedi (1983)	4
698	Birdcage, The (1996)	2
682	Primal Fear (1996)	3
144	Empire Strikes Back, The (1980)	4
887	Willy Wonka and the Chocolate Factory (1971)	5
389	Before Sunrise (1995)	4
370	Dante's Peak (1997)	2
138	Truth About Cats & Dogs, The (1996)	4

```
only showing top 10 rows
```

```
[In]: df.groupBy('userId').count().orderBy('count',
      ascending=False).show(10,False)
```

```
[Out]:
```

userId	count
405	737
655	685
13	636
450	540
276	518
416	493
537	490
303	484
234	480
393	448

```
only showing top 10 rows
```

```
[In]: df.groupby('userId').count().orderBy('count',
      ascending=True).show(10,False)
[Out]:
```

userId	count
732	20
631	20
636	20
926	20
93	20
300	20
572	20
596	20
685	20
34	20

only showing top 10 rows

The user with the highest number of records has rated 737 movies, and each user has rated at least 20 movies.

```
[In]: df.groupby('title').count().orderBy('count',
      ascending=False).show(10,False)
[Out]:
```

title	count
Star Wars (1977)	583
Contact (1997)	509
Fargo (1996)	508
Return of the Jedi (1983)	507
Liar Liar (1997)	485
English Patient, The (1996)	481
Scream (1996)	478
Toy Story (1995)	452
Air Force One (1997)	431
Independence Day (ID4) (1996)	429

only showing top 10 rows

The movie with highest number of ratings is *Star Wars* (1977) and has been rated 583 times, and each movie has been rated by at least by 1 user.

Step 4: Feature Engineering

We now convert the movie title column from categorical to numerical values using `StringIndexer`. We import the `stringIndexer` and `IndexToString` from the PySpark library.

```
[In]: from pyspark.sql.functions import *  
[In]: from pyspark.ml.feature import StringIndexer,  
      IndexToString
```

Next, we create the `stringindexer` object by mentioning the input column and output column. Then we fit the object on the dataframe and apply it on the movie title column to create new dataframe with numerical values.

```
[In]: stringIndexer = StringIndexer(inputCol="title",  
      outputCol="title_new")
```

```
[In]: model = stringIndexer.fit(df)
```

```
[In]: indexed = model.transform(df)
```

Let's validate the numerical values of the title column by viewing few rows of the new dataframe (`indexed`).

```
[In]: indexed.show(10)  
[Out]:
```

```
+-----+-----+-----+-----+
|userId|          title|rating|title_new|
+-----+-----+-----+-----+
|  932|   Cape Fear (1991)|    3|   161.0|
|  721|  Piano, The (1993)|    3|   173.0|
|  642|Low Down Dirty Sh...|    2|  1115.0|
|  798|That Darn Cat! (1...|    4|   686.0|
|  535|African Queen, Th...|    4|   199.0|
|  765|Stealing Beauty (...|    5|   521.0|
|  927|Poison Ivy II (1995)|    3|  1041.0|
|  544|   G.I. Jane (1997)|    3|   152.0|
|  788|Godfather: Part I...|    4|   108.0|
|  706|Birdcage, The (1996)|    4|    43.0|
+-----+-----+-----+-----+
only showing top 10 rows
```

As we can see, we now we have an additional column (title_new) with numerical values representing the movie titles. We have to repeat the same procedure in case the user_id is also a categorical type. Just to validate the movie counts, we rerun the groupBy on a new dataframe.

```
[In]: indexed.groupBy('title_new').count().orderBy('count',
         ascending=False).show(10,False)
[Out]:
```

```
+-----+-----+
|title_new|count|
+-----+-----+
| 0.0     | 583 |
| 1.0     | 509 |
| 2.0     | 508 |
| 3.0     | 507 |
| 4.0     | 485 |
| 5.0     | 481 |
| 6.0     | 478 |
| 7.0     | 452 |
| 8.0     | 431 |
| 9.0     | 429 |
+-----+-----+
only showing top 10 rows
```

Step 5: Splitting the Dataset

Now that we have prepared the data for building the recommender model, we can split the dataset into training and test sets. We split it into a 75 to 25 ratio to train the model and test its accuracy.

```
[In]: train,test=indexed.randomSplit([0.75,0.25])
```

```
[In]: train.count()
```

```
[Out]: 75104
```

```
[In]: test.count()
```

```
[Out]: 24876
```

Step 6: Build and Train Recommender Model

We import the ALS function from the PySpark ml library and build the model on the training dataset. There are multiple hyperparameters that can be tuned to improve the performance of the model. Two of the important ones are `nonnegative='True'` doesn't create negative ratings in recommendations and `coldStartStrategy='drop'` to prevent any NaN ratings predictions.

```
[In]: from pyspark.ml.recommendation import ALS
```

```
[In]: rec=ALS(maxIter=10,regParam=0.01,userCol='userId',  
             itemCol='title_new',ratingCol='rating',nonnegative=True,  
             coldStartStrategy="drop")
```

```
[In]: rec_model=rec.fit(train)
```

Step 7: Predictions and Evaluation on Test Data

The final part of the entire exercise is to check the performance of the model on unseen or test data. We use the transform function to make predictions on the test data and RegressionEvaluate to check the RMSE value of the model on test data.

```
[In]: predicted_ratings=rec_model.transform(test)
```

```
[In]: predicted_ratings.printSchema()
```

```
root
```

```
-- userId: integer (nullable = true)
-- title: string (nullable = true)
-- rating: integer (nullable = true)
-- title_new: double (nullable = false)
-- prediction: float (nullable = false)
```

```
[In]: predicted_ratings.orderBy(rand()).show(10)
```

```
[Out]:
```

```
+-----+-----+-----+-----+-----+
|userId|          title|rating|title_new|prediction|
+-----+-----+-----+-----+-----+
|    92|Tie Me Up! Tie Me...|    4|    766.0|  3.1512196|
|   222|    Batman (1989)|    3|    116.0|  3.503284|
|   178|Beauty and the Be...|    4|    114.0|  4.1487904|
|   303|Jerry Maguire (1996)|    5|     15.0|  4.348913|
|   134|    Flubber (1997)|    2|    579.0|  2.5635276|
|   295|    Henry V (1989)|    4|    268.0|  4.2598643|
|   889|Adventures of Pri...|    2|    305.0|  2.9040515|
|   374|Men in Black (1997)|    3|     31.0|  3.602631|
|   559|Killing Fields, T...|    4|    276.0|  4.55797|
|   290|Star Trek: The Mo...|    1|    286.0|  3.2992659|
+-----+-----+-----+-----+-----+
only showing top 10 rows
```

```
[xIn]: from pyspark.ml.evaluation import RegressionEvaluator
```

```
[In]: evaluator=RegressionEvaluator(metricName='rmse',
    predictionCol='prediction',labelCol='rating')
```

```
[In]: rmse=evaluator.evaluate(predictions)
```

```
[In] : print(rmse)
```

```
[Out]: 1.0293574739493354
```

The RMSE is not very high; we are making an error of one point in the actual rating and predicted rating. This can be improved further by tuning the model parameters and using the hybrid approach.

Step 8: Recommend Top Movies That Active User Might Like

After checking the performance of the model and tuning the hyperparameters, we can move ahead to recommend top movies to users that they have not seen and might like. The first step is to create a list of unique movies in the dataframe.

```
[In]: unique_movies=indexed.select('title_new').distinct()
```

```
[In]: unique_movies.count()
```

```
[Out]: 1664
```

So, we have in total 1,664 distinct movies in the dataframe.

```
[In]: a = unique_movies.alias('a')
```

We can select any user within the dataset for which we need to recommend other movies. In our case, we go ahead with `userId = 85`.

```
[In]: user_id=85
```

We will filter the movies that this active user has already rated or seen.

```
[In]: watched_movies=indexed.filter(indexed['userId'] ==
      user_id).select('title_new').distinct()

[In]: watched_movies.count()
[Out]: 287

[In]: b=watched_movies.alias('b')
```

So, there are total of 287 unique movies out of 1,664 movies that this active user has already rated. So, we would want to recommend movies from the remaining 1,377 items. We now combine both the tables to find the movies that we can recommend by filtering null values from the joined table.

```
[In]: total_movies = a.join(b, a.title_new == b.title_new,how='left')

[In]: total_movies.show(10,False)

[Out]:
```

title_new	title_new
299.0	null
558.0	null
305.0	305.0
596.0	null
1051.0	null
934.0	null
496.0	496.0
769.0	null
692.0	null
720.0	null

only showing top 10 rows


```
[In]: remaining_movies=total_movies.where(col("b.title_new").
      isNull()).select(a.title_new).distinct()
```

```
[In]: remaining_movies.count()
```

```
[Out]: 1377
```

```
[In]: remaining_movies=remaining_movies.withColumn("userId",lit
      (int(user_id)))
```

```
[In]: remaining_movies.show(10,False)
```

```
[Out]:
```

title_new	userId
299.0	85
558.0	85
596.0	85
1051.0	85
934.0	85
769.0	85
692.0	85
720.0	85
576.0	85
810.0	85

only showing top 10 rows

Finally, we can now make the predictions on this remaining movie's dataset for the active user using the recommender model that we built earlier. We filter only a few top recommendations that have the highest predicted ratings.

```
[In]: recommendations=rec_model.transform(remaining_movies).
      orderBy('prediction',ascending=False)
```

```
[In]: recommendations.show(5,False)
```

```
[Out]:
```

title_new	userId	prediction
1433.0	85	4.9689837
1322.0	85	4.6927013
1271.0	85	4.605163
1470.0	85	4.5409293
705.0	85	4.532007

So, movie titles 1433 and 1322 have the highest predicted rating for this active user (85). We can make it more intuitive by adding the movie title back to the recommendations. We use `IndexToString` function to create an additional column that returns the movie title.

```
[In]:
    movie_title = IndexToString(inputCol="title_new",
                               outputCol="title",labels=model.labels)
[In]: final_recommendations=movie_title.
      transform(recommendations)
```

```
[In]: final_recommendations.show(10,False)
```

[Out]:

title_new	userId	prediction	title
1433.0	85	4.9689837	Boys, Les (1997)
1322.0	85	4.6927013	Faust (1994)
1271.0	85	4.605163	Whole Wide World, The (1996)
1470.0	85	4.5409293	Some Mother's Son (1996)
705.0	85	4.532007	Laura (1944)
303.0	85	4.5236835	Close Shave, A (1995)
1121.0	85	4.4936523	Crooklyn (1994)
1195.0	85	4.4636283	Pather Panchali (1955)
285.0	85	4.456875	Wrong Trousers, The (1993)
638.0	85	4.4495435	Shall We Dance? (1996)

only showing top 10 rows

So, the recommendations for the `userId` (85) are *Boys, Les* (1997) and *Faust* (1994). This can be nicely wrapped in a single function that executes the above steps in sequence and generates recommendations for active users. The complete code is available on the GitHub repo with this function built in.

Conclusion

In this chapter, we went over various types of recommendation models along with the strengths and limitations of each. We then created a collaborative filtering based recommender system in PySpark using the ALS method to recommend movies to the users.