MODULE 4

CHAPTER 9

Recommender Systems

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand recommender systems and their business applications.
- Learn about the datasets and algorithm required for building recommendation systems.
- Learn recommender system development techniques such as association rules and collaborative filtering.
- Learn how to build and evaluate recommendation systems using Python libraries.

9.1 | OVERVIEW

Marketing is about connecting the best products or services to the right customers. In today's digital world, personalization is essential for meeting customer's needs more effectively, thereby increasing customer satisfaction and the likelihood of repeat purchases. Recommendation systems are a set of algorithms which recommend most relevant items to users based on their preferences predicted using the algorithms. It acts on behavioral data, such as customer's previous purchase, ratings or reviews to predict their likelihood of buying a new product or service.

Amazon's "Customers who buy this item also bought", Netflix's "shows and movies you may want to watch" are examples of recommendation systems. Recommender systems are very popular for recommending products such as movies, music, news, books, articles, groceries and act as a backbone for cross-selling across industries.

In this chapter, we will be discussing the following three algorithms that are widely used for building recommendation systems:

1. Association Rules
2. Collaborative Filtering
3. Matrix Factorization

9.1.1 | Datasets

For exploring the algorithms specified in the previous section, we will be using the following two publicly available datasets and build recommendations.

1. **groceries.csv:** This dataset contains transactions of a grocery store and can be downloaded from *http://www.sci.csueastbay.edu/~esuess/classes/Statistics_6620/Presentations/ml13/groceries.csv.*
2. **Movie Lens:** This dataset contains 20000263 ratings and 465564 tag applications across 27278 movies. As per the source of data, these data were created by 138493 users between January 09, 1995 and March 31, 2015. This dataset was generated on October 17, 2016. Users were selected and included randomly. All selected users had rated at least 20 movies. The dataset can be downloaded from the link *https://grouplens.org/datasets/movielens/*.

9.2 | ASSOCIATION RULES (ASSOCIATION RULE MINING)

Association rule finds combinations of items that frequently occur together in orders or baskets (in a retail context). The items that frequently occur together are called *itemsets*. Itemsets help to discover relationships between items that people buy together and use that as a basis for creating strategies like combining products as combo offer or place products next to each other in retail shelves to attract customer attention. An application of association rule mining is in Market Basket Analysis (MBA). MBA is a technique used mostly by retailers to find associations between items purchased by customers.

To illustrate the association rule mining concept, let us consider a set of baskets and the items in those baskets purchased by customers as depicted in Figure 9.1.

Items purchased in different baskets are:

1. Basket 1: egg, beer, sugar, bread, diaper
2. Basket 2: egg, beer, cereal, bread, diaper
3. Basket 3: milk, beer, bread
4. Basket 4: cereal, diaper, bread

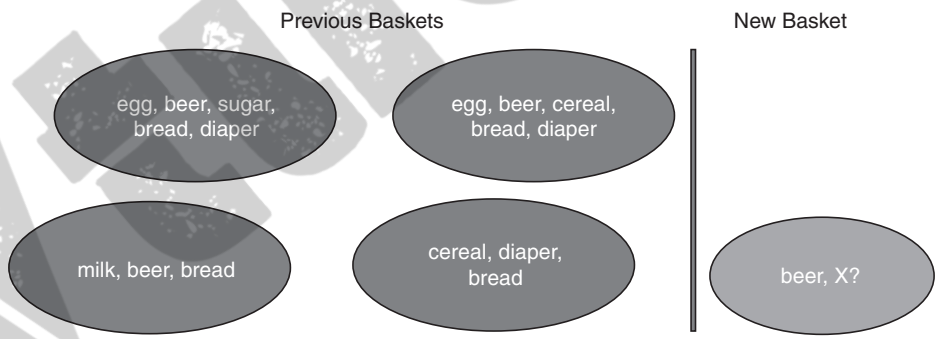

FIGURE 9.1 Baskets and the items in the baskets.

The primary objective of a recommender system is to predict items that a customer may purchase in the future based on his/her purchases so far. In future, if a customer buys beer, can we predict what he/she is most likely to buy along with *beer*? To predict this, we need to find out which items have shown a strong association with *beer* in previously purchased baskets. We can use association rule mining technique to find this out.

Association rule considers all possible combination of items in the previous baskets and computes various measures such as *support*, *confidence,* and *lift* to identify rules with stronger associations. One of the challenges in association rule mining is the number of combination of items that need to be considered; as the number of unique items sold by the seller increases, the number of associations can increase exponentially. And in today's world, retailers sell millions of items. Thus, association rule mining may require huge computational power to go through all possible combinations.

One solution to this problem is to eliminate items that possibly cannot be part of any itemsets. One such algorithm the association rules use *apriori algorithm*. The *apriori* algorithm was proposed by Agrawal and Srikant (1994). The rules generated are represented as

$$\{\text{diapers}\} \rightarrow \{\text{beer}\}$$

which means that customers who purchased diapers also purchased beer in the same basket. {diaper, beer} together is called itemset. {diaper} is called the antecedent and the {beer} is called the consequent. Both antecedents and consequents can have multiple items, e.g. {diaper, milk} → {beer, bread} is also a valid rule. Each rule is measured with a set of metrics, which are explained in the next section.

9.2.1 | Metrics

Concepts such as support, confidence, and lift are used to generate association rules. These concepts are explained below.

9.2.1.1 Support

Support indicates the frequencies of items appearing together in baskets with respect to all possible baskets being considered (or in a sample). For example, the support for (beer, diaper) will be 2/4 (based on the data shown in Figure 9.1), that is, 50% as it appears together in 2 baskets out of 4 baskets.

Assume that X and Y are items being considered. Let

1. N be the total number of baskets.
2. N_{XY} represent the number of baskets in which X and Y appear together.
3. N_X represent the number of baskets in which X appears.
4. N_Y represent the number of baskets in which Y appears.

Then the support between X and Y, Support(X, Y), is given by

$$\text{Support}(X,\ Y) = \frac{N_{XY}}{N} \tag{9.1}$$

To filter out stronger associations, we can set a minimum support (for example, minimum support of 0.01). This means the itemset must be present in at least 1% of baskets. Apriori algorithm uses minimum support criteria to reduce the number of possible itemset combinations, which in turn reduces computational requirements.

If minimum support is set at 0.01, an association between X and Y will be considered *if and only if* both X and Y have minimum support of 0.01. Hence, *apriori* algorithm computes support for each item independently and eliminates items with support less than minimum support. The support of each individual item can be calculated using Eq. (9.1).

9.2.1.2 Confidence

Confidence measures the proportion of the transactions that contain X, which also contain Y. X is called antecedent and Y is called consequent. Confidence can be calculated using the following formula:

$$\text{Confidence}(X \rightarrow Y) = P(Y \mid X) = \frac{N_{XY}}{N_X} \tag{9.2}$$

where $P(Y|X)$ is the conditional probability of Y given X.

9.2.1.3 Lift

Lift is calculated using the following formula:

$$\text{Lift} = \frac{\text{Support}(X,Y)}{\text{Support}(X) \times \text{Support}(Y)} = \frac{N_{XY}}{N_X N_Y} \tag{9.3}$$

Lift can be interpreted as the degree of association between two items. Lift value 1 indicates that the items are independent (no association), lift value of less than 1 implies that the products are substitution (purchase one product will decrease the probability of purchase of the other product) and lift value of greater than 1 indicates purchase of Product X will increase the probability of purchase of Product Y. Lift value of greater than 1 is a necessary condition of generating association rules.

9.2.2 | Applying Association Rules

We will create association rules using the transactions data available in the *groceries.csv* dataset. Each line in the dataset is an order and contains a variable number of items. Each item in each order is separated by a comma in the dataset.

9.2.2.1 Loading the Dataset

Python's *open()* method can be used to open the file and *readlines()* to read each line. The following code block can be used for loading and reading the data:

```
all_txns = []

#open the file
with open('groceries.csv') as f:
    #read each line
    content = f.readlines()
    #Remove white space from the beginning and end of the line
    txns = [x.strip() for x in content]
    # Iterate through each line and create a list of transactions
    for each_txn in txns:
        #Each transaction will contain a list of item in the
        transaction
        all_txns.append( each_txn.split(',') )
```

The steps in this code block are explained as follows:

1. The code opens the file *groceries.csv.*
2. Reads all the lines from the file.
3. Removes leading or trailing white spaces from each line.
4. Splits each line by a comma to extract items.
5. Stores the items in each line in a list.

In the end, the variable *all_txns* will contain a list of orders and list of items in each order. An order is also called a transaction.

To print the first five transactions, we can use the following code:

```
all_txns[0:5]
```

The output is shown below:

```
[['citrus fruit', 'semi-finished bread', 'margarine', 'ready soups'],
 ['tropical fruit', 'yogurt', 'coffee'],
 ['whole milk'],
 ['pip fruit', 'yogurt', 'cream cheese', 'meat spreads'],
 ['other vegetables',
 'whole milk',
 'condensed milk',
 'long life bakery product']]
```

9.2.2.2 Encoding the Transactions

Python library *mlxtend* provides methods to generate association rules from a list of transactions. But these methods require the data to be fed in specific format. The transactions and items need to be converted into a tabular or matrix format. Each row represents a transaction and each column represents an item. So, the matrix size will be of $M \times N$, where M represents the total number of transactions and N represents all unique items available across all transactions (or the number of items sold by the seller). The items available in each transaction will be represented in one-hot-encoded format, that is, the item is encoded 1 if it exists in the transaction or 0 otherwise. The *mlxtend* library has a feature pre-processing implementation class called *OnehotTransactions* that will take *all_txns* as an input and convert the transactions and items into one-hot-encoded format. The code for converting the transactional data using one-hot encoding is as follows:

```
# Import all required libraries
import pandas as pd
import numpy as np
from mlxtend.preprocessing import OnehotTransactions
from mlxtend.frequent_patterns import apriori, association_rules
```

The print method can be used for printing the first five transactions and items, indexed from 10 to 20. The results are shown in Table 9.1.

```
# Initialize OnehotTransactions
one_hot_encoding = OnehotTransactions()
# Transform the data into one-hot-encoding format
one_hot_txns = one_hot_encoding.fit(all_txns).transform(all_txns)
# Convert the matrix into the dataframe.
one_hot_txns_df = pd.DataFrame(one_hot_txns,
                               columns=one_hot_encoding.columns_)
```

```
one_hot_txns_df.iloc[5:10, 10:20]
```

TABLE 9.1 One-hot-encoded transactions data

	Berries	Beverages	Bottled beer	Bottled water	Brandy	Brown bread	Butter	Butter milk	Cake bar	Candles
5	0	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

It can be noticed from Table 9.1 that transaction with index 5 contains an item called *butter* (item purchased by the customer) and transaction with index 7 contains an item *bottled beer*. All other entries in Table 9.1 are 0, which implies that these items were not purchased. The transactional matrices are likely to be sparse since each customer is likely to buy very few items in comparison to the total number of items sold by the seller. The following code can be used for finding the size (shape or dimension) of the matrix.

```
one_hot_txns_df.shape
```

```
(9835, 171)
```

The sparse matrix has a dimension of 9835 × 171. So, a total of 9835 transactions and 171 items are available. This matrix can be fed to generate rules, which will be discussed in the next sub-section.

9.2.2.3 Generating Association Rules

We will use *apriori* algorithms to generate itemset. The total number of itemset will depend on the number of items that exist across all transactions. The number of items in the data can be obtained using the following code:

```
len(one_hot_txns_df.columns)
```

```
171
```

The code gives us an output of 171, that is, as mentioned in the previous section, there are 171 items. For itemset containing 2 items in each set, the total number of itemsets will be ${}^{171}C_2$, that is, the number of itemset will be 14535. It is a very large number and computationally intensive. To limit the number of

generated rules, we will apply minimum support value. All items that do not have the minimum support will be removed from the possible itemset combinations.

Apriori algorithm takes the following parameters:

1. **df: pandas** – DataFrame in a one-hot-encoded format.
2. **min_support: float** – A float between 0 and 1 for minimum support of the itemsets returned. Default is 0.5.
3. **use_colnames: boolean** – If true, uses the DataFrames' column names in the returned DataFrame instead of column indices.

We will be using a minimum support of 0.02, that is, the itemset is available in at least 2% of all transactions. The following commands can be used for setting minimum support.

```
frequent_itemsets = apriori(one_hot_txns_df,
                            min_support=0.02,
                            use_colnames=True)
```

The following command can be used for printing 10 randomly sampled itemsets and their corresponding support. The itemsets are shown in Table 9.2.

```
frequent_itemsets.sample(10, random_state = 90)
```

TABLE 9.2 Itemsets with a minimum support value of 0.02

	Support	Itemsets
60	0.020437	[bottled beer, whole milk]
52	0.033859	[sugar]
89	0.035892	[other vegetables, tropical fruit]
105	0.021047	[root vegetables, tropical fruit]
88	0.032740	[other vegetables, soda]
16	0.058058	[coffee]
111	0.024504	[shopping bags, whole milk]
36	0.079817	[newspapers]
119	0.056024	[whole milk, yogurt]
55	0.071683	[whipped/sour cream]

The *apriori* algorithm filters out frequent itemsets which have minimum support of greater than 2%. From Table 9.2, we can infer that *whole milk* and *yogurt* appear together in about 5.6% of the baskets. These itemsets can be passed to *association_rules* for generating rules and corresponding metrics. The following commands are used. The corresponding association rules are shown in Table 9.3

1. **df : pandas** – DataFrame of frequent itemsets with columns ['support', 'itemsets'].
2. **metric** – In this use 'confidence' and 'lift' to evaluate if a rule is of interest. Default is 'confidence'.
3. **min_threshold** – Minimal threshold for the evaluation metric to decide whether a candidate rule is of interest.

```
rules = association_rules(frequent_itemsets, # itemsets
                          metric="lift", # lift
                          min_threshold=1)
```

```
rules.sample(5)
```

TABLE 9.3 Association rules

	Antecedants	Consequents	Support	Confidence	Lift
7	(soda)	(rolls/buns)	0.174377	0.219825	1.195124
55	(yogurt)	(bottled water)	0.139502	0.164723	1.490387
74	(soda)	(yogurt)	0.174377	0.156851	1.124368
89	(root vegetables)	(whole milk)	0.108998	0.448694	1.756031
59	(citrus fruit)	(yogurt)	0.082766	0.261671	1.875752

9.2.2.4 Top Ten Rules

Let us look at the top 10 association rules sorted by *confidence*. The rules stored in the variable *rules* are sorted by confidence in descending order. They are printed as seen in Table 9.4.

```
rules.sort_values('confidence',
                  ascending = False)[0:10]
```

TABLE 9.4 Top 10 association rules based on confidence sorted from highest to lowest

	Antecedants	Consequents	Support	Confidence	Lift
42	(yogurt, other vegetables)	(whole milk)	0.043416	0.512881	2.007235
48	(butter)	(whole milk)	0.055414	0.497248	1.946053
120	(curd)	(whole milk)	0.053279	0.490458	1.919481
80	(other vegetables, root vegetables)	(whole milk)	0.047382	0.489270	1.914833
78	(root vegetables, whole milk)	(other vegetables)	0.048907	0.474012	2.449770
27	(domestic eggs)	(whole milk)	0.063447	0.472756	1.850203
0	(whipped/sour cream)	(whole milk)	0.071683	0.449645	1.759754
89	(root vegetables)	(whole milk)	0.108998	0.448694	1.756031
92	(root vegetables)	(other vegetables)	0.108998	0.434701	2.246605
24	(frozen vegetables)	(whole milk)	0.048094	0.424947	1.663094

From Table 9.4, we can infer that the probability that a customer buys (whole milk), given he/she has bought (yogurt, other vegetables), is 0.51. Now, these rules can be used to create strategies like keeping the items together in store shelves or cross-selling.

9.2.2.5 *Pros and Cons of Association Rule Mining*

The following are advantages of using association rules:

1. Transactions data, which is used for generating rules, is always available and mostly clean.
2. The rules generated are simple and can be interpreted.

However, association rules do not take the preference or ratings given by customers into account, which is an important information pertaining for generating rules. If customers have bought two items but disliked one of them, then the association should not be considered. Collaborative filtering takes both, what customers bought and how they liked (rating) the items, into consideration before recommending.

Association rules mining is used across several use cases including product recommendations, fraud detection from transaction sequences, medical diagnosis, weather prediction, etc.

9.3 | COLLABORATIVE FILTERING

Collaborative filtering is based on the notion of similarity (or distance). For example, if two users **A** and **B** have purchased the same products and have rated them similarly on a common rating scale, then **A** and **B** can be considered similar in their buying and preference behavior. Hence, if **A** buys a new product and rates high, then that product can be recommended to **B**. Alternatively, the products that **A** has already bought and rated high can be recommended to **B**, if not already bought by **B**.

9.3.1 | How to Find Similarity between Users?

Similarity or the distance between users can be computed using the rating the users have given to the common items purchased. If the users are similar, then the similarity measures such as Jaccard coefficient and cosine similarity will have a value closer to 1 and distance measures such as Euclidian distance will have low value. Calculating similarity and distance have already been discussed in Chapter 7. Most widely used distances or similarities are Euclidean distance, Jaccard coefficient, cosine similarity, and Pearson correlation.

We will be discussing collaborative filtering technique using the example described below. The picture in Figure 9.2 depicts three users *Rahul, Purvi,* and *Gaurav* and the books they have bought and rated.

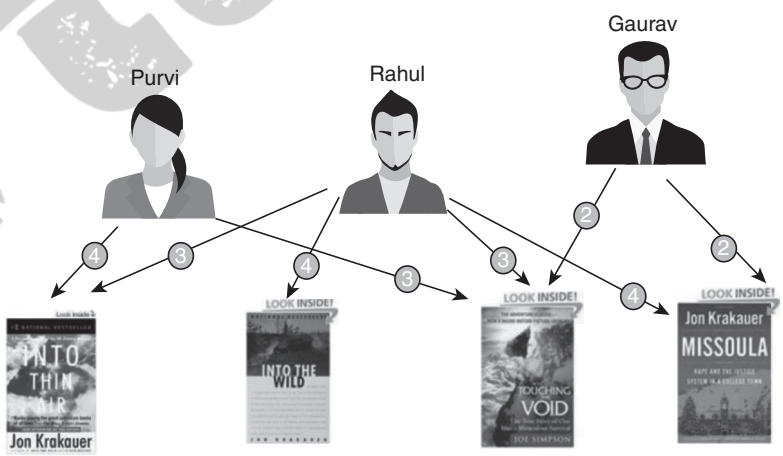

FIGURE 9.2 Users, their purchases and ratings.

The users are represented using their rating on the Euclidean space in Figure 9.3. Here the dimensions are represented by the two books *Into Thin Air* and *Missoula*, which are the two books commonly bought by *Rahul, Purvi,* and *Gaurav*.

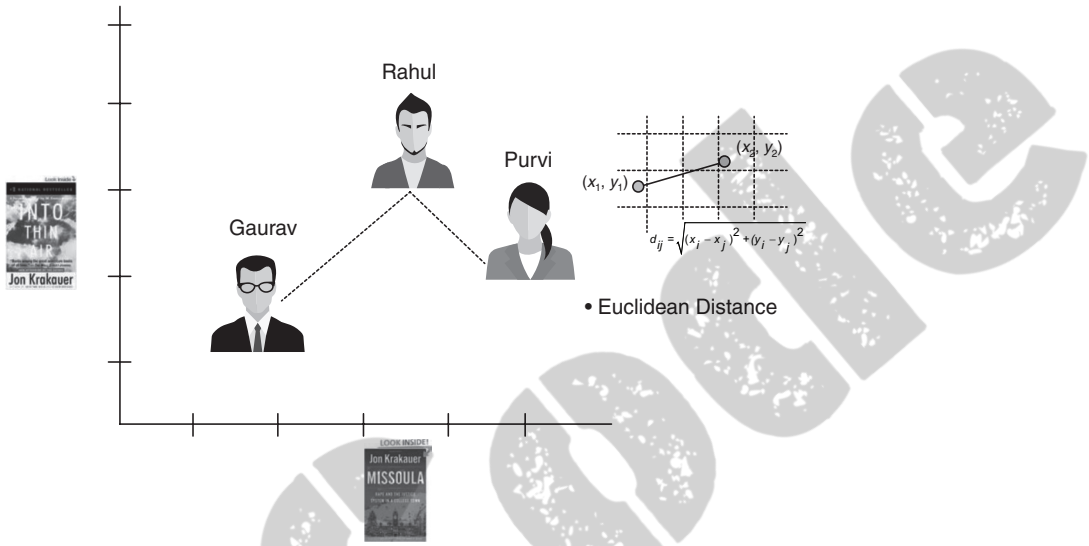

FIGURE 9.3 Euclidean distance based on user's ratings.

Figure 9.3 shows that *Rahul's* preferences are similar to *Purvi's* rather than to *Gaurav's*. So, the other book, *Into the Wild*, which *Rahul* has bought and rated high, can now be recommended to *Purvi*.

Collaborative filtering comes in two variations:

1. **User-Based Similarity:** Finds *K* similar users based on common items they have bought.
2. **Item-Based Similarity:** Finds *K* similar items based on common users who have bought those items.

Both algorithms are similar to *K*-Nearest Neighbors (KNN), which was discussed in Chapter 6.

9.3.2 | User-Based Similarity

We will use *MovieLens* dataset (see https://grouplens.org/datasets/movielens/) for finding similar users based on common movies the users have watched and how they have rated those movies. The file *ratings.csv* in the dataset contains ratings given by users. Each line in this file represents a rating given by a user to a movie. The ratings are on the scale of 1 to 5. The dataset has the following features:

1. userId
2. movieId
3. rating
4. timestamp

9.3.2.1 Loading the Dataset

The following loads the file onto a DataFrame using *pandas' read_csv()* method.

```
rating_df = pd.read_csv( "ml-latest-small/ratings.csv" )
```

Let us print the first five records.

```
rating_df.head(5)
```

The output is shown in Table 9.5.

TABLE 9.5 Movie ID and rating

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205

The *timestamp* column will not be used in this example, so it can be dropped from the dataframe.

```
rating_df.drop( 'timestamp', axis = 1, inplace = True )
```

The number of unique users in the dataset can be found using method *unique()* on *userId* column.

```
len( rating_df.userId.unique() )
```

```
671
```

Similarly, the number of unique movies in the dataset is

```
len( rating_df.movieId.unique() )
```

```
9066
```

Before proceeding further, we need to create a pivot table or matrix and represent users as rows and movies as columns. The values of the matrix will be the ratings the users have given to those movies. As there are 671 users and 9066 movies, we will have a matrix of size 671 × 9066. The matrix will be very sparse as very few cells will be filled with the ratings using only those movies that users have watched.

Those movies that the users have not watched and rated yet, will be represented as NaN. Pandas DataFrame has pivot method which takes the following three parameters:

1. **index:** Column value to be used as DataFrame's index. So, it will be *userId* column of *rating_df*.
2. **columns:** Column values to be used as DataFrame's columns. So, it will be *movieId* column of *rating_df*.
3. **values:** Column to use for populating DataFrame's values. So, it will be *rating* column of *rating_df*.

```
user_movies_df = rating_df.pivot( index='userId',
                                  columns='movieId',
                                  values = "rating"
                                ).reset_index(drop=True)
user_movies_df.index=rating_df.userId.unique()
```

Let us print the first 5 rows and first 15 columns. The result is shown in Table 9.6.

```
user_movies_df.iloc[0:5, 0:15]
```

TABLE 9.6 First few records

movieId	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

The DataFrame contains NaN for those entries where users have seen a movie and not rated. We can impute those NaNs with 0 values using the following codes. The results are shown in Table 9.7.

```
user_movies_df.fillna( 0, inplace = True
) user_movies_df.iloc[0:5, 0:10]
```

TABLE 9.7 NaN entries replaced with 0.0

movieId	1	2	3	4	5	6	7	8	9	10
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0
5	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

9.3.2.2 Calculating Cosine Similarity between Users

Each row in *user_movies_df* represents a user. If we compute the similarity between rows, it will represent the similarity between those users. *sklearn.metrics.pairwise_distances* can be used to compute distance between all pairs of users. *pairwise_distances()* takes a *metric* parameter for what distance measure to use. We will be using cosine similarity for finding similarity.

Cosine similarity closer to 1 means users are very similar and closer to 0 means users are very dissimilar. The following code can be used for calculating the similarity.

```
from sklearn.metrics import pairwise_distances
from scipy.spatial.distance import cosine, correlation

user_sim = 1 - pairwise_distances( user_movies_df.values,
                                   metric="cosine" )
#Store the results in a dataframe
user_sim_df = pd.DataFrame( user_sim )
#Set the index and column names to user ids (0 to 671)
user_sim_df.index = rating_df.userId.unique()
user_sim_df.columns = rating_df.userId.unique()
```

We can print the similarity between first 5 users by using the following code. The result is shown in Table 9.8.

```
user_sim_df.iloc[0:5, 0:5]
```

TABLE 9.8 Cosine similarity values

	1	2	3	4	5
1	1.000000	0.000000	0.000000	0.074482	0.016818
2	0.000000	1.000000	0.124295	0.118821	0.103646
3	0.000000	0.124295	1.000000	0.081640	0.151531
4	0.074482	0.118821	0.081640	1.000000	0.130649
5	0.016818	0.103646	0.151531	0.130649	1.000000

The total dimension of the matrix is available in the *shape* variable of *user_sim_df* matrix.

```
user_sim_df.shape
```

```
(671, 671)
```

user_sim_df matrix shape shows that it contains the cosine similarity between all possible pairs of users. And each cell represents the cosine similarity between two specific users. For example, the similarity between userid 1 and userid 5 is 0.016818.

The diagonal of the matrix shows the similarity of an user with itself (i.e., 1.0). This is true as each user is most similar to himself or herself. But we need the algorithm to find other users who are similar to a specific user. So, we will set the diagonal values as 0.0 . The result is shown in Table 9.9.

```
np.fill_diagonal( user_sim, 0 )
user_sim_df.iloc[0:5, 0:5]
```

TABLE 9.9 Diagonal similarity value set to 0

	1	2	3	4	5
1	0.000000	0.000000	0.000000	0.074482	0.016818
2	0.000000	0.000000	0.124295	0.118821	0.103646
3	0.000000	0.124295	0.000000	0.081640	0.151531
4	0.074482	0.118821	0.081640	0.000000	0.130649
5	0.016818	0.103646	0.151531	0.130649	0.000000

All diagonal values are set to 0, which helps to avoid selecting self as the most similar user.

9.3.2.3 Filtering Similar Users

To find most similar users, the maximum values of each column can be filtered. For example, the most similar user to first 5 users with *userid* 1 to 5 can be obtained using the following code:

```
user_sim_df.idxmax(axis=1)[0:5]
```

```
1   325
2   338
3   379
4   518
5   313
dtype: int64
```

The above result shows user 325 is most similar to user 1, user 338 is most similar to user 2, and so on.

To dive a little deeper to understand the similarity, let us print the similarity values between user 2 and users ranging from 331 to 340.

```
user_sim_df.iloc[1:2, 330:340]
```

Output

	331	332	333	334	335	336	337	338	339	340
2	0.030344	0.002368	0.052731	0.047094	0.0	0.053044	0.05287	0.581528	0.093863	0.081814

The output shows that the cosine similarity between *userid* 2 and *userid* 338 is 0.581528 and highest. But why is user 338 most similar to user 2? This can be explained intuitively if we can verify that the two users have watched several movies in common and rated very similarly. For this, we need to read *movies* dataset, which contains the movie id along with the movie name.

9.3.2.4 Loading the Movies Dataset

Movie information is contained in the file *movies.csv*. Each line of this file contains the *movieid*, the movie name, and the movie genre.

Movie titles are entered manually or imported from https://www.themoviedb.org/ and include the year of release in parentheses. Errors and inconsistencies may exist in these titles. The movie can be loaded using the following codes:

```
movies_df = pd.read_csv( "ml-latest-small/movies.csv" )
```

We will print the first 5 movie details using the following code. The result is shown in Table 9.10.

```
movies_df[0:5]
```

TABLE 9.10 Movie details

	movieid	title	genres
0	1	Toy Story (1995)	Adventure | Animation | Children | Comedy | Fantasy
1	2	Jumanji (1995)	Adventure | Children | Fantasy
2	3	Grumpier Old Men (1995)	Comedy | Romance
3	4	Waiting to Exhale (1995)	Comedy | Drama | Romance
4	5	Father of the Bride Part II (1995)	Comedy

The *genres* column is dropped from the DataFrame, as it is not going to be used in this analysis.

```
movies_df.drop( 'genres', axis = 1, inplace = True )
```

9.3.2.5 Finding Common Movies of Similar Users

The following method takes *userids* of two users and returns the common movies they have watched and their ratings.

```
def get_user_similar_movies( user1, user2 ):
    # Inner join between movies watched between two users will give
    # the common movies watched.
    common_movies = rating_df[rating_df.userId == user1].merge(
          rating_df[rating_df.userId == user2],
          on = "movieId",
          how = "inner" )
```

```
    # join the above result set with movies details
    return common_movies.merge( movies_df, on = 'movieId' )
```

To find out the movies, user 2 and user 338 have watched in common and how they have rated each one of them, we will filter out movies that both have rated at least 4 to limit the number of movies to print. The movies are shown in Table 9.11.

```
common_movies = get_user_similar_movies( 2, 338 )
```

```
common_movies[(common_movies.rating_x >= 4.0) &
              ((common_movies.rating_y >= 4.0))]
```

TABLE 9.11 Common movies between user 2 and user 338 with least rating of 4.0

	userId_x	movieId	rating_x	userId_y	rating_y	title
0	2	17	5.0	338	4.0	Sense and Sensibility (1995)
2	2	47	4.0	338	4.0	Seven (a.k.a. Se7en) (1995)
5	2	150	5.0	338	4.0	Apollo 13 (1995)
28	2	508	4.0	338	4.0	Philadelphia (1993)
29	2	509	4.0	338	4.0	Piano, The (1993)
31	2	527	4.0	338	5.0	Schindler's List (1993)
34	2	589	5.0	338	5.0	Terminator 2: Judgment Day (1991)

From the table we can see that users 2 and 338 have watched 7 movies in common and have rated almost on the same scale. Their preferences seem to be very similar.

How about users with dissimilar behavior? Let us check users 2 and 332, whose cosine similarity is 0.002368.

```
common_movies = get_user_similar_movies( 2, 332 )
common_movies
```

	userId_x	movieId	rating_x	userId_y	rating_y	title
0	2	552	3.0	332	0.5	Three Musketeers, The (1993)

Users 2 and 332 have only one movie in common and have rated very differently. They indeed are very dissimilar.

9.3.2.6 Challenges with User-Based Similarity

Finding user similarity does not work for new users. We need to wait until the new user buys a few items and rates them. Only then users with similar preferences can be found and recommendations can be made based on that. This is called *cold start* problem in recommender systems. This can be overcome by using item-based similarity. *Item-based similarity* is based on the notion that if two items have been bought by

many users and rated similarly, then there must be some inherent relationship between these two items. In other terms, in future, if a user buys one of those two items, he or she will most likely buy the other one.

9.3.3 | Item-Based Similarity

If two movies, movie A and movie B, have been watched by several users and rated very similarly, then movie A and movie B can be similar in taste. In other words, if a user watches movie A, then he or she is very likely to watch B and vice versa.

9.3.3.1 Calculating Cosine Similarity between Movies

In this approach, we need to create a pivot table, where the rows represent movies, columns represent users, and the cells in the matrix represent ratings the users have given to the movies. So, the *pivot()* method will be called with *movieId* as *index* and *userId* as *columns* as described below:

```
rating_mat = rating_df.pivot(index='movieId',
                             columns='userId',
values = "rating").reset_index(drop = True)
# Fill all NaNs with 0
rating_mat.fillna(0, inplace = True)
# Find the correlation between movies
movie_sim = 1 - pairwise_distances(rating_mat.values,
                                   metric="correlation")
# Fill the diagonal with 0, as it repreresents the auto-correlation
# of movies
movie_sim_df = pd.DataFrame( movie_sim )
```

Now, the following code is used to print similarity between the first 5 movies. The results are shown in Table 9.12.

```
movie_sim_df.iloc[0:5, 0:5]
```

TABLE 9.12 Movie similarity

	0	1	2	3	4
0	1.000000	0.223742	0.183266	0.071055	0.105076
1	0.223742	1.000000	0.123790	0.125014	0.193144
2	0.183266	0.123790	1.000000	0.147771	0.317911
3	0.071055	0.125014	0.147771	1.000000	0.150562
4	0.105076	0.193144	0.317911	0.150562	1.000000

The shape of the above similarity matrix is

```
movie_sim_df.shape
```

```
(9066, 9066)
```

There are 9066 movies and the dimension of the matrix (9066, 9066) shows that the similarity is calculated for all pairs of 9066 movies.

9.3.3.2 Finding Most Similar Movies

In the following code, we write a method *get_similar_movies()* which takes a *movieid* as a parameter and returns the similar movies based on cosine similarity. Note that movieid and index of the movie record in the *movies_df are not same.* We need to find the index of the movie record from the movieid and use that to find similarities in the *movie_sim_df*. It takes another parameter *topN* to specify how many similar movies will be returned.

```
def get_similar_movies( movieid, topN = 5 ):
    # Get the index of the movie record in movies_df
    movieidx = movies_df[movies_df.movieId == movieid].index[0]
    movies_df['similarity'] = movie_sim_df.iloc[movieidx]
    top_n = movies_df.sort_values( ["similarity"], ascending =
                                                    False )[0:topN]
    return top_n
```

The above method *get_similar_movies()* takes movie id as an argument and returns other movies which are similar to it. Let us find out how the similarities play out by finding out movies which are similar to the movie *Godfather*. And if it makes sense at all! The movie id for the movie Godfather is *858.*

```
movies_df[movies_df.movieId == 858]
```

	movieId	title	similarity
695	858	Godfather, The (1972)	1.0

```
get_similar_movies(858)
```

Table 9.13 shows the movies that are similar to Godfather.

TABLE 9.13 Movies similar to Godfather

	movieId	title	similarity
695	858	Godfather, The (1972)	1.000000
977	1221	Godfather: Part II, The (1974)	0.709246
969	1213	Goodfellas (1990)	0.509372
951	1193	One Flew Over the Cuckoo's Nest (1975)	0.430101
1744	2194	Untouchables, The (1987)	0.418966

It can be observed from Table 9.13 that users who watched *'Godfather, The'*, also watched *'Godfather: Part II'* the most. This makes absolute sense! It also indicates that the users have watched *Goodfellas (1990), One Flew Over the Cuckoo's Nest (1975),* and *Untouchables, The (1987).*

So, in future, if any user watches *'Godfather, The'*, the other movies can be recommended to them. Let us find out which movies are similar to the movie Dumb and Dumber.

```
movies_df[movies_df.movieId == 231]
```

	movieId	title	similarity
203	231	Dumb & Dumber (Dumb and Dumber) (1994)	0.054116

```
get_similar_movies(231)
```

Table 9.14 shows the movies that are similar to Dumb and Dumber

TABLE 9.14 Movies similar to Dumb and Dumber

	movieId	title	similarity
203	231	Dumb & Dumber (Dumb and Dumber) (1994)	1.000000
309	344	Ace Ventura: Pet Detective (1994)	0.635735
18	19	Ace Ventura: When Nature Calls (1995)	0.509839
447	500	Mrs. Doubtfire (1993)	0.485764
331	367	Mask, The (1994)	0.461103

We can see from the table, most of the movies are of *comedy* genre and belong to the actor *Jim Carrey*.

9.4 | USING SURPRISE LIBRARY

For real-world implementations, we need a more extensive library which hides all the implementation details and provides abstract Application Programming Interfaces (APIs) to build recommender systems. *Surprise* is a Python library for accomplishing this. It provides the following features:

1. Various ready-to-use prediction algorithms like neighborhood methods (user similarity and item similarity), and matrix factorization-based. It also has built-in similarity measures such as cosine, mean square distance (MSD), Pearson correlation coefficient, etc.
2. Tools to evaluate, analyze, and compare the performance of the algorithms. It also provides methods to recommend.

We import the required modules or classes from surprise library. All modules or classes and their purpose are discussed in the subsequent sections.

```
from surprise import Dataset, Reader, KNNBasic, evaluate, accuracy
```

The *surprise.Dataset* is used to load the datasets and has a method *load_from_df* to convert DataFrames to Dataset. *Reader* class can be used to provide the range of rating scales that is being used.

```
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(rating_df[['userId',
                                       'movieId',
                                       'rating']], reader=reader)
```

9.4.1 | User-Based Similarity Algorithm

The *surprise.prediction_algorithms.knns.KNNBasic* provides the collaborative filtering algorithm and takes the following parameters:

1. **K:** The (max) number of neighbors to take into account for aggregation.
2. **min_k:** The minimum number of neighbors to take into account for aggregation, if there are not enough neighbors.
3. **sim_options - (dict):** A dictionary of options for the similarity measure.
   (a) **name**: Name of the similarity to be used, e.g., *cosine, msd or pearson.*
   (b) **user_based**: True for user-based similarity and False for item-based similarity.

The following code implements movies recommendation based on *Pearson* correlation and 20 nearest similar users.

```
## Set coefficient and similarity parameters for building model
item_based_cosine_sim = {'name': 'pearson',
                         'user_based': True}

knn = KNNBasic(k= 20,
               min_k = 5,
               sim_options = item_based_cosine_sim)
```

The *surprise.model_selection* provides *cross_validate* method to split the dataset into multiple folds, runs the algorithm, and reports the accuracy measures. It takes the following parameters:

1. **algo:** The algorithm to evaluate.
2. **data (Dataset):** The dataset on which to evaluate the algorithm.
3. **Measures:** The performance measures to compute. Allowed names are function names as defined in the accuracy module. Default is ['rmse', 'mae'].
4. **cv:** The number of folds for K-Fold cross validation strategy.

We can do 5-fold cross-validation to measure *RMSE* score to find out how the algorithm performs on the dataset.

```
from surprise.model_selection import cross_validate

cv_results = cross_validate(knn,
                            data,
                            measures=['RMSE'],
                            cv=5,
                            verbose=False)
```

It reports test accuracy for each fold along with the time it takes to build and test the models. Let us take the average accuracy across all the folds.

```
np.mean(cv_results.get('test_rmse'))
```

```
0.9909387452695102
```

This model can predict with root mean square error (RMSE) of 0.99. But can we do better?

9.4.2 | Finding the Best Model

Similar to finding the most optimal hyper-parameters using *GridSearchCV* in *sklearn* as discussed in Chapter 6: Advanced Machine Learning, *Surprise* provides *GridSearchCV* to search through various models and similarity indexes to find the model that gives the highest accuracy.

The *surprise.model_selection.search.GridSearchCV* takes the following parameters:

1. **algo_class:** The class of the algorithm to evaluate (e.g., KNNBasic).
2. **param_grid:** Dictionary with hyper parameters as keys and list of corresponding possible values that will be used to search optimal parameter values. All combinations will be evaluated with desired algorithm.
3. **measures:** The performance measures to compute (i.e., ['rmse', 'mae']).
4. **cv:** The number of folds for K-Fold cross validation strategy.
5. **refit:** If True, refit the algorithm on the whole dataset using the set of parameters that gave the best average performance. If False, it does not refit on the whole dataset.

The GridSearchCV method returns the best model and its parameters. The possible values for the parameters that will be searched to find the most optimal parameters are as below:

1. Number of neighbors [10, 20].
2. Similarity indexes ['cosine', 'pearson'] .
3. User-based or item-based similarity.

```
from surprise.model_selection.search import GridSearchCV
```

```
param_grid = {'k': [10, 20],
              'sim_options': {'name': ['cosine', 'pearson'],
                              'user_based': [True, False]
              }

grid_cv = GridSearchCV(KNNBasic,
                       param_grid,
                       measures=['rmse'],
                       cv=5,
                       refit=True)
grid_cv.fit(data)
```

Let us print the score and parameters of the best model.

```
# Best RMSE score

print(grid_cv.best_score['rmse'])

# Combination of parameters that gave the best RMSE score
print(grid_cv.best_params['rmse'])
```

```
0.9963783863084851
{'sim_options': {'name': 'cosine', 'user_based': True}, 'k': 20}
```

The best model is user-based collaborative filtering with cosine similarity and 20 similar users. Details of the grid search are captured in the variable *cv_results*. We can convert it to a DataFrame and print a few columns like *param_sim_options* and *mean_test_rmse*.

```
results_df = pd.DataFrame.from_dict(grid_cv.cv_results)
results_df[['param_k', 'param_sim_options', 'mean_test_rmse',
            'rank_test_rmse']]
```

TABLE 9.15 Best model

	param_k	param_sim_options	mean_test_rmse	rank_test_rmse
0	10	{'name': 'cosine', 'user_based': True}	1.009724	4
1	20	{'name': 'cosine', 'user_based': True}	0.996378	1
2	10	{'name': 'cosine', 'user_based': False}	1.048802	8
3	20	{'name': 'cosine', 'user_based': False}	1.015225	6
4	10	{'name': 'pearson', 'user_based': True}	1.012283	5
5	20	{'name': 'pearson', 'user_based': True}	1.000766	2
6	10	{'name': 'pearson', 'user_based': False}	1.030900	7
7	20	{'name': 'pearson', 'user_based': False}	1.004205	3

The detailed output of grid search is shown in Table 9.15. Each record represents the parameters used to build the model and the corresponding RMSE of the model. The last column *rank_test_rmse* shows the rank of the model as per the RMSE on test data (*mean_test_rmse*) among all the models.

9.4.3 | Making Predictions

To make predictions, the weighted ratings are calculated using the ratings of K nearest users, and the movies with highest weighted ratings are recommended. For example, if r_i is the rating of nearest user (or neighbor) u_i and there are K nearest neighbors, s_i is the similarity between the user and the neighbor, then predicted rating for the movie would be

$$r = \frac{\sum_{i=1}^{K} r_i s_i}{\sum_{i=1}^{K} r_i} \tag{9.4}$$

All models in *surprise* library provide a method called *predict(),* which takes user id and movie id and predicts its ratings. It returns a *Prediction* class, which has a variable *est* (stands for estimated value) that gives the predicted rating.

```
grid_cv.predict( 1, 2 )
```

```
Prediction(uid=1, iid=2, r_ui=None, est=2.5532967054839784,
details= {'was_impossible': False, 'actual_k': 20})
```

9.5 | MATRIX FACTORIZATION

Matrix factorization is a matrix decomposition technique. Matrix decomposition is an approach for reducing a matrix into its constituent parts. Matrix factorization algorithms decompose the user-item matrix into the product of two lower dimensional rectangular matrices.

In Figure 9.4, the original matrix contains users as rows, movies as columns, and rating as values. The matrix can be decomposed into two lower dimensional rectangular matrices.

Users–Movies Rating Matrix

		Movies				
		M1	M2	M3	M4	M5
Users	U1	3	4	2	5	1
	U2	2	4	1	2	4
	U3	3	3	5	2	2

Users–Factors Matrix

		Factors		
		F1	F2	F3
Users	U1	0.73	3.22	0
	U2	0	1.57	2.53
	U3	1.62	0	1.44

Factors–Movies Matrix

		Movies				
		M1	M2	M3	M4	M5
Factors	F1	1.47	1	2.73	1.73	0
	F2	0.6	1.01	0	1.27	0.31
	F3	0.42	0.95	0.39	0	1.39

FIGURE 9.4 Matrix factorization.

The Users–Movies matrix contains the ratings of 3 users (U1, U2, U3) for 5 movies (M1 through M5). This Users–Movies matrix is factorized into a (3, 3) Users–Factors matrix and (3, 5) Factors–Movies matrix. Multiplying the Users–Factors and Factors–Movies matrix will result in the original Users–Movies matrix.

The idea behind matrix factorization is that there are latent factors that determine why a user rates a movie, and the way he/she rates. The factors could be the story or actors or any other specific attributes of the movies. But we may never know what these factors actually represent. That is why they are called latent factors. A matrix with size (*n, m*), where *n* is the number of users and *m* is the number of movies, can be factorized into (*n, k*) and (*k, m*) matrices, where *k* is the number of factors.

The Users–Factors matrix represents that there are three factors and how each user has preferences towards these factors. Factors–Movies matrix represents the attributes the movies possess.

In the above example, U1 has the highest preference for factor F2, whereas U2 has the highest preference for factor F3. Similarly, the F2 factor is high in movies M2 and M4. Probably this is the reason why U1 has given high ratings to movies M2 (4) and M4 (5).

One of the popular techniques for matrix factorization is Singular Vector Decomposition (SVD). *Surprise* library provides SVD algorithm, which takes the number of factors (*n_factors*) as a parameter. We will use 5 latent factors for our example.

```
from surprise import SVD

# Use 10 factors for building the model
svd = SVD( n_factors = 5 )
```

Let us use five-fold cross-validation for testing model's performance.

```
cv_results = cross_validate(svd,
                            data,
                            measures=['RMSE'],
                            cv=5,
                            verbose=True)
```

```
Evaluating RMSE of algorithm SVD on 5 split(s).
                Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)  0.8904  0.8917  0.8983  0.8799  0.8926  0.8906  0.0060
Fit time        1.91    1.99    1.96    1.87    1.84    1.91    0.06
Test time       0.21    0.18    0.18    0.17    0.18    0.18    0.01
```

Mean RMSE for SVD is about 0.8906, which is better than earlier algorithms.

CHAPTER 10

Text Analytics

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the challenges associated with the handling of text data.
- Learn various pre-processing steps to prepare text data for modelling.
- Learn Naïve–Bayes classification algorithm.
- Learn to develop model for sentiment classification.

10.1 | OVERVIEW

In today's world, one of the biggest sources of information is text data, which is unstructured in nature. Finding customer sentiments from product reviews or feedbacks, extracting opinions from social media data are a few examples of text analytics. Finding insights from text data is not as straight forward as structured data and it needs extensive data pre-processing. All the techniques that we have learnt so far require data to be available in a structured format (matrix form). Algorithms that we have explored so far, such as regression, classification, or clustering, can be applied to text data only when the data is cleaned and prepared. For example, predicting stock price movements from news articles is an example of regression in which the features are positive and negative sentiments about a company. Classifying customer sentiment from his or her review comments as positive and negative is an example of classification using text data.

In this chapter, we will use a dataset that is available at *https://www.kaggle.com/c/si650winter11/* data (the original data was contributed by the University of Michigan) for building a classification model to classify sentiment. The data consists of sentiments expressed by users on various movies. Here each comment is a record, which is either classified as positive or negative.

10.2 | SENTIMENT CLASSIFICATION

In the dataset described in the previous paragraph, *sentiment_train* dataset contains review comments on several movies. Comments in the dataset are already labeled as either positive or negative. The dataset contains the following two fields separated by a *tab* character:

1. **text:** Actual review comment on the movie.
2. **sentiment:** Positive sentiments are labelled as 1 and negative sentiments are labelled as 0.

10.2.1 | Loading the Dataset

Loading the data using pandas' *read_csv()* method is done as follows:

```
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings('ignore')

train_ds = pd.read_csv("sentiment_train", delimiter="\t")
train_ds.head(5)
```

First five records of loaded data are shown in Table 10.1.

TABLE 10.1 First five records of loaded data

	Sentiment	Text
0	1	The Da Vinci Code book is just awesome.
1	1	this was the first clive cussler i've ever read, but even books like Relic, and Da Vinci code were more plausible than this.
2	1	i liked the Da Vinci Code a lot.
3	1	i liked the Da Vinci Code a lot.
4	1	I liked the Da Vinci Code but it ultimatly didn't seem to hold it's own.

In Table 10.1, few of the texts may have been truncated while printing as the default column width is limited. This can be changed by setting *max_colwidth* parameter to increase the width size.

Each record or example in the column *text* is called a document. Use the following code to print the first five positive sentiment documents.

```
pd.set_option('max_colwidth', 800)
train_ds[train_ds.sentiment == 1][0:5]
```

Table 10.2 summarizes the first five positive sentiments. Sentiment value of 1 denotes positive sentiment.

TABLE 10.2 First five positive sentiments

	Sentiment	Text
0	1	The Da Vinci Code book is just awesome.
1	1	this was the first clive cussler i've ever read, but even books like Relic, and Da Vinci code were more plausible than this.
2	1	i liked the Da Vinci Code a lot.
3	1	i liked the Da Vinci Code a lot.
4	1	I liked the Da Vinci Code but it ultimatly didn't seem to hold it's own.

To print first five negative sentiment documents use

```
train_ds[train_ds.sentiment == 0][0:5]
```

Table 10.3 summarizes the first five negative sentiments. Sentiment value of 0 denotes negative sentiment.

TABLE 10.3 List of negative comments

	Sentiment	Text
3943	0	da vinci code was a terrible movie.
3944	0	Then again, the Da Vinci code is super shitty movie, and it made like 700 million.
3945	0	The Da Vinci Code comes out tomorrow, which sucks.
3946	0	i thought the da vinci code movie was really boring.
3947	0	God, Yahoo Games has this truly-awful looking Da Vinci Code-themed skin on it's chessboard right now.

In the next section, we will be discussing exploratory data analysis on text data.

10.2.2 | Exploring the Dataset

Exploratory data analysis can be carried out by counting the number of comments, positive comments, negative comments, etc. For example, we can check how many reviews are available in the dataset? Are the positive and negative sentiment reviews well represented in the dataset? Printing metadata of the DataFrame using *info()* method.

```
train_ds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6918 entries, 0 to 6917
Data columns (total 2 columns):
sentiment    6918 non-null int64
text         6918 non-null object
dtypes: int64(1), object(1)
memory usage: 108.2+ KB
```

From the output we can infer that there are 6918 records available in the dataset. We create a count plot (Figure 10.1) to compare the number of positive and negative sentiments.

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

plt.figure(figsize=(6,5))
# Create count plot
```

```
ax = sn.countplot(x='sentiment', data=train_ds)
# Annotate
for p in ax.patches:
        ax.annotate(p.get_height(), (p.get_x()+0.1,
        p.get_height()+50))
```

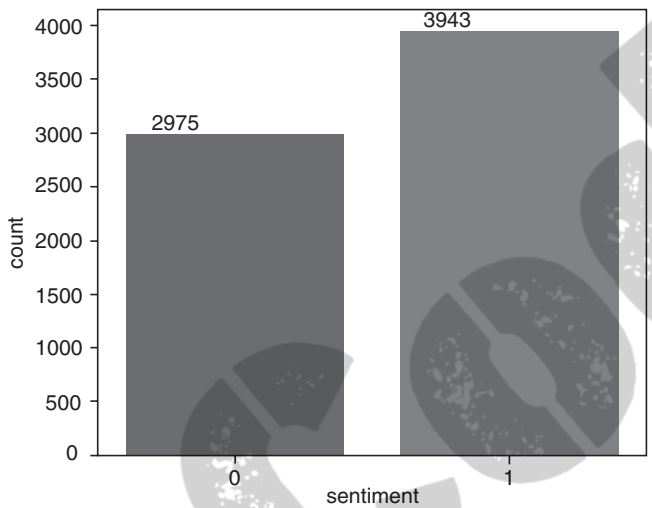

FIGURE 10.1 Number of records of positive and negative sentiments.

From Figure 10.1, we can infer that there are total 6918 records (feedback on movies) in the dataset. Out of 6918 records, 2975 records belong to negative sentiments, while 3943 records belong to positive sentiments. Thus, positive and negative sentiment documents have fairly equal representation in the dataset.

Before building the model, text data needs pre-processing for feature extraction. The following section explains step by step text pre-processing techniques.

10.2.3 | Text Pre-processing

Unlike structured data, features (independent variables) are not explicitly available in text data. Thus, we need to use a process to extract features from the text data. One way is to consider each word as a feature and find a measure to capture whether a word exists or does not exist in a sentence. This is called the bag-of-words (BoW) model. That is, each sentence (comment on a movie or a product) is treated as a bag of words. Each sentence (record) is called a document and collection of all documents is called corpus.

10.2.3.1 Bag-of-Words (BoW) Model

The first step in creating a BoW model is to create a dictionary of all the words used in the corpus. At this stage, we will not worry about grammar and only occurrence of the word is captured. Then we will

convert each document to a vector that represents words available in the document. There are three ways to identify the importance of words in a BoW model:

1. Count Vector Model
2. Term Frequency Vector Model
3. Term Frequency-Inverse Document Frequency (TF-IDF) Model

We will discuss these vector models in the following subsections.

Count Vector Model

Consider the following two documents:

1. **Document 1 (positive sentiment):** I really really like IPL.
2. **Document 2 (negative sentiment):** I never like IPL.

Note: IPL stands for Indian Premier League.

The complete vocabulary set (aka dictionary) for the above two documents will have words such as I, really, never, like, IPL. These words can be considered as features **(x1 through x5)**. For creating count vectors, we count the occurrence of each word in the document as shown in Table 10.4. The *y*-column in Table 10.4 indicates the sentiment of the statement: 1 for positive and 0 for negative sentiment.

TABLE 10.4 Count vector for document 1 and document 2

Documents	x1	x2	x3	x4	x5	y
	I	really	never	like	ipl	
I really really like ipl	1	2	0	1	1	1
I never like ipl	1	0	1	1	1	0

Term Frequency Vector Model

Term frequency (TF) vector is calculated for each document in the corpus and is the frequency of each term in the document. It is given by,

$$\text{Term Frequency } (TF_i) = \frac{\text{Number of occurrences of word } i \text{ in the document}}{\text{Total number of words in the document}} \tag{10.1}$$

where TF_i is the term frequency for word (aka token). TF representation for the two documents is shown in Table 10.5.

TABLE 10.5 TF vector

	x1	x2	x3	x4	x5	y
	I	really	never	like	ipl	
I really really like ipl	0.2	0.4	0	0.2	0.2	1
I never like ipl	0.25	0	0.25	0.25	0.25	0

Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF measures how important a word is to a document in the corpus. The importance of a word (or token) increases proportionally to the number of times a word appears in the document but is reduced by the frequency of the word present in the corpus. TF-IDF for a word *i* in the document is given by

$$TF - IDF_i = TF_i \times \ln\left(1 + \frac{N}{N_i}\right) \tag{10.2}$$

where N is the total number of documents in the corpus, N_i is the number of documents that contain word *i*.

The IDF value for each word for the above two documents is given in Table 10.6.

TABLE 10.6 IDF values

	x1	x2	x3	x4	x5
IDF Values	0.693	1.098	1.098	0.693	0.693

The TF-IDF values for the two documents are shown in Table 10.7

TABLE 10.7 TF-IDF values

	x1	x2	x3	x4	x5	y
	1	really	never	like	ipl	
I really really like ipl	0.1386	0.4394	0.0	0.1386	0.1386	1
I never like ipl	0.1732	0.0	0.2746	0.1732	0.1732	0

10.2.3.2 Creating Count Vectors for sentiment_train Dataset

Each document in the dataset needs to be transformed into TF or TF-IDF vectors. *sklearn.feature_extraction.text* module provides classes for creating both TF and TF-IDF vectors from text data. We will use *CountVectorizer* to create count vectors. In CountVectorizer, the documents will be represented by the number of times each word appears in the document.

We use the following code to process and create a dictionary of all words present across all the documents. The dictionary will contain all unique words across the corpus. And each word in the dictionary will be treated as feature.

```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize the CountVectorizer
count_vectorizer = CountVectorizer()
# Create the dictionary from the corpus
feature_vector = count_vectorizer.fit(train_ds.text)
# Get the feature names
features = feature_vector.get_feature_names()
print("Total number of features: ", len(features))
```

```
Total number of features: 2132
```

Total number of features or unique words in the corpus are 2132. The random sample of features can be obtained by using the following *random.sample()* method.

```
import random

random.sample(features, 10)
```

Let us look at some of the words randomly:

```
['surprised',
 'apart',
 'rosie',
 'dating',
 'dan',
 'outta',
 'local',
 'eating',
 'aka',
 'learn']
```

Using the above dictionary, we can convert all the documents in the dataset to count vectors using *transform*() method of count vectorizer:

```
train_ds_features = count_vectorizer.transform(train_ds.text)
type(train_ds_features)
```

The dimension of the DataFrame *train_ds_features,* that contains the count vectors of all the documents, is given by *shape* variable of the DataFrame.

```
train_ds_features.shape
```

```
(6918, 2132)
```

After converting the document into a vector, we will have a sparse matrix with 2132 features or dimensions. Each document is represented by a count vector of 2132 dimensions and if a specific word exists in a document, the corresponding dimension of the vector will be set to the count of that word in the document. But most of the documents have only few words in them, hence most of the dimensions in the vectors will have value set to 0. That is a lot of 0's in the matrix! So, the matrix is stored as a sparse matrix. Sparse matrix representation stores only the non-zero values and their index in the vector. This optimizes storage as well as computational needs. To know how many actual non-zero values are present in the matrix, we can use *getnnz()* method on the DataFrame.

```
train_ds_features.getnnz()
```

```
65398
```

Computing proportion of non-zero values with respect to zero values in the matrix can be obtained by dividing the number of non-zero values (i.e., 65398) by the dimension of the matrix (i.e., 6918 × 2132), that is, 65398/(6918 × 2132).

```
print("Density of the matrix: ",
       train_ds_features.getnnz() * 100 /
      (train_ds_features.shape[0] * train_ds_features.shape[1]))
```

```
Density of the matrix:      0.4434010415225908
```

The matrix has less than 1% non-zero values, that is, more than 99% values are zero values. This is a very sparse representation.

10.2.3.3 Displaying Document Vectors

To visualize the count vectors, we will convert this matrix into a *DataFrame* and set the column names to the actual feature names. The following commands are used for displaying the count vector:

```
# Converting the matrix to a dataframe
train_ds_df = pd.DataFrame(train_ds_features.todense())
# Setting the column names to the features i.e. words
train_ds_df.columns = features
```

Now, let us print the first record.

```
train_ds[0:1]
```

	Sentiment	Text
0	1	The Da Vinci Code book is just awesome.

We cannot print the complete vector as it has 2132 dimensions. Let us print the dimensions (words) from index 150 to 157. This index range contains the word *awesome*, which actually should have been encoded into 1.

```
train_ds_df.iloc[0:1, 150:157]
```

	Away	awesome	awesomely	awesomeness	awesomest	awful	awkward
0	0	1	0	0	0	0	0

The feature *awesome* is set to 1, while the other features are set to 0. Now select all the columns as per the words in the sentence and print below.

```
train_ds_df[['the', 'da', "vinci", "code", "book", 'is', 'just',
             'awesome']][0:1]
```

	the	da	vinci	code	book	is	Just	awesome
0	1	1	1	1	1	1	1	1

Yes, the features in the count vector are appropriately set to 1. The vector represents the sentence *"The Da Vinci Code book is just awesome"*.

10.2.3.4 Removing Low-frequency Words

One of the challenges of dealing with text is the number of words or features available in the corpus is too large. The number of features could easily go over tens of thousands. Some words would be common words and be present across most of the documents, while some words would be rare and present only in very few documents.

Frequency of each feature or word can be analyzed using histogram. To calculate the total occurrence of each feature or word, we will use *np.sum()* method.

```
# Summing up the occurrences of features column wise
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts_df = pd.DataFrame(dict(features = features,
                                      counts = features_counts))
```

The histogram in Figure 10.2 shows that a large number of features have very rare occurrences.

```
plt.figure(figsize=(12,5))
plt.hist(feature_counts_df.counts, bins=50, range = (0, 2000));
plt.xlabel('Frequency of words')
plt.ylabel('Density');
```

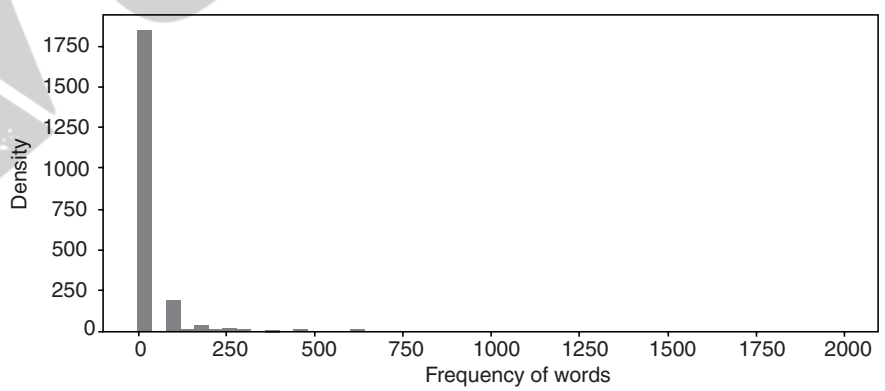

FIGURE 10.2 Histogram of frequently used words across the documents.

To find rare words in the dictionary, for example words that are present in any one of the document, we can filter the features by count equal to 1:

```
len(feature_counts_df[feature_counts_df.counts == 1])
```

```
1228
```

There are 1228 words which are present only once across all documents in the corpus. These words can be ignored. We can restrict the number of features by setting *max_features* parameters to 1000 while creating the count vectors.

```
# Initialize the CountVectorizer
count_vectorizer = CountVectorizer(max_features=1000)
# Create the dictionary from the corpus
feature_vector = count_vectorizer.fit(train_ds.text)
# Get the feature names
features = feature_vector.get_feature_names()
# Transform the document into vectors
train_ds_features = count_vectorizer.transform(train_ds.text)
# Count the frequency of the features
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features = features,
                                   counts = features_counts))
```

```
feature_counts.sort_values('counts',
                           ascending = False)[0:15]
```

Now print first 15 words and their count in descending order.

	Counts	Features
866	3306	The
37	2154	And
358	2093	Harry
675	2093	potter
138	2002	Code
934	2001	Vinci
178	2001	Da
528	2000	mountain
104	2000	brokeback
488	1624	Love
423	1520	Is

	Counts	Features
941	1176	Was
60	1127	awesome
521	1094	mission
413	1093	impossible

It can be noticed that the selected list of features contains words like **the**, **is**, **was**, **and**, etc. These words are irrelevant in determining the sentiment of the document. These words are called **stop words** and can be removed from the dictionary. This will reduce the number of features further.

10.2.3.5 Removing Stop Words

sklearn.feature_extraction.text provides a list of pre-defined stop words in English, which can be used as a reference to remove the stop words from the dictionary, that is, feature set.

```
from sklearn.feature_extraction import
text my_stop_words = text.ENGLISH_STOP_WORDS

#Printing first few stop words
print("Few stop words: ", list(my_stop_words)[0:10])
```

```
Few stop words: ['mill', 'another', 'every', 'whereafter',
'during', 'themselves', 'back', 'five', 'if', 'not']
```

Also, additional stop words can be added to this list for removal. For example, the movie names and the word "*movie*" itself can be a stop word in this case. These words can be added to the existing list of stop words for removal. For example,

```
# Adding custom words to the list of stop words
my_stop_words = text.ENGLISH_STOP_WORDS.union(['harry', 'potter',
'code', 'vinci', 'da','harry', 'mountain', 'movie', 'movies'])
```

10.2.3.6 Creating Count Vectors

All vectorizer classes take a list of stop words as a parameter and remove the stop words while building the dictionary or feature set. And these words will not appear in the count vectors representing the documents. We will create new count vectors by passing the *my_stop_words* as stop words list.

```
# Setting stop words list
count_vectorizer = CountVectorizer(stop_words = my_stop_words,
                                   max_features = 1000)
feature_vector = count_vectorizer.fit(train_ds.text)
```

```
train_ds_features = count_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features = features,
                                   counts = features_counts))
```

```
feature_counts.sort_values("counts", ascending = False)[0:15]
```

Print the first 15 words and their count in descending order.

	Counts	Features
73	2000	brokeback
408	1624	love
39	1127	awesome
436	1094	mission
341	1093	impossible
390	974	like
745	602	sucks
743	600	sucked
297	578	hate
652	374	really
741	365	stupid
362	287	just
374	276	know
742	276	suck
409	256	loved

It can be noted that the stop words have been removed. But we also notice another problem. Many words appear in multiple forms. For example, *love* and *loved.* The vectorizer treats the two words as two separate words and hence creates two separate features. But, if a word has similar meaning in all its form, we can use only the root word as a feature. **Stemming** and **Lemmatization** are two popular techniques that are used to convert the words into root words.

1. **Stemming:** This removes the differences between inflected forms of a word to reduce each word to its root form. This is done by mostly chopping off the end of words (suffix). For instance, *love* or *loved* will be reduced to the root word *love*. The root form of a word may not even be a real word. For example, *awesome* and *awesomeness* will be stemmed to *awesom*. One problem with stemming is that chopping of words may result in words that are not part of vocabulary

(e.g., awesom). PorterStemmer and LancasterStemmer are two popular algorithms for stemming, which have rules on how to chop off a word.

2. **Lemmatization:** This takes the morphological analysis of the words into consideration. It uses a language dictionary (i.e., English dictionary) to convert the words to the root word. For example, **stemming** would fail to differentiate between *man* and *men, while* lemmatization can bring these words to its original form *man.*

Natural Language Toolkit (NLTK) is a very popular library in Python that has an extensive set of features for natural language processing. NLTK supports *PorterStemmer*, *EnglishStemmer*, and *LancasterStemmer* for stemming, while *WordNetLemmatizer* for lemmatization.

These features can be used in *CountVectorizer*, while creating count vectors. We need to create a utility method, which takes documents, tokenizes it to create words, stems the words and remove the stop words before returning the final set of words for creating vectors.

```
from nltk.stem.snowball import PorterStemmer

stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()

#Custom function for stemming and stop word removal

def stemmed_words(doc):
    ### Stemming of words
    stemmed_words = [stemmer.stem(w) for w in analyzer(doc)]
    ### Remove the words in stop words list
    non_stop_words = [word for word in stemmed_words if not in my_
                      stop_words]
    return non_stop_words
```

CountVectorizer takes a custom analyzer for stemming and stop word removal, before creating count vectors. So, the custom function *stemmed_words()* is passed as an analyzer.

```
count_vectorizer = CountVectorizer(analyzer=stemmed_words,
                                   max_features = 1000)
feature_vector = count_vectorizer.fit(train_ds.text)
train_ds_features = count_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features = features,
                                   counts = features_counts))
feature_counts.sort_values("counts", ascending = False)[0:15]
```

Print the first 15 words and their count in descending order.

	Counts	Features
80	1930	brokeback
297	1916	harri
407	1837	love
803	1378	suck
922	1142	wa
43	1116	awesom
345	1090	imposs
433	1090	mission
439	1052	movi
393	823	like
299	636	hate
54	524	becaus
604	370	realli
796	364	stupid
379	354	know

It can be noted that words *love, loved, awesome* have all been stemmed to the root words.

10.2.3.7 Distribution of Words Across Different Sentiment

The words which have positive or negative meaning occur across documents of different sentiments. This could give an initial idea of how these words can be good features for predicting the sentiment of documents. For example, let us consider the word *awesome*.

```
# Convert the document vector matrix into dataframe
train_ds_df = pd.DataFrame(train_ds_features.todense())
# Assign the features names to the column
train_ds_df.columns = features
# Assign the sentiment labels to the train_ds
train_ds_df['sentiment'] = train_ds.sentiment
```

```
sn.barplot(x = 'sentiment', y = 'awesom', data = train_ds_df,
estimator=sum);
```

As show in Figure 10.3, the word *awesom* (stemmed word for *awesome*) appears mostly in positive sentiment documents. How about a neutral word like *realli*?

```
sn.barplot(x = 'sentiment', y = 'realli', data = train_ds_df,
estimator=sum);
```

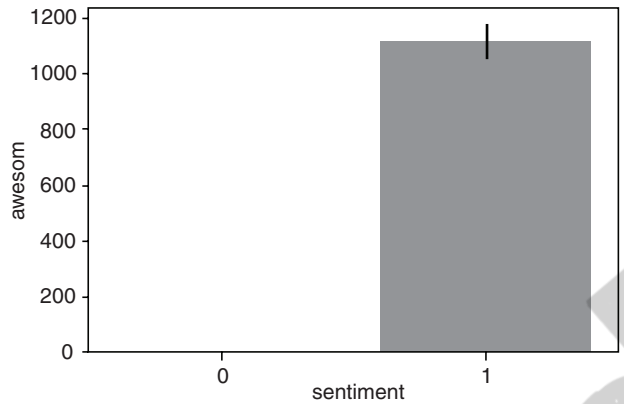

FIGURE 10.3 Frequency of word awesom in documents with positive versus negative sentiments.

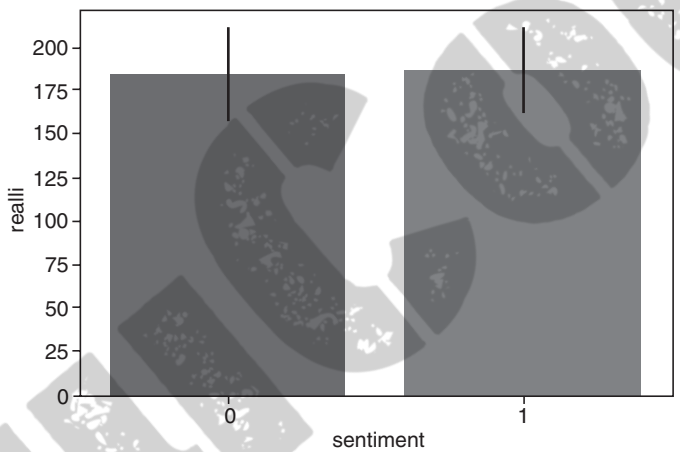

FIGURE 10.4 Frequency of word realli in documents with positive versus negative sentiments.

As shown in Figure 10.4, the word *realli* (stemmed word for *really*) occurs almost equally across positive and negative sentiments. How about the word *hate*?

```
sn.barplot(x = 'sentiment', y = 'hate', data = train_ds_df,
estimator=sum);
```

As shown in Figure 10.5, the word *hate* occurs mostly in negative sentiments than positive sentiments. This absolutely makes sense.

This gives us an initial idea that the words *awesom* and *hate* could be good features in determining sentiments of the document.

10.3 | NAÏVE–BAYES MODEL FOR SENTIMENT CLASSIFICATION

We will build a Naïve–Bayes model to classify sentiments. Naïve–Bayes classifier is widely used in Natural Language Processing and proved to give better results. It works on the concept of Bayes' theorem.

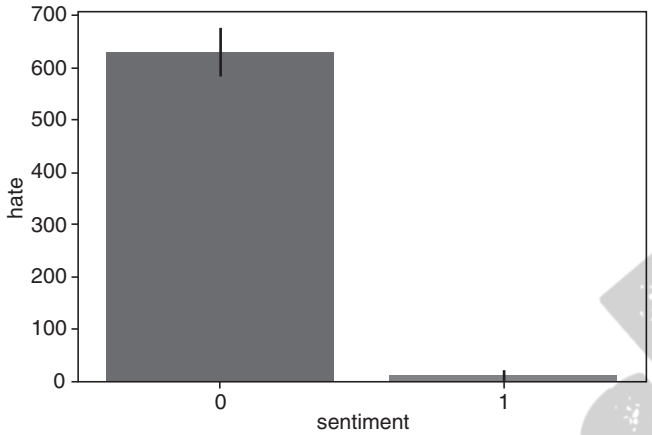

FIGURE 10.5 Frequency of word hate in documents with positive versus negative sentiments.

Assume that we would like to predict whether the probability of a document is positive (or negative) given that the document contains a word *awesome*. This can be computed if the probability of the word *awesome* appearing in a document given that it is a positive (or negative) sentiment multiplied by the probability of the document being positive (or negative).

$$P(doc = +ve \mid word = awesome) \propto P(word = awesome \mid doc = +ve) * P(doc = +ve)$$

The posterior probability of the sentiment is computed from the **prior** probabilities of all the words it contains. The assumption is that the occurrences of the words in a document are considered independent and they do not influence each other. So, if the document contains N words and words are represented as W_1, W_2, …., W_N, then

$$P(doc = +ve \mid word = W_1, W_2, \ldots W_N) \propto \prod_{i=1}^{N} P_i(word = W_i \mid doc = +ve) * P(doc = +ve)$$

sklearn.naive_bayes provides a class *BernoulliNB* which is a Naïve–Bayes classifier for multivariate Bernoulli models. BernoulliNB is designed for Binary/Boolean features (feature is either present or absent), which is the case here.

The steps involved in using Naïve–Bayes Model for sentiment classification are as follows:

1. Split dataset into train and validation sets.
2. Build the Naïve–Bayes model.
3. Find model accuracy.

We will discuss these in the following subsections.

10.3.1 | Split the Dataset

Split the dataset into 70:30 ratio for creating training and test datasets using the following code.

```
from sklearn.model_selection import train_test_split
```

```
train_X, test_X, train_y, test_y = train_test_split(train_ds_ features,
                                                    train_ds.sentiment,
                                                    test_size = 0.3,
                                                    random_state = 42)
```

10.3.2 | Build Naïve–Bayes Model

Build Naïve–Bayes model using the training set.

```
from sklearn.naive_bayes import BernoulliNB

nb_clf = BernoulliNB()
nb_clf.fit(train_X.toarray(), train_y)
```

```
BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None,
fit_prior=True)
```

10.3.3 | Make Prediction on Test Case

Predicted class will be the one which has the higher probability based on the Naïve–Bayes' probability calculation. Predict the sentiments of the test dataset using *predict()* method.

```
test_ds_predicted = nb_clf.predict(test_X.toarray())
```

10.3.4 | Finding Model Accuracy

Let us print the classification report.

```
from sklearn import metrics

print(metrics.classification_report(test_y, test_ds_predicted))
```

```
              Precision    recall     f1-score    support
           0       0.98       0.97       0.98        873
           1       0.98       0.99       0.98       1203
 avg / total       0.98       0.98       0.98       2076
```

The model is classifying with very high accuracy. Both average precision and recall is about 98% for identifying positive and negative sentiment documents. Let us draw the confusion matrix (Figure 10.6).

```
from sklearn import metrics

cm = metrics.confusion_matrix(test_y, test_ds_predicted)
sn.heatmap(cm, annot=True, fmt='.2f');
```

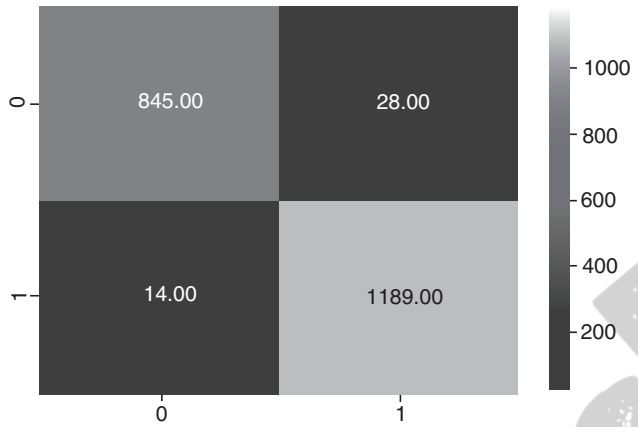

FIGURE 10.6 Confusion matrix of sentiment classification model.

In the confusion matrix, the rows represent the actual number positive and negative documents in the test set, whereas the columns represent what the model has predicted. Label 1 means positive sentiment and label 0 means negative sentiment. Figure 10.6 shows, as per the model prediction, that there are only 14 positive sentiment documents classified wrongly as negative sentiment documents (False Negatives) and there are only 28 negative sentiment documents classified wrongly as positive sentiment documents (False Positives). Rest all have been classified correctly.

10.4 | USING TF-IDF VECTORIZER

TfidfVectorizer is used to create both TF Vectorizer and TF-IDF Vectorizer. It takes a parameter *use_idf* (default *True*) to create TF-IDF vectors. If *use_idf* set to *False*, it will create only TF vectors and if it is set to *True*, it will create TF-IDF vectors.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(analyzer=stemmed_words,
                                   max_features = 1000)
feature_vector = tfidf_vectorizer.fit(train_ds.text)
train_ds_features = tfidf_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
```

TF-IDF are continuous values and these continuous values associated with each class can be assumed to be distributed according to Gaussian distribution. So, Gaussian Naïve–Bayes can be used to classify these documents. We will use *GaussianNB*, which implements the Gaussian Naïve–Bayes algorithm for classification.

```
from sklearn.naive_bayes import GaussianNB

train_X, test_X, train_y, test_y = train_test_split(train_ds_features,
                                                    train_ds.sentiment,
                                                    test_size = 0.3,
                                                    random_state = 42)
```

```
nb_clf = GaussianNB()
nb_clf.fit(train_X.toarray(), train_y)
```

```
GaussianNB(priors=None)
```

```
test_ds_predicted = nb_clf.predict(test_X.toarray())
print(metrics.classification_report(test_y, test_ds_predicted))
```

```
             Precision    Recall  f1-score   support

          0       0.96      0.96      0.96       873
          1       0.97      0.97      0.97      1203

avg/total       0.97      0.97      0.97      2076
```

The precision and recall seem to be pretty much same. The accuracy is very high in this example as the dataset is clean and carefully curated. But it may not be the case in the real world.

10.5 | CHALLENGES OF TEXT ANALYTICS

The text could be highly context-specific. The language people use to describe movies may not be the same for other products, say apparels. So, the training data needs to come from similar context (distribution) for building the model. The language may be highly informal. The language people use on social media may be a mix of languages or emoticons. The training data also needs to contain similar examples for learning. Bag-of-words model completely ignores the structure of the sentence or sequence of words in the sentence. This can be overcome to a certain extent by using n-grams.

10.5.1 | Using n-Grams

The models we built in this chapter, created features out of each token or word. But the meaning of some of the words might be dependent on the words it precedes or succeeds, for example *not happy*. It should be considered as one feature and not as two different features. n-gram is a contiguous sequence of *n* words. When two consecutive words are treated as one feature, it is called bigram**;** three consecutive words is called trigram and so on.

We will write a new custom analyzer *get_stemmed_tokens()*, which splits the sentences and stems the words from them before creating n-grams. The following code block removes non-alphabetic characters and then applies stemming.

```
from nltk.stem import PorterStemmer
# Library for regular expressions
import re

stemmer = PorterStemmer()

def get_stemmed_tokens(doc):
    # Tokenize the documents to words
    all_tokens = [word for word in nltk.word_tokenize(doc)]
    clean_tokens = []
```

```
    # Remove all characters other than alphabets. It takes a
    # regex for matching.
    for each_token in all_tokens:
        if re.search('[a-zA-Z]', each_token):
           clean_tokens.append(each_token)
    # Stem the words
    stemmed_tokens = [stemmer.stem(t) for t in clean_tokens]
    return stemmed_tokens
```

Now TfidfVectorizer takes the above method as a custom tokenizer. It also takes *ngram_range* parameter, which is a tuple value, for creating n-grams. A value (1, 2) means create features with one word and two consecutive words as features.

```
tfidf_vectorizer = TfidfVectorizer(max_features=500,
                                   stop_words='english',
                                   tokenizer=get_stemmed_okens,
                                   ngram_range=(1,2))

feature_vector = tfidf_vectorizer.fit(train_ds.text)
train_ds_features = tfidf_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
```

10.5.2 | Build the Model Using n-Grams

Split the dataset to 70:30 ratio for creating training and test datasets and then apply *BernoulliNB* for classification. We will apply the model to predict the test set and then print the classification report.

```
train_X, test_X, train_y, test_y = train_test_split(train_ds_features,
                                                    train_ds.sentiment,
                                                    test_size = 0.3,
                                                    random_state = 42)

nb_clf = BernoulliNB()
nb_clf.fit(train_X.toarray(), train_y)
test_ds_predicted = nb_clf.predict(test_X.toarray())
print(metrics.classification_report(test_y, test_ds_predicted))
```

```
              precision    recall     f1-score    support
           0       1.00      0.94       0.97        873
           1       0.96      1.00       0.98       1203
   avg/total       0.97      0.97       0.97       2076
```

The *recall* for identifying positive sentiment documents (with label 1) have increased to almost 1.0.