CHAPTER 10

Analyzing Music Trends and Recommendations

Recommendation engines are probably one of the most popular and well known Machine Learning applications. A lot of people who don't belong to the Machine Learning community often assume that recommendation engines are its only use. Although we know that Machine Learning has a vast subspace where recommendation engines are just one of the candidates, there is no denying the popularity of recommendation engines. One of the reasons for their popularity is their ubiquitous nature; anyone who is online, in any way, has been in touch with a recommendation engine in some form or the other. They are used for recommending products on ecommerce sites, travel destinations on travel portal, songs/videos on streaming sites, restaurants on food aggregator portals, etc. A long list underlines their universal application.

The popularity of recommendation engines stems from two very important points about them:

- *They are easy to implement:* Recommendation engines are easy to integrate in an already existing workflow. All we need is to collect some data regarding our user trends and patterns, which normally can be extracted from the business' transactional database.
- *They work:* This statement is equally true for all the other Machine Learning solutions that we have discussed. But an important distinction comes from the fact that they have a very limited downside. For example, consider a travel portal, which suggests a set of most popular locations from its dataset. The recommendation system can be a trivial one but its mere presence in front of the user is likely to generate user interest. The firm can definitely gain by working a sophisticated recommendation engine but even a very simple one is guaranteed to pay some dividends with minimal investments. This point makes them a very attractive proposition.

This chapter studies how we can use transactional data to develop different types of recommendation engines. We will learn about an auxiliary dataset of a very interesting dataset, "The million song dataset". We will go through user listening history and then use it to develop multiple recommendation engines with varying levels of sophistication.

D. Sarkar et al., *Practical Machine Learning with Python*, https://doi.org/10.1007/978-1-4842-3207-1_10

The Million Song Dataset Taste Profile

The million song dataset is a very popular dataset and is available at https://labrosa.ee.columbia.edu/millionsong/. The original dataset contained quantified audio features of around a million songs ranging over multiple years. The dataset was created as a collaborative project between The Echonest (http://the.echonest.com/) and LABRosa (http://labrosa.ee.columbia.edu/). Although we will not be using this dataset directly, we will be using some parts of it.

Several other datasets were spawned from the original million song dataset. One of those datasets was called The Echonest Taste Profile Subset. This particular dataset was created by The Echonest with some undisclosed partners. The dataset contains play counts by anonymous users for songs contained in the million songs dataset. The taste profile dataset is quite big, as it contains around 48 million lines of triplets. The triplets contain the following information:

(*user id, song id, play counts*)

Each row gives the play counts of a song identified by the song ID for the user identified by the user ID. The overall dataset contains around a million unique users and around 384,000 songs from the million song dataset are contained in it.

The readers can download the dataset from http://labrosa.ee.columbia.edu/millionsong/sites/default/files/challenge/train_triplets.txt.zip. The size of the compressed dataset is around 500MB and, upon decompression, you need a space of around 3.5GB. Once you have the data downloaded and uncompressed, you will see how to subset the dataset to reduce its size.

■ The million song dataset has several other useful auxiliary datasets. We will not be covering them in detail here, but we encourage the reader to explore these datasets and use their imagination for developing innovative use cases.

Exploratory Data Analysis

Exploratory data analysis is an important part of any data analysis workflow. By this time, we have established this fact firmly in the mindset of our readers. The exploratory data analysis becomes even more important in the cases of large datasets, as this will often lead us to information that we can use to trim down the dataset a little. As we will see, sometimes we will also go beyond the traditional data access tools to bypass the problems posed by the large size of data.

Loading and Trimming Data

The first step in the process is loading the data from the uncompressed files. As the data size is around 3GB, we will not load the complete data but we will only load a specified number of rows from the dataset. This can be achieved by using the `nrows` parameter of the `read_csv` function provided by `pandas`.

```
In [2]: triplet_dataset = pd.read_csv(filepath_or_buffer=data_home+'train_triplets.txt',
   ...: nrows=10000,sep='\t', header=None, names=['user','song','play_count'])
```

Since the dataset doesn't have a header, we also provided the column name to the function. A subset of the data is shown in Figure 10-1.

	user	song	play_count
0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOAKIMP12A8C130995	1
1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOAPDEY12A81C210A9	1
2	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBBMDR12A8C13253B	2
3	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBFNSP12AF72A0E22	1
4	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBFOVM12A58A7D494	1
5	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBNZDC12A6D4FC103	1
6	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBSUJE12A6D4F8CF5	2
7	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBVFZR12A6D4F8AE3	1
8	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBXALG12A8C13C108	1
9	b80344d063b5ccb3212f76538f3d9e43d87dca9e	SOBXHDL12A81C204C0	1

Figure 10-1. *Sample rows from The Echonest taste profile dataset*

The first thing we may want to do in the dataset of this size is determine how many unique users (or songs) we should consider. In the original dataset, we have around a million users, but we want to determine the number of users we should consider. For example, if 20% of all the users account for around 80% of total play counts, then it would be a good idea to focus our analysis on those 20% users. Usually this can be done by summarizing the dataset by users (or by songs) and getting a cumulative sum of the play counts. Then we can find out how many users account for 80% of the play counts, etc. But due to the size of the data, the cumulative summation function provided by pandas will run into trouble. So we will write code to read the file line by line and extract the play count information on a user (or song). This will also serve as a possible method that readers can use in case the dataset size exceeds the memory available on their systems. The code snippet that follows will read the file line by line, extract total play count of all the users, and persist that information for later use.

```
In [2]: output_dict = {}
   ...: with open(data_home+'train_triplets.txt') as f:
   ...: for line_number, line in enumerate(f):
   ...:       user = line.split('\t')[0]
   ...:       play_count = int(line.split('\t')[2])
   ...:       if user in output_dict:
   ...:           play_count +=output_dict[user]
   ...:           output_dict.update({user:play_count})
   ...:       output_dict.update({user:play_count})
   ...: output_list = [{'user':k,'play_count':v} for k,v in output_dict.items()]
   ...: play_count_df = pd.DataFrame(output_list)
   ...: play_count_df = play_count_df.sort_values(by = 'play_count', ascending = False)
   ...: play_count_df.to_csv(path_or_buf='user_playcount_df.csv', index = False)
```

The persisted dataframe can be then loaded and used based on our requirements. We can use a similar strategy to extract play counts for each of the songs. A few lines from the dataset are shown in Figure 10-2

	play_count	user
0	13132	093cb74eb3c517c5179ae24caf0ebec51b24d2a2
1	9884	119b7c88d58d0c6eb051365c103da5caf817bea6
2	8210	3fa44653315697f42410a30cb766a4eb102080bb
3	7015	a2679496cd0af9779a92a13ff7c6af5c81ea8c7b
4	6494	d7d2d888ae04d16e994d6964214a1de81392ee04
5	6472	4ae01afa8f2430ea0704d502bc7b57fb52164882
6	6150	b7c24f770be6b802805ac0e2106624a517643c17
7	5656	113255a012b2affeab62607563d03fbdf31b08e7
8	5620	6d625c6557df84b60d90426c0116138b617b9449
9	5602	99ac3d883681e21ea68071019dba828ce76fe94d

Figure 10-2. *Play counts for some users*

The first thing we want to find out about our dataset is the number of users that we will need to account for around 40% of the play counts. We have arbitrarily chosen a value of 40% to keep the dataset size manageable; you can experiment with these figures to get different sized datasets and even leverage big data processing and analysis frameworks like Spark on top of Hadoop to analyze the complete dataset! The following code snippet will determine the subset of users that account for this percentage of data. In our case around 10,0000 users account for 40% of play counts, hence we will subset those users.

```
In [2]: total_play_count = sum(song_count_df.play_count)
   ...: (float(play_count_df.head(n=100000).play_count.sum())/total_play_count)*100
   ...: play_count_subset = play_count_df.head(n=100000)
```

In similar way, we can determine the number of unique songs required to explain 80% of the total play count. In our case, we will find that 30,000 songs account for around 80% of the play count. This information is already a great find, as around 10% of the songs are contributing to 80% of the play count. Using a code snippet similar to one given previously, we can determine the subset of such songs. With these songs and user subsets, we can subset our original dataset to reduce the dataset to contain only filtered users and songs. The code snippet that follows uses these dataframes to filter the original dataset and then persists the resultant dataset for future uses.

```
In [2]: triplet_dataset =
   ...: pd.read_csv(filepath_or_buffer=data_home+'train_triplets.txt',sep='\t', header=None,
   ...: names=['user','song','play_count'])
   ...: triplet_dataset_sub = triplet_dataset[triplet_dataset.user.isin(user_subset) ]
```

```
   ...: del(triplet_dataset)
   ...: triplet_dataset_sub_song =
   ...: triplet_dataset_sub[triplet_dataset_sub.song.isin(song_subset)]
   ...: del(triplet_dataset_sub)
   ...: triplet_dataset_sub_song.to_csv(path_or_buf=data_home+'triplet_dataset_sub_song.
        csv', index = False)
```

This subsetting will give us a dataframe with around 10 million rows of tuples. We will use this as the starting dataset for our all future analyses. You can play around with these numbers to arrive at different datasets and possibly different results.

Enhancing the Data

The data we loaded is just the triplet data so we are not able to see the song name, the artist name, or the album names. We can enhance our data by adding this information about the songs. This information is part of the million song database. This data is provided as a SQLite database file. First we will download the data by downloading the `track_metadata.db` file from the web page at https://labrosa.ee.columbia.edu/millionsong/pages/getting-dataset#subset.

The next step is to read this SQLite database to a dataframe and extract track information by merging it with our triplet dataframe. We will also drop some extra columns that we won't be using for our analysis. The code snippet that follows will load the entire dataset, join it with our subsetted triplet data, and drop the extra columns.

```
In [2]: conn = sqlite3.connect(data_home+'track_metadata.db')
   ...: cur = conn.cursor()
   ...: cur.execute("SELECT name FROM sqlite_master WHERE type='table'")
   ...: cur.fetchall()

Out[2]: [('songs',)]
```

The output of the above snippet shows that the database contains a table named `songs`. We will get all the rows from this table and read it into a dataframe.

```
In [5]: del(track_metadata_df_sub['track_id'])
   ...: del(track_metadata_df_sub['artist_mbid'])
   ...: track_metadata_df_sub = track_metadata_df_sub.drop_duplicates(['song_id'])
   ...: triplet_dataset_sub_song_merged = pd.merge(triplet_dataset_sub_song, track_metadata_
        df_sub, how='left', left_on='song', right_o
   ...: n='song_id')
   ...: triplet_dataset_sub_song_merged.rename(columns={'play_count':'listen_
        count'},inplace=True)
   ...: del(triplet_dataset_sub_song_merged['song_id'])
   ...: del(triplet_dataset_sub_song_merged['artist_id'])
   ...: del(triplet_dataset_sub_song_merged['duration'])
   ...: del(triplet_dataset_sub_song_merged['artist_familiarity'])
   ...: del(triplet_dataset_sub_song_merged['artist_hotttnesss'])
   ...: del(triplet_dataset_sub_song_merged['track_7digitalid'])
   ...: del(triplet_dataset_sub_song_merged['shs_perf'])
   ...: del(triplet_dataset_sub_song_merged['shs_work'])
```

The final dataset, merged with the triplets dataframe looks similar to the depiction in Figure 10-3. This will form the starting dataframe for our exploratory data analysis.

	user	song	listen_count	title	release	artist_name	year
0	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOADQPP12A67020C82	12	You And Me Jesus	Tribute To Jake Hess	Jake Hess	2004
1	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOAFTRR12AF72A8D4D	1	Harder Better Faster Stronger	Discovery	Daft Punk	2007
2	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOANQFY12AB0183239	1	Uprising	Uprising	Muse	0
3	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOAYATB12A6701FD50	1	Breakfast At Tiffany's	Home	Deep Blue Something	1993
4	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOBOAFP12A8C131F36	7	Lucky (Album Version)	We Sing. We Dance. We Steal Things.	Jason Mraz & Colbie Caillat	0
5	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOBONKR12A58A7A7E0	26	You're The One	If There Was A Way	Dwight Yoakam	1990
6	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOBZZDU12A6310D8A3	7	Don't Dream It's Over	Recurring Dream_ Best Of Crowded House (Domest...	Crowded House	1986
7	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOCAHRT12A8C13A1A4	5	S.O.S.	SOS	Jonas Brothers	2007
8	d6589314c0a9bcbca4fee0c93b14bc402363afea	SODASIJ12A6D4F5D89	1	The Invisible Man	The Invisible Man	Michael Cretu	1985
9	d6589314c0a9bcbca4fee0c93b14bc402363afea	SODEAWL12AB0187032	8	American Idiot [feat. Green Day & The Cast Of ...	The Original Broadway Cast Recording 'American...	Green Day	0

Figure 10-3. *Play counts dataset merged with songs metadata*

Visual Analysis

Before we start developing various recommendation engines, let's do some visual analysis of our dataset. We will try to see what the different trends are regarding the songs, albums, and releases.

Most Popular Songs

The first information that we can plot for our data concerns the popularity of the different songs in the dataset. We will try to determine the top 20 songs in our dataset. A slight modification of this popularity will also serve as our most basic recommendation engine.

The following code snippet gives us the most popular songs from our dataset.

```
In [7]: import matplotlib.pyplot as plt; plt.rcdefaults()
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
   ...: popular_songs = triplet_dataset_sub_song_merged[['title','listen_count']].
        groupby('title').sum().reset_index()
   ...: popular_songs_top_20 = popular_songs.sort_values('listen_count', ascending=False).
        head(n=20)
   ...: objects = (list(popular_songs_top_20['title']))
   ...: y_pos = np.arange(len(objects))
   ...: performance = list(popular_songs_top_20['listen_count'])
   ...:
   ...: plt.bar(y_pos, performance, align='center', alpha=0.5)
   ...: plt.xticks(y_pos, objects, rotation='vertical')
   ...: plt.ylabel('Item count')
   ...: plt.title('Most popular songs')
   ...: plt.show()
```

The plot that's generated by the code snippet is shown in Figure 10-4. The plot shows that the most popular song of our dataset is "You're the One". We can also search through our track dataframe to see that the band responsible for that particular track is The Black Keys.

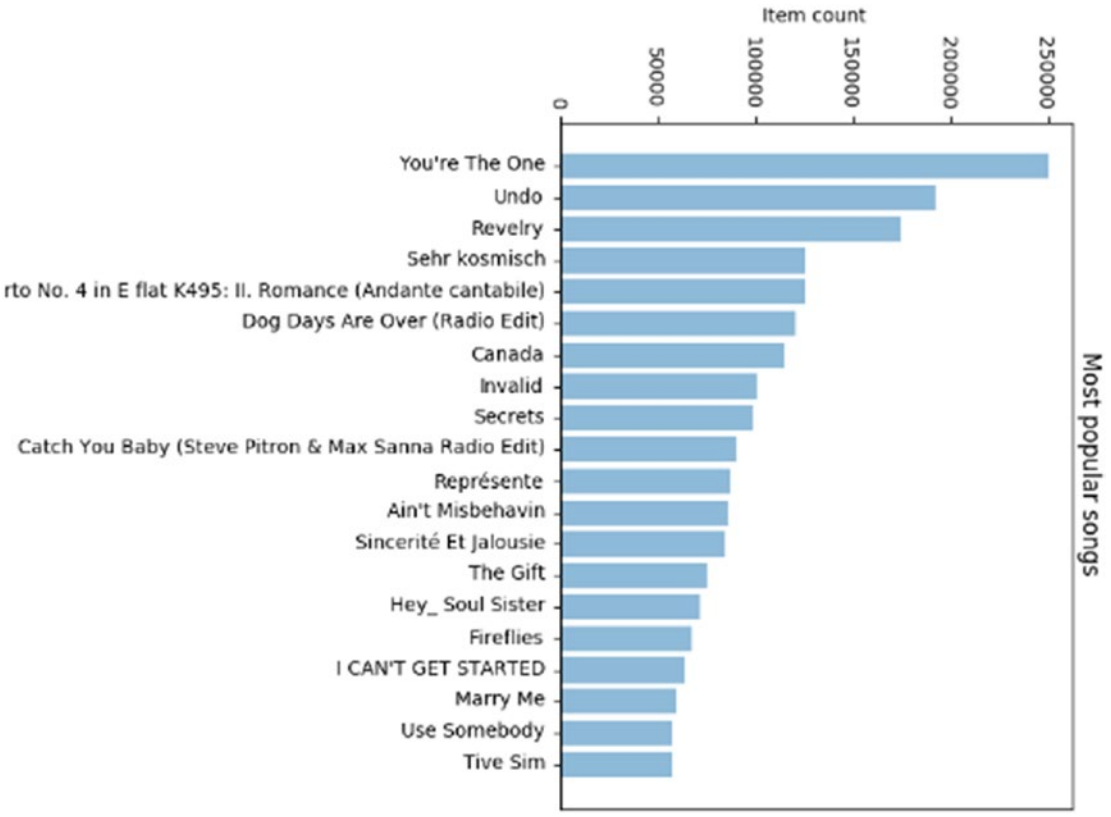

Figure 10-4. *Most popular songs*

Most Popular Artist

The next information the readers may be interested in is, who are the most popular artists in the dataset? The code to plot this information is quite similar to the code given previously, so we will not include the exact code snippet. The resultant graph is shown in Figure 10-5.

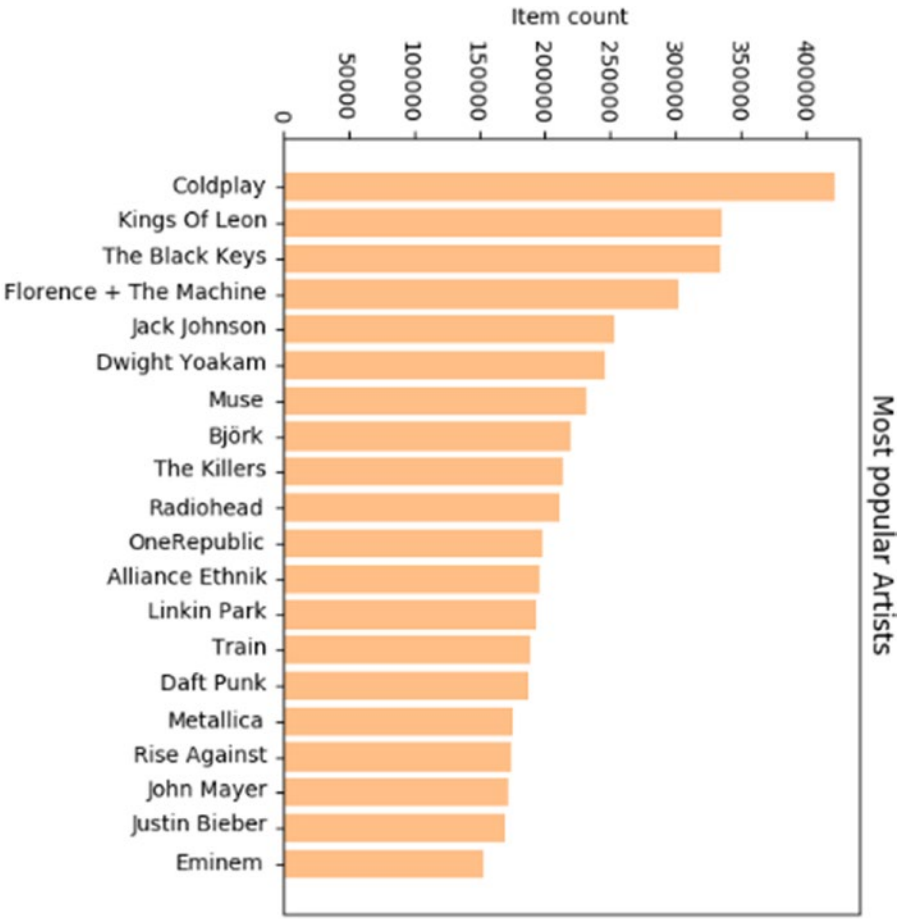

Figure 10-5. *Most popular artists*

We can read the plot to see that Coldplay is one of the most popular artists according to our dataset. A keen music enthusiast can see that we don't have a lot of representation from the classic artists like U2 or The Beatles, except for maybe Metallica and Radiohead. This underlines two key points—first, the data is mostly sourced from the generation that's not just always listening to the classic artists online and second, that very rarely we have an artist that's not of the present generation but still scores high when it comes to online plays. Surprisingly the only example of such behavior in our dataset is Metallica and Radiohead. They have their origins in the 1980s but are still pretty popular when it comes to online play counts. Diverse music genre representations are depicted however in the top artists with popular rap artists like Eminem, alternative rock bands like Linkin Park and The Killers and even pop\rock bands like Train and OneRepublic, besides classic rock or metal bands like Radiohead and Metallica!

Another slightly off reading is that Coldplay is the most popular artist, but they don't have a candidate in the most popular songs list. This indirectly hints at an even distribution of those play counts across all of their tracks. You can take it as exercise to determine song play distribution for each of the artists who appear in the plot in Figure 10-5. This will give you a clue as to whether the artist holds a skewed or uniform popularity. This idea can be further developed to a full-blown recommendation engine.

User vs. Songs Distribution

The last information that we can seek from our dataset is regarding the distribution of song count for users. This information will tell us how the number of songs that users listen to on average are distributed. We can use this information to create different categories of users and modify the recommendation engine on that basis. The users who are listening to a very select number of songs can be used for developing simple recommendation engines and the users who provide us lots of insight into their behavior can be candidates for developing complex recommendation engines.

Before we go on to plot that distribution, let's try to find some statistical information about that distribution. The following code calculates that distribution and then shows summary statistics about it.

```
In [11]: user_song_count_distribution = triplet_dataset_sub_song_merged[['user','title']].
groupby('user').count().reset_index().sort_values(by='title', ascending = False)
    ...: user_song_count_distribution.title.describe()
Out[11]:
count    99996.000000
mean       107.752160
std         79.741555
min          1.000000
25%         53.000000
50%         89.000000
75%        141.000000
max       1189.000000
Name: title, dtype: float64
```

This gives us some important information about how the song counts are distributed across the users. We see that on an average, a user will listen to 100+ songs, but some users have a more voracious appetite for song diversification. Let's try to visualize this particular distribution. The code that follows will help us plot the distribution of play counts across our dataset. We have intentionally kept the number of bins to a small amount, as that can give approximate information about the number of classes of users we have.

```
In [12]: x = user_song_count_distribution.title
    ...: n, bins, patches = plt.hist(x, 50, facecolor='green', alpha=0.75)
    ...: plt.xlabel('Play Counts')
    ...: plt.ylabel('Probability')
    ...: plt.title(r'$\mathrm{Histogram\ of\ User\ Play\ Count\ Distribution}\ $')
    ...: plt.grid(True)
    ...: plt.show()
```

The distribution plot generated by the code is shown in Figure 10-6. The image clearly shows that, although we have a huge variance in the minimum and maximum play count, the bulk of the mass of the distribution is centered on 100+ song counts.

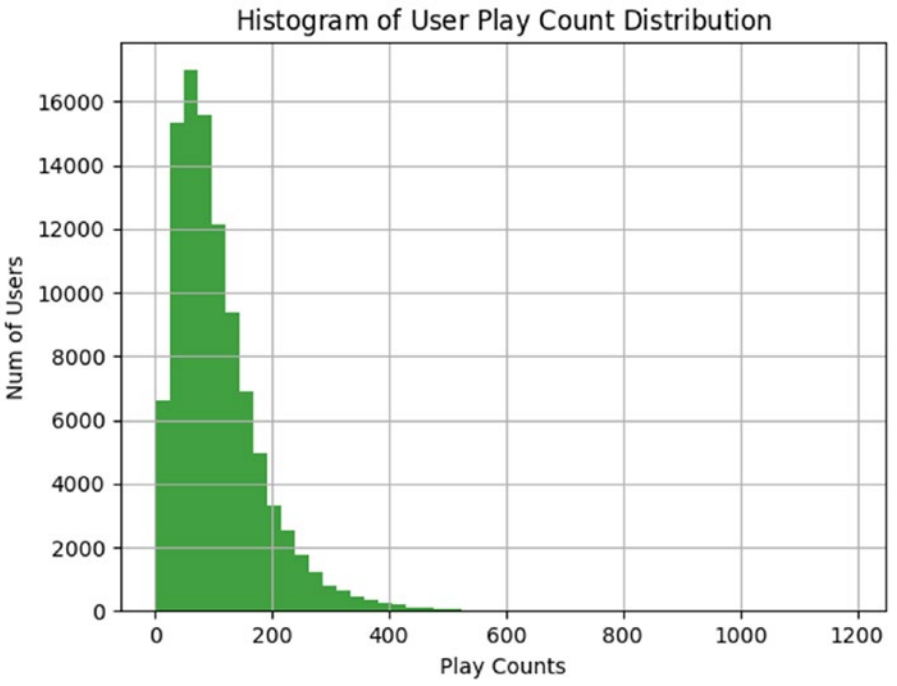

Figure 10-6. *Distribution of play counts for users*

Given the nature of the data, we can perform a large variety of cool visualizations; for example, plotting how the top tracks of artists are played, the play count distribution on a yearly basis, etc. But we believe by now you are sufficiently skilled in both the art of asking pertinent questions and answering them through visualizations. So we will conclude our exploratory data analysis and move on to the major focus of the chapter, which is development of recommendation engines. But feel free to try out additional analysis and visualizations on this data and if you find out something cool, as always, feel free to send across a pull request to the book's code repository!

Recommendation Engines

The work of a recommendation engine is succinctly captured in its name—all it needs to do is make recommendations. But we must admit that this description is deceptively simple. Recommendation engines are a way of modeling and rearranging information available about user preferences and then using this information to provide informed recommendations on the basis of that information. The basis of the recommendation engine is always the recorded interaction between the users and products. For example, a movie recommendation engine will be based on the ratings provided to different movies by the users; a news article recommender will take into account the articles the user has read in past; etc.

This section uses the user-song play count dataset to uncover different ways in which we can recommend new tracks to different users. We will start with a very basic system and try to evolve linearly into a sophisticated recommendation system. Before we go into building those systems, we will examine their utility and the various types of recommendation engines.

Types of Recommendation Engines

The major area of distinction in different recommendation engines comes from the entity that they assume is the most important in the process of generating recommendations. There are different options for choosing the central entity and that choice will determine the type of recommendation engine we will develop.

- **User-based recommendation engines:** In these types of recommendation engines, the user is the central entity. The algorithm will look for similarities among users and on the basis of those similarities will come up with the recommendation.
- **Content-based recommendation engines:** On the other end of the recommendation engine spectrum, we have the content-based recommendation engine. In these, the central entity is the content that we are trying to recommend; for example, in our case the entity will be songs we are trying to recommend. These algorithm will attempt to find features about the content and find similar content. Then these similarities will be used to make recommendations to the end users.
- **Hybrid-recommendation engines:** These types of recommendation engines will take into account both the features of the users and the content to develop recommendations. These are also sometimes termed as collaborative filtering recommendation engines as they "collaborate" by using the similarities of content as well as users. These are one of the most effective classes of recommendation engines, as they take the best features of both classes of recommendation engines.

Utility of Recommendation Engines

In the previous chapter, we discussed an important requirement of any organization, understanding the customer. This requirement is made more important for online businesses, which have almost no physical interaction with their customers. Recommendation engines provide wonderful opportunities to these organizations to not only understand their clientele but also to use that information to increase their revenues. Another important advantage of recommendation engines is that they potentially have very limited downside. The worst thing the user can do is not pay attention to the recommendation made to him. The organization can easily integrate a crude recommendation engine in its interaction with the users and then, on the basis of its performance, make the decision to develop a more sophisticated version. Although unverified claims are often made about the impact of recommendation engines on the sales of major online service providers like Netflix, Amazon, YouTube, etc., an interesting insight into their effectiveness is provided by several papers. We encourage you to read one such paper at `http://ai2-s2-pdfs.s3.amazonaws.com/ba21/7822b81c3c9449014cb92e197d8a6baa4914.pdf`. The study claims that a good recommendation engine tends to increase sales volume by around 35% and also leads customers to discover more products, which in turns adds to a positive customer experience.

Before we start discussing various recommendation engines, we would like to thank our friend and fellow Data Scientist, Author and Course Instructor, Siraj Raval for helping us with a major chunk of the code used in this chapter pertaining to recommendation engines as well as sharing his codebase for developing our recommendation engines (check out Siraj's GitHub at `https://github.com/llSourcell`). We would be modifying some of his code samples to develop the recommendation engines that we discuss in the subsequent sections. Interested readers can also check out Siraj's YouTube channel at `www.youtube.com/c/sirajology` where he makes excellent videos on machine learning, deep learning, artificial intelligence and other fun educational content.

Popularity-Based Recommendation Engine

The simplest recommendation engine is naturally the easiest to develop. As we can easily develop recommendation engines, this type of recommendation engine is a very straightforward one to develop. The driving logic of this recommendation engine is that if some item is liked (or listened to) by a vast majority of our user base, then it is a good idea to recommend that item to users who have not interacted with that item.

The code to develop this kind of recommendation is extremely easy and is effectively just a summarization procedure. To develop these recommendations, we will determine which songs in our dataset have the most users listening to them and then that will become our standard recommendation set for each user. The code that follows defines a function that will do this summarization and return the resultant dataframe.

```
In [1]:
def create_popularity_recommendation(train_data, user_id, item_id):
    #Get a count of user_ids for each unique song as recommendation score
    train_data_grouped = train_data.groupby([item_id]).agg({user_id: 'count'}).reset_index()
    train_data_grouped.rename(columns = {user_id: 'score'},inplace=True)

    #Sort the songs based upon recommendation score
    train_data_sort = train_data_grouped.sort_values(['score', item_id], ascending = [0,1])

    #Generate a recommendation rank based upon score
    train_data_sort['Rank'] = train_data_sort['score'].rank(ascending=0, method='first')

    #Get the top 10 recommendations
    popularity_recommendations = train_data_sort.head(20)
    return popularity_recommendations

In [2]: recommendations = create_popularity_recommendation(triplet_dataset_sub_song_
merged,'user','title')
In [3]: recommendations
```

We can use this function on our dataset to generate the top 10 recommendations to each of our users. The output of our plain vanilla recommendation system is shown in Figure 10-7. Here you can see that the recommendations are very similar to the list of the most popular songs that you saw in the last section, which is expected as the logic behind both is the same—only the output is different.

	title	score	Rank
19580	Sehr kosmisch	18629	1.0
5780	Dog Days Are Over (Radio Edit)	17636	2.0
27314	You're The One	16082	3.0
19542	Secrets	15139	4.0
18636	Revelry	14942	5.0
25070	Undo	14682	6.0
7531	Fireflies	13087	7.0
9641	Hey_ Soul Sister	12991	8.0
25216	Use Somebody	12790	9.0
9922	Horn Concerto No. 4 in E flat K495: II. Romanc...	12343	10.0
24291	Tive Sim	11825	11.0
3629	Canada	11591	12.0
23468	The Scientist	11534	13.0
4194	Clocks	11358	14.0
12136	Just Dance	11056	15.0

Figure 10-7. *Recommendation by the popularity recommendation engine*

Item Similarity Based Recommendation Engine

In the last section, we witnessed one of the simplest recommendation engines. In this section we deal with a slightly more complex solution. This recommendation engine is based on calculating similarities between a user's items and the other items in our dataset.

Before we proceed further with our development effort, let's describe how we plan to calculate "item-item" similarity, which is central to our recommendation engine. Usually to define similarity among a set of items, we need a feature set on the basis of which both items can be described. In our case it will mean features of the songs on the basis of which one song can be differentiated from another. Although as we don't have ready access to these features (or do we?), we will define the similarity in terms of the users who listen to these songs. Confused? Consider this mathematical formula, which should give you a little more insight into the metric.

$$similarity_{ij} = intersection(users_i, users_j)/union(users_i, users_j)$$

This similarity metric is known as the Jaccard index (https://en.wikipedia.org/wiki/Jaccard_index) and in our case we can use it to define the similarities between two songs. The basic idea remains that if two songs are being listened to by a large fraction of common users out of the total listeners, the two songs can be said to be similar to each other. On the basis of this similarity metric, we can define the steps that the algorithm will take to recommend a song to a user *k*.

1. Determine the songs listened to by the user *k*.
2. Calculate the similarity of each song in the user's list to those in our dataset, using the similarity metric defined previously.
3. Determine the songs that are most similar to the songs already listened to by the user.
4. Select a subset of these songs as recommendation based on the similarity score.

As the Step 2 can become a computation-intensive step when we have a large number of songs, we will subset our data to 5,000 songs to make the computation more feasible. We will select the most popular 5,000 songs so it is quite unlikely that we would miss out on any important recommendations.

```
In [4]:
song_count_subset = song_count_df.head(n=5000)
user_subset = list(play_count_subset.user)
song_subset = list(song_count_subset.song)
triplet_dataset_sub_song_merged_sub = triplet_dataset_sub_song_merged[triplet_dataset_sub_
song_merged.song.isin(song_subset)]
```

This code will subset our dataset to contain only most popular 5,000 songs. We will then create our similarity based recommendation engine and generate a recommendation for a random user. We leverage Siraj's Recommenders module here for the item similarity based recommendation system.

```
In [5]:
train_data, test_data = train_test_split(triplet_dataset_sub_song_merged_sub, test_size =
0.30, random_state=0)
is_model = Recommenders.item_similarity_recommender_py()
is_model.create(train_data, 'user', 'title')
user_id = list(train_data.user)[7]
user_items = is_model.get_user_items(user_id)
is_model.recommend(user_id)
```

The recommendations for random users are shown in Figure 10-8. Notice the stark difference from the popularity based recommendation engine. So this particular person is almost guaranteed to not like the most popular songs in our dataset.

	user_id	song	score	rank
0	2a2f776cbac6df64d6cb505e7e834e01684673b6	Meteor	0.099810	1
1	2a2f776cbac6df64d6cb505e7e834e01684673b6	Coda	0.093718	2
2	2a2f776cbac6df64d6cb505e7e834e01684673b6	Tuesday Moon	0.084476	3
3	2a2f776cbac6df64d6cb505e7e834e01684673b6	Tron	0.083682	4
4	2a2f776cbac6df64d6cb505e7e834e01684673b6	Acadian Coast	0.081797	5
5	2a2f776cbac6df64d6cb505e7e834e01684673b6	Love Letter To Japan	0.081042	6
6	2a2f776cbac6df64d6cb505e7e834e01684673b6	Heavy Water	0.080742	7
7	2a2f776cbac6df64d6cb505e7e834e01684673b6	Balloons (Single version)	0.079459	8
8	2a2f776cbac6df64d6cb505e7e834e01684673b6	Blackbirds	0.078346	9
9	2a2f776cbac6df64d6cb505e7e834e01684673b6	Diamond Dave	0.076680	10

Figure 10-8. *Recommendation by the item similarity based recommendation engine*

■ **Note** At the start of this section, we mentioned we don't readily have access to song's features that we can use to define similarity. As part of the million song database, we have those features available for each of the songs in the dataset. You are encouraged to replace this implicit similarity, based on common users, with the explicit similarity based on features of the song and see how the recommendations change.

Matrix Factorization Based Recommendation Engine

Matrix factorization based recommendation engines are probably the most used recommendation engines when it comes to implementing recommendation engines in production. In this section, we give an intuition-based introduction to matrix factorization based recommendation engines. We avoid going into a heavily mathematical discussion, as from a practitioner's perspective the intent is to see how we can leverage this to get valuable recommendations from real data.

Matrix factorization refers to identification of two or more matrices from an initial matrix, such that when these matrices are multiplied we get the original matrix. Matrix factorization can be used to discover latent features between two different kinds of entities. What are these latent features? Let's discuss that for a moment before we go for a mathematical explanation.

Consider for a moment why you like a certain song—the answer may range from the soulful lyrics, to catchy music, to it being melodious, and many more. We can try to explain a song in mathematical terms by measuring its beats, tempo, and other such features and then define similar features in terms of the user. For example, we can define that from a user's listening history we know that he likes songs with beats that are bit on the higher side, etc. Once we have consciously defined such "features," we can use them to find matches for a user based on some similarity criteria. But more often than not, the tough part in this process is defining these features, as we have no handbook for what will make a good feature. It is mostly based on domain experts and a bit of experimentation. But as you will see in this section, you can use matrix factorization to discover these latent features and they seem to work great.

The starting point of any matrix factorization-based method is the utility matrix, as shown in Table 10-1. The utility matrix is a matrix of `user X item` dimension in which each row represents a user and each column stands for an item.

Table 10-1. *Example of a Utility Matrix*

	Item 1	Item 2	Item 3	Item 4	Item 5
User A	2				5
User B		1		5	
User C	5	1			

Notice that we have a lot of missing values in the matrix; these are the items that the user hasn't rated, either because he hasn't watched it or because he doesn't want to watch it. We can right away guess, say Item 4 is a recommendation for User C because User B and User C don't like Item 2, so it is likely they may end up liking the same items, in this case Item 4.

The process of matrix factorization means finding out a low rank approximation of the utility matrix. So we want to break down the utility matrix U into two low rank matrices so that we can recreate the matrix U by multiplying those two matrices. Mathematically,

$$R = U * I^t$$

and

$$|R| = |U| * |I|$$

Here R is our original rating matrix, U is our user matrix, and I is our item matrix. Assuming the process helps us identify K latent features, our aim is to find two matrices X and Y such that their product (matrix multiplication) approximates R.

X = |U| x K matrix (A matrix with dimensions of num_users * factors)

Y = |P| x K matrix (A matrix with dimensions of factors * num_movies)

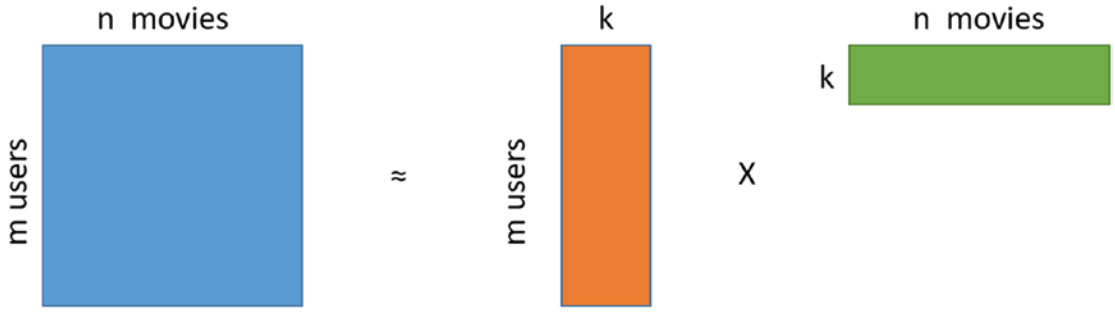

Figure 10-9. *Matrix Factorization*

We can also try to explain the concept of matrix factorization as an image. Based on Figure 10-9, we can regenerate the original matrix by multiplying the two matrices together. To make a recommendation to the user, we can multiply the corresponding user's row from the first matrix by the item matrix and determine the items from the row with maximum ratings. That will become our recommendations for the user. The first matrix represents the association between the users and the latent features, while the second matrix takes care of the associations between items (songs in our case) and the latent features. Figure 10-9 depicts a typical matrix factorization operation for a movie recommender system but the intent is to understand the methodology and extend it to build a music recommendation system in this scenario.

Matrix Factorization and Singular Value Decomposition

There are multiple algorithms available for determining factorization of any matrix. We use one of the simplest algorithms, which is the singular value decomposition or SVD. Remember that we discussed the mathematics behind SVD in Chapter 1. Here, we explain how the decomposition provided by SVD can be used as matrix factorization.

You may remember from Chapter 1 that singular value decomposition of a matrix returns three different matrices: U, S, and V. You can follow these steps to determine the factorization of a matrix using the output of SVD function.

- Factorize the matrix to obtain U, S, and V matrices.
- Reduce the matrix S to first k components. (The function we are using will only provide k dimensions, so we can skip this step.)
- Compute the square root of reduced matrix S_k to obtain the matrix $S_k^{1/2}$.
- Compute the two resultant matrix $U^*S_k^{1/2}$ and $S_k^{1/2}*V$ as these will serve as our two factorized matrices, as depicted in Figure 10-9.

We can then generate the prediction of user i for product j by taking the dot product of the i^{th} row of the first matrix with the j^{th} column of the second matrix. This information gives us all the knowledge required to build a matrix factorization based recommendation engine for our data.

Building a Matrix Factorization Based Recommendation Engine

After the discussion of the mechanics of matrix factorization based recommendation engines, let's try to create such a recommendation engine on our data. The first thing that we notice is that we have no concept of "rating" in our data; all we have are the play counts of various songs. This is a well known problem in the case of recommendation engines and is called the "implicit feedback" problem. There are many ways to solve this problem but we will look at a very simple and intuitive solution. We will replace the play count with a fractional play count. The logic being that this will measure the strength of "likeness" for a song in the range of [0,1]. We can argue about better methods to address this problem, but this is an acceptable solution to our problem. The following code will complete the task.

```
In [7]:
triplet_dataset_sub_song_merged_sum_df = triplet_dataset_sub_song_merged[['user','listen_
count']].groupby('user').sum().rese
t_index()
triplet_dataset_sub_song_merged_sum_df.rename(columns={'listen_count':'total_listen_
count'},inplace=True)
triplet_dataset_sub_song_merged = pd.merge(triplet_dataset_sub_song_merged,triplet_dataset_
sub_song_merged_sum_df)
triplet_dataset_sub_song_merged['fractional_play_count'] = triplet_dataset_sub_song_
merged['listen_count']/triplet_dataset_s
ub_song_merged['total_listen_count']
```

The modified dataframe is shown in Figure 10-10.

	user	song	listen_count	fractional_play_count
0	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOADQPP12A67020C82	12	0.036474
1	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOAFTRR12AF72A8D4D	1	0.003040
2	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOANQFY12AB0183239	1	0.003040
3	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOAYATB12A6701FD50	1	0.003040
4	d6589314c0a9bcbca4fee0c93b14bc402363afea	SOBOAFP12A8C131F36	7	0.021277

Figure 10-10. *Dataset with implicit feedback*

The next transformation of data that is required is to convert our dataframe into a `numpy` matrix in the format of utility matrix. We will convert our dataframe into a sparse matrix, as we will have a lot of missing values and sparse matrices are suitable for representation of such a matrix. Since we won't be able to transform our song IDs and user IDs into a numpy matrix, we will convert these indices into numerical indices. Then we will use these transformed indices to create our sparse numpy matrix. The following code will create such a matrix.

```
In [8]:
from scipy.sparse import coo_matrix
small_set = triplet_dataset_sub_song_merged
user_codes = small_set.user.drop_duplicates().reset_index()
song_codes = small_set.song.drop_duplicates().reset_index()
user_codes.rename(columns={'index':'user_index'}, inplace=True)
song_codes.rename(columns={'index':'song_index'}, inplace=True)
song_codes['so_index_value'] = list(song_codes.index)
user_codes['us_index_value'] = list(user_codes.index)
small_set = pd.merge(small_set,song_codes,how='left')
small_set = pd.merge(small_set,user_codes,how='left')
mat_candidate = small_set[['us_index_value','so_index_value','fractional_play_count']]
data_array = mat_candidate.fractional_play_count.values
row_array = mat_candidate.us_index_value.values
col_array = mat_candidate.so_index_value.values
data_sparse = coo_matrix((data_array, (row_array, col_array)),dtype=float)

In [8]: data_sparse
Out   : <99996x30000 sparse matrix of type '<class 'numpy.float64'>'
        with 10774785 stored elements in COOrdinate format>
```

Once we have converted our matrix into a sparse matrix, we will use the `svds` function provided by the `scipy` library to break down our utility matrix into three different matrices. We can specify the number of latent factors we want to factorize our data. In the example we will use 50 latent factors but users are encouraged to experiment with different values of latent factors and observe how the recommendation change as a result. The following code creates the decomposition of our matrix and predicts the recommendation for the same user as in the item similarity recommendation use case. We leverage our `compute_svd(...)` function to perform the SVD operation and then use the `compute_estimated_matrix(...)` for the low rank matrix approximation after factorization. Detailed steps with function implementations as always are present in the jupyter notebook.

```
In [9]:
K=50
#Initialize a sample user rating matrix
urm = data_sparse
MAX_PID = urm.shape[1]
MAX_UID = urm.shape[0]

#Compute SVD of the input user ratings matrix
U, S, Vt = compute_svd(urm, K)
uTest = [27513]

#Get estimated rating for test user
print("Predicted ratings:")
uTest_recommended_items = compute_estimated_matrix(urm, U, S, Vt, uTest, K, True)
for user in uTest:
    print("Recommendation for user with user id {}". format(user))
    rank_value = 1
    for i in uTest_recommended_items[user,0:10]:
        song_details = small_set[small_set.so_index_value == i].drop_duplicates('so_index_
        value')[['title','artist_name']]
        print("The number {} recommended song is {} BY {}".format(rank_value, list(song_
        details['title'])[0],list(song_details['artist_name'])[0]))
        rank_value+=1
```

The recommendations made by the matrix factorization based system are also shown in Figure 10-11. If you refer to the jupyter notebook for the code, you will observe that the user with ID 27513 is the same one for whom we performed the item similarity based recommendations earlier. Refer to the notebook for further details on how the different functions we used earlier have been implemented and used for our recommendations! Note how the recommendations have changed from the previous two systems. It looks like this user might like listening to some Coldplay and Radiohead, definitely interesting!

```
Recommendation for user with user id 27513
The number 1 recommended song is Behind The Sea [Live In Chicago] BY Panic At The Disco
The number 2 recommended song is Una Confusion BY LU
The number 3 recommended song is Home BY Edward Sharpe & The Magnetic Zeros
The number 4 recommended song is Dead Souls BY Nine Inch Nails
The number 5 recommended song is The City Is At War (Album Version) BY Cobra Starship
The number 6 recommended song is Tighten Up BY The Black Keys
The number 7 recommended song is Climbing Up The Walls BY Radiohead
The number 8 recommended song is Yellow BY Coldplay
The number 9 recommended song is Creep (Explicit) BY Radiohead
The number 10 recommended song is West One (Shine On Me) BY The Ruts
```

Figure 10-10. *Recommendation using the matrix factorization based recommender*

We used one of the simplest matrix factorization algorithms for our use case; you can try to find out other more sophisticated implementation of the matrix factorization routine, which will lead to a different recommendation system. Another topic which we have ignored a bit is the conversion of song play count into a measure of "implicit feedback". The system that we chose is acceptable but is far from perfect. There is a lot of literature that discusses the handling of this issue. We encourage you to find different ways of handling it and experiment with various measures!

A Note on Recommendation Engine Libraries

You might have noticed that we did not use any readily available packages for building our recommendation system. Like for every possible task, Python has multiple libraries available for building recommendation engines too. But we have refrained from using such libraries because we wanted to give you a taste of what goes on behind a recommendation engine. Most of these libraries will take the sparse matrix format as input data and allow you to develop recommendation engines. We encourage you to continue with your experimentation with at least one of those libraries, as it will give you an idea of exploring different possible implementations of the same problem and the differences those implementations can make. Some libraries that you can use for such exploration include `scikit-surprise`, `lightfm`, `crab`, `rec_sys`, etc.

Summary

In this chapter, we learned about recommendation systems, which are an important and widely known Machine Learning application. We discovered a very popular data source that allowed us to peek inside a small section of online music listeners. We then started on the learning curve of building different types of recommendation engines. We started with a simple vanilla version based on popularity of songs. After that, we upped the complexity quotient of our recommendation engines by developing an item similarity based recommendation engine. We urge you to extend that recommendation engine using the metadata provided as part of million song database. Finally, we concluded the chapter by building a recommendation engine that takes an entirely different point of view. We explored some basics of matrix factorization and learned how a very basic factorization method like singular value decomposition can be used for developing recommendations. We ended the chapter by mentioning about the different libraries that you can use to develop sophisticated engines from the dataset that we created.

We would like to close this chapter by further underlining the importance of recommendation engines, especially in the context of online content delivery. An uncorroborated fact that's often associated with recommendation systems is that "60% of Netflix movie viewings are through recommendations". According to Wikipedia, Netflix has an annual revenue of around $8 billion. Even if half of that is coming from movies and even if the figure is 30% instead of 60%, it means that around $1 billion of Netflix revenue can be attributed to recommendation engines. Although we will never be able to verify these figures, they are definitely a strong argument for recommendation engines and the value they can generate.