**Chapter 2. Data Mining Techniques Used in Recommender Systems**

Though the primary objective of this book is to build recommender systems, a walkthrough of the commonly used data-mining techniques is a necessary step before jumping into building recommender systems. In this chapter, you will learn about popular data preprocessing techniques, data-mining techniques, and data-evaluation techniques commonly used in recommender systems. The first section of the chapter tells you how a data analysis problem is solved, followed by data preprocessing steps such as similarity measures and dimensionality reduction. The next section of the chapter deals with data mining techniques and their evaluation techniques.

Similarity measures include:

* Euclidean distance
* Cosine distance
* Pearson correlation

Dimensionality reduction techniques include:

* Principal component analysis

Data-mining techniques include:

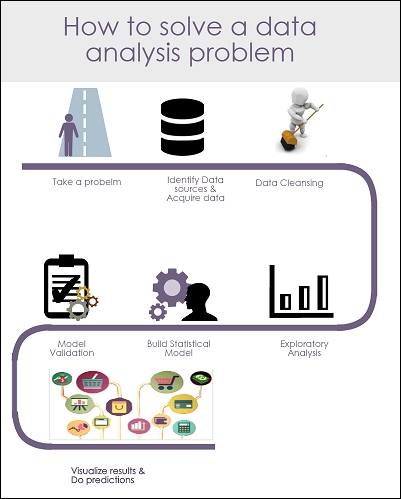
* k-means clustering
* Support vector machine
* Ensemble methods, such as bagging, boosting, and random forests

**Solving a data analysis problem**

Any data analysis problem involves a series of steps such as:

* Identifying a business problem.
* Understanding the problem domain with the help of a domain expert.
* Identifying data sources and data variables suitable for the analysis.
* Data preprocessing or a cleansing step, such as identifying missing values, quantitative and qualitative variables and transformations, and so on.
* Performing exploratory analysis to understand the data, mostly through visual graphs such as box plots or histograms.
* Performing basic statistics such as mean, median, modes, variances, standard deviations, correlation among the variables, and covariance to understand the nature of the data.
* Dividing the data into training and testing datasets and running a model using machine-learning algorithms with training datasets, using cross-validation techniques.
* Validating the model using the test data to evaluate the model on the new data. If needed, improve the model based on the results of the validation step.
* Visualize the results and deploy the model for real-time predictions.

The following image displays the resolution to a data analysis problem:



*Data analysis steps*

# Data preprocessing techniques

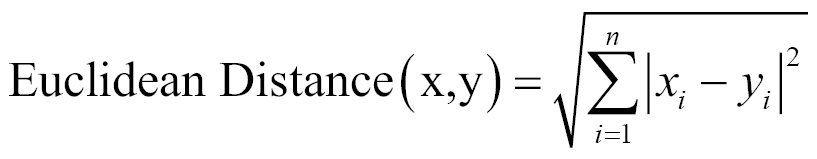
Data preprocessing is a crucial step for any data analysis problem. The model's accuracy depends mostly on the quality of the data. In general, any data preprocessing step involves data cleansing, transformations, identifying missing values, and how they should be treated. Only the preprocessed data can be fed into a machine-learning algorithm. In this section, we will focus mainly on data preprocessing techniques. These techniques include similarity measurements (such as Euclidean distance, Cosine distance, and Pearson coefficient) and dimensionality-reduction techniques, such as **Principal component analysis** (**PCA**), which are widely used in recommender systems. Apart from PCA, we have **singular value decomposition** (**SVD**), subset feature selection methods to reduce the dimensions of the dataset, but we limit our study to PCA.

## Similarity measures

As discussed in the previous chapter, every recommender system works on the concept of similarity between items or users. In this section, let's explore some similarity measures such as Euclidian distance, Cosine distance, and Pearson correlation.

### EUCLIDIAN DISTANCE

The simplest technique for calculating the similarity between two items is by calculating its Euclidian distance. The Euclidean distance between two points/objects (point x and point y) in a dataset is defined by the following equation:



In this equation, (x, y) are two consecutive data points, and n is the number of attributes for the dataset.

R script to calculate the Euclidean distance is as follows:

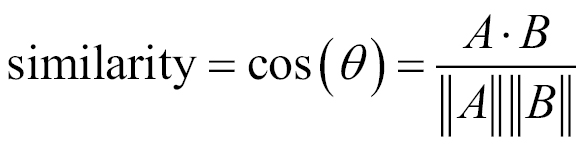
x1 <- rnorm(30)

x2 <- rnorm(30)

Euc\_dist = dist(rbind(x1,x2) ,method="euclidean")

### COSINE DISTANCE

Cosine similarity is a measure of similarity between two vectors of an inner product space thatmeasures the cosine of the angle between them. Cosine similarity is given by this equation:



R script to calculate the cosine distance is as follows:

vec1 = c( 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 )

vec2 = c( 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0 )

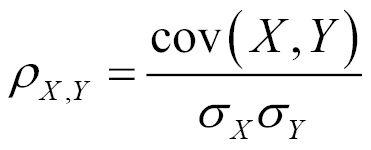
library(lsa)

cosine(vec1,vec2)

In this equation, x is the matrix containing all variables in a dataset. The cosine function is available in the lsa package.

### PEARSON CORRELATION

Similarity between two products can also be given by the correlation existing between their variables. Pearson's correlation coefficient is a popular correlation coefficient calculated between two variables as the covariance of the two variables divided by the product of their standard deviations. This is given by ƿ (rho):



R script is given by these lines of code:

Coef = cor(mtcars, method="pearson")

where mtcars is the dataset

Empirical studies showed that Pearson coefficient outperformed other similarity measures for user-based collaborative filtering recommender systems. The studies also show that Cosine similarity consistently performs well in item-based collaborative filtering.

## Dimensionality reduction

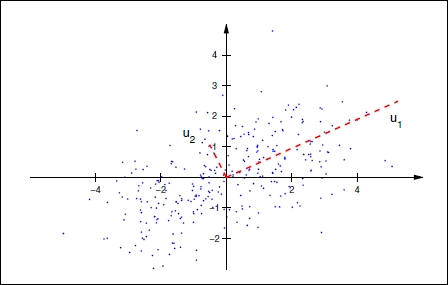
One of the most commonly faced problems while building recommender systems is high-dimensional and sparse data. At many times, we face a situation where we have a large set of features and fewer data points. In such situations, when we fit a model to the dataset, the predictive power of the model will be lower. This scenario is often termed as the curse of dimensionality. In general, adding more data points or decreasing the feature space, also known as dimensionality reduction, often reduces the effects of the curse of dimensionality. In this chapter, we will discuss PCA, a popular dimensionality reduction technique to reduce the effects of the curse of dimensionality.

### PRINCIPAL COMPONENT ANALYSIS

Principal component analysis is a classical statistical technique for dimensionality reduction. The PCAalgorithm transforms the data with high-dimensional space to a space with fewer dimensions. The algorithm linearly transforms m-dimensional input space to n-dimensional (n<m) output space, with the objective to minimize the amount of information/variance lost by discarding (m-n) dimensions. PCA allows us to discard the variables/features that have less variance.

Technically speaking, PCA uses orthogonal projection of highly correlated variables to a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables. This linear transformation is defined in such a way that the first principal component has the largest possible variance. It accounts for as much of the variability in the data as possible by considering highly correlated features. Each succeeding component in turn has the highest variance using the features that are less correlated with the first principal component and that are orthogonal to the preceding component.

Let's understand this in simple terms. Assume we have three dimensional data space with two features more correlated with each other than with the third. We now want to reduce the data to two-dimensional space using PCA. The first principal component is created in such a way that it explains maximum variance using the two correlated variables along the data. In the following graph, the first principal component (bigger line) is along the data explaining most variance. To choose the second principal component, we need to choose another line that has the highest variance, is uncorrelated, and is orthogonal to the first principal component. The implementation and technical details of PCA are beyond the scope of this book, so we will discuss how it is used in R.



We will illustrate PCA using the USArrests dataset. The USArrests dataset contains crime-relatedstatistics, such as Assault, Murder, Rape, and UrbanPop per 100,000 residents in 50 states in the US:

#PCA

data(USArrests)

head(states)

[1] "Alabama" "Alaska" "Arizona" "Arkansas" "California" "Colorado"

names(USArrests)

[1] "Murder" "Assault" "UrbanPop" "Rape"

#let us use apply() to the USArrests dataset row wise to calculate the variance to see how each variable is varying

apply(USArrests , 2, var)

Murder Assault UrbanPop Rape

18.97047 6945.16571 209.51878 87.72916

#We observe that Assault has the most variance. It is important to note at this point that

#Scaling the features is a very step while applying PCA.

#Applying PCA after scaling the feature as below

pca =prcomp(USArrests , scale =TRUE)

pca

**Standard deviations**:

[1] 1.5748783 0.9948694 0.5971291 0.4164494

**Rotation**:

PC1 PC2 PC3 PC4

Murder -0.5358995 0.4181809 -0.3412327 0.64922780

Assault -0.5831836 0.1879856 -0.2681484 -0.74340748

UrbanPop -0.2781909 -0.8728062 -0.3780158 0.13387773

Rape -0.5434321 -0.1673186 0.8177779 0.08902432

#Now lets us understand the components of pca output.

names(pca)

[1] "sdev" "rotation" "center" "scale" "x"

#Pca$rotation contains the principal component loadings matrix which explains

#proportion of each variable along each principal component.

#now let us learn interpreting the results of pca using biplot graph. Biplot is used to how the proportions of each variable along the two principal components.

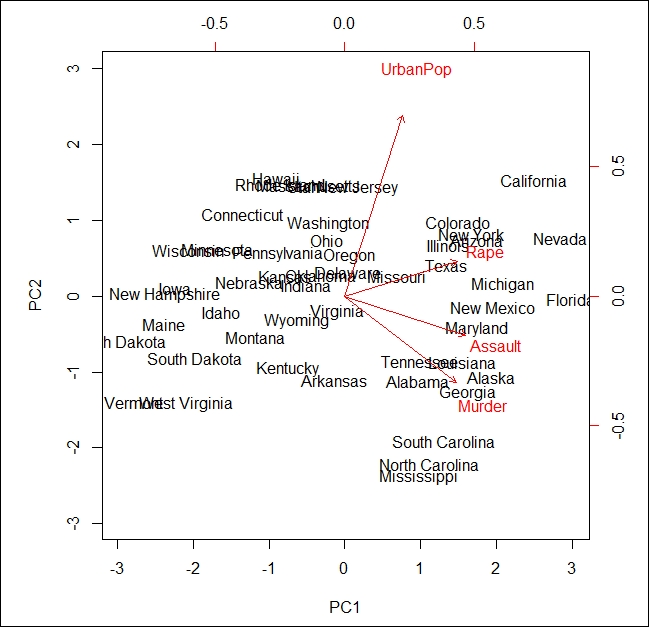
#below code changes the directions of the biplot, if we donot include the below two lines the plot will be mirror image to the below one.

pca$rotation=-pca$rotation

pca$x=-pca$x

biplot (pca , scale =0)

The output of the preceding code is as follows:



In the preceding image, known as a biplot, we can see the two principal components (**PC1** and **PC2**) of the USArrests dataset. The red arrows represent the loading vectors, which represent how the feature space varies along the principal component vectors.

From the plot, we can see that the first principal component vector, **PC1**, more or less places equal weight on three features: **Rape**, **Assault**, and **Murder**. This means that these three features are more correlated with each other than the **UrbanPop** feature. In the second principal component, **PC2** places more weight on **UrbanPop** than the remaining 3 features are less correlated with them.

# Data mining techniques

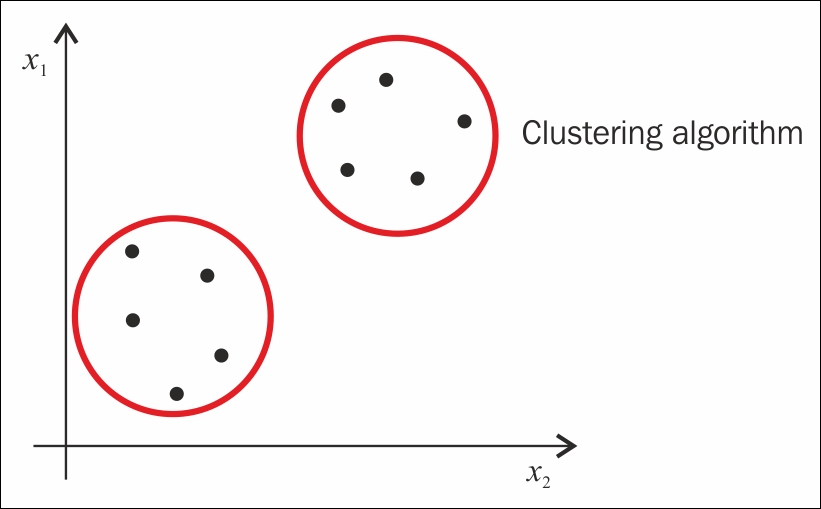
In this section, we will look at commonly used data-mining algorithms, such as k-means clustering, support vector machines, decision trees, bagging, boosting, and random forests. Evaluation techniques such as cross validation, regularization, confusion matrix, and model comparison are explained in brief.

# Cluster analysis

Cluster analysis is the process of grouping objects together in a way that objects in one group are more similar than objects in other groups.

An example would be identifying and grouping clients with similar booking activities on a travel portal, as shown in the following figure.

In the preceding example, each group is called a cluster, and each member (data point) of the cluster behaves in a manner similar to its group members.



*Cluster analysis*

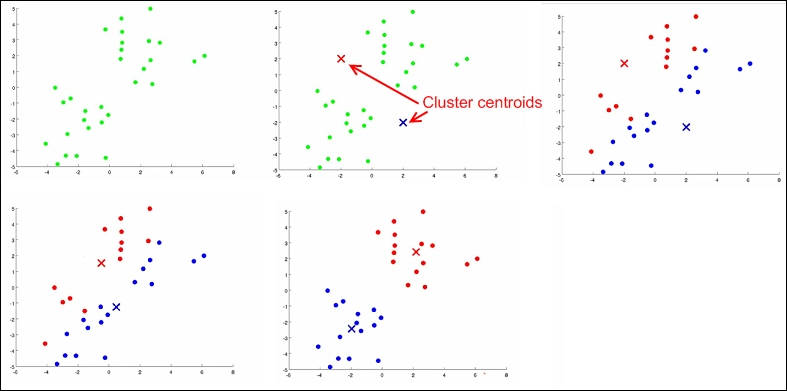
Cluster analysis is an unsupervised learning method. In supervised methods, such as regression analysis, we have input variables and response variables. We fit a statistical model to the input variables to predict the response variable. Whereas in unsupervised learning methods, however, we do not have any response variable to predict; we only have input variables. Instead of fitting a model to the input variables to predict the response variable, we just try to find patterns within the dataset. There are three popular clustering algorithms: hierarchical cluster analysis, k-means cluster analysis, and two-step cluster analysis. In the following section, we will learn about k-means clustering.

## Explaining the k-means cluster algorithm

k-means is an unsupervised, iterative algorithm where k is the number of clusters to be formed from the data. Clustering is achieved in two steps:

1. **Cluster assignment step**: In this step, we randomly choose two cluster points (red dot and green dot) and assign each data point to the cluster point that is closer to it (top part of the following image).
2. **Move centroid step**: In this step, we take the average of the points of all the examples in each group and move the centroid to the new position, that is, mean position calculated (bottom part of the following image).

The preceding steps are repeated until all the data points are grouped into two groups and the mean of the data points after moving the centroid doesn't change.



*Steps of cluster analysis*

The preceding image shows how a clustering algorithm works on data to form clusters. See the R implementation of k-means clustering on iris dataset as follows:

#k-means clustering

library(cluster)

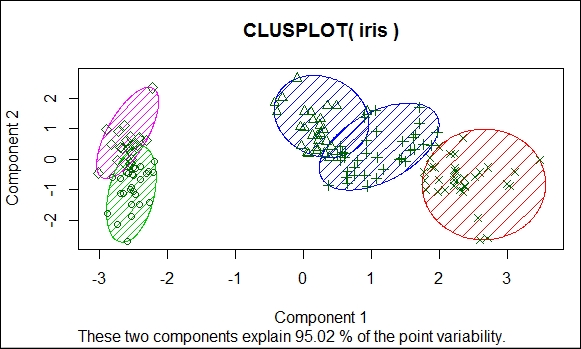
data(iris)

iris$Species = as.numeric(iris$Species)

kmeans<- kmeans(x=iris, centers=5)

clusplot(iris,kmeans$cluster, color=TRUE, shade=TRUE,labels=13, lines=0)

The output of the preceding code is as follows:



*Cluster analysis results*

The preceding image shows the formation of clusters on the iris data, and the clusters account for 95 percent of the data. In the preceding example, the number of clusters of k value is selected using the elbow method, as shown here:

library(cluster)

library(ggplot2)

data(iris)

iris$Species = as.numeric(iris$Species)

cost\_df <- data.frame()

for(i in 1:100){

kmeans<- kmeans(x=iris, centers=i, iter.max=50)

cost\_df<- rbind(cost\_df, cbind(i, kmeans$tot.withinss))

}

names(cost\_df) <- c("cluster", "cost")

#Elbow method to identify the idle number of Cluster

#Cost plot

ggplot(data=cost\_df, aes(x=cluster, y=cost, group=1)) +

theme\_bw(base\_family="Garamond") +

geom\_line(colour = "darkgreen") +

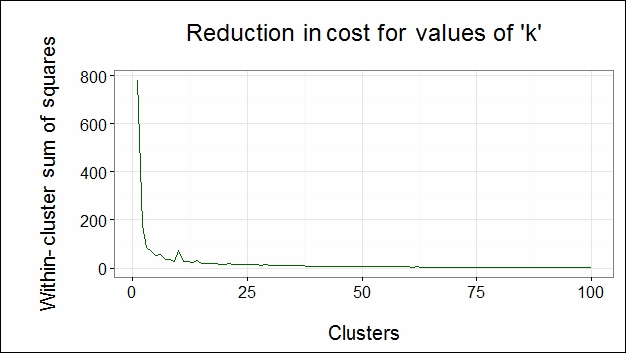
theme(text = element\_text(size=20)) +

ggtitle("Reduction In Cost For Values of 'k'\n") +

xlab("\nClusters") +

ylab("Within-Cluster Sum of Squares\n")

The following image shows the cost reduction for k values:

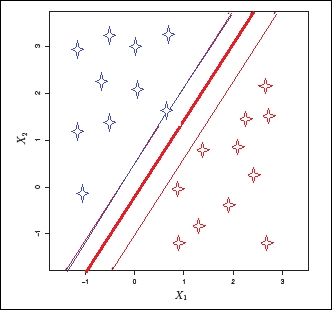


From the preceding figure, we can observe that the direction of the cost function is changed at cluster number 5. Hence, we choose 5 as our number of clusters k. Since the number of optimal clusters is found at the elbow of the graph, we call it the elbow method.

### SUPPORT VECTOR MACHINE

Support vector machine algorithms are a form of supervised learning algorithms employed to solve classification problems. SVM is generally treated as one of the best algorithms to deal with classification problems. Given a set of training examples, where each data point falls into one of two categories, an SVM training algorithm builds a model that assigns new data points into one category or the other. This model is a representation of the examples as a points in space, mapped so that the examples of the separate categories are divided by a margin that is as wide as possible, as shown in the following image. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. In this section, we will go through an overview and implementation of SVMs without going into mathematical details.

When SVM is applied to a p-dimensional dataset, the data is mapped to a p-1 dimensional hyperplane, and the algorithm finds a clear boundary with a sufficient margin between classes. Unlike other classification algorithms that also create a separating boundary to classify data points, SVM tries to choose a boundary that has the maximum margin to separate the classes, as shown in the following image:



Consider a two-dimensional dataset having two classes, as shown in the preceding image. Now, when the SVM algorithm is applied, first it checks whether a one-dimensional hyperplane exists to map all the data points. If the hyperplane exists, the linear classifier creates a decision boundary with a margin to separate the classes. In the preceding image, the thick red line is the decision boundary, and the thinner blue and red lines are the margins of each class from the boundary. When new test data is used to predict the class, the new data falls into one of the two classes.

Here are some key points to be noted:

* Though an infinite number of hyperplanes can be created, SVM chooses only one hyperplane that has the maximum margin, that is, the separating hyperplane that is farthest from the training observations.
* This classifier is only dependent on the data points that lie on the margins of the hyperplane, that is, on thin margins in the image, but not on other observations in the dataset. These points are called support vectors.
* The decision boundary is affected only by the support vectors but not by other observations located away from the boundaries. If we change the data points other than the support vectors, there would not be any effect on the decision boundary. However, if the support vectors are changed, the decision boundary changes.
* A large margin on the training data will also have a large margin on the test data to classify the test data correctly.
* Support vector machines also perform well with non-linear datasets. In this case, we use radial kernel functions.

See the R implementation of SVM on the iris dataset in the following code snippet. We used the e1071package to run SVM. In R, the SVM() function contains the implementation of support vector machines present in the e1071 package.

Now, we will see that the SVM() method is called with the tune() method, which does cross validation and runs the model on different values of the cost parameters.

The cross-validation method is used to evaluate the accuracy of the predictive model before testing on future unseen data:

#SVM

library(e1071)

data(iris)

sample = iris[sample(nrow(iris)),]

train = sample[1:105,]

test = sample[106:150,]

tune =tune(svm,Species~.,data=train,kernel ="radial",scale=FALSE,ranges =list(cost=c(0.001,0.01,0.1,1,5,10,100)))

tune$best.model

**Call**:

best.tune(method = svm, train.x = Species ~ ., data = train, ranges = list(cost = c(0.001,

0.01, 0.1, 1, 5, 10, 100)), kernel = "radial", scale = FALSE)

**Parameters**:

SVM-Type: C-classification

SVM-Kernel: radial

cost: 10

gamma: 0.25

Number of Support Vectors: 25

summary(tune)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:

cost

10

- best performance: 0.02909091

- Detailed performance results:

cost error dispersion

1 1e-03 0.72909091 0.20358585

2 1e-02 0.72909091 0.20358585

3 1e-01 0.04636364 0.08891242

4 1e+00 0.04818182 0.06653568

5 5e+00 0.03818182 0.06538717

6 1e+01 0.02909091 0.04690612

7 1e+02 0.07636364 0.08679584

model =svm(Species~.,data=train,kernel ="radial",cost=10,scale=FALSE)

// cost =10 is chosen from summary result of tune variable

The tune$best.model object tells us that the model works best with the cost parameter as 10 and total number of support vectors as 25:

pred = predict(model,test)

**Decision trees**

Decision trees are a simple, fast, tree-based supervised learning algorithm to solve classification problems. Though not very accurate when compared to other logistic regression methods, this algorithm comes in handy while dealing with recommender systems.

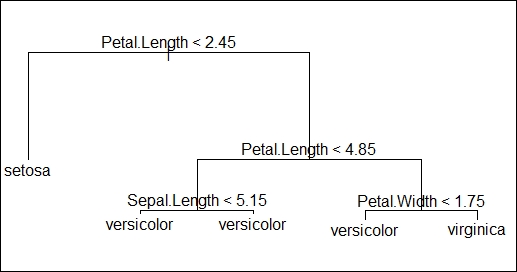
We define the decision trees with an example. Imagine a situation where you have to predict the class of flower based on its features such as petal length, petal width, sepal length, and sepal width. We will apply the decision tree methodology to solve this problem:

1. Consider the entire data at the start of the algorithm.
2. Now, choose a suitable question/variable to divide the data into two parts. In our case, we chose to divide the data based on petal length *> 2.45* and *<= 2.45*. This separates flower class setosa from the rest of the classes.
3. Now, further divide the data having petal length *>2.45*, based on the same variable with petal length *< 4.5* and *>= 4.5*, as shown in the following image.
4. This splitting of the data will be further divided by narrowing down the data space until we reach a point where all the bottom points represent the response variables or where further logical split cannot be done on the data.

In the following decision tree image, we have one root node, four internal nodes where data split occurred, and five terminal nodes where data split cannot be done any further. They are defined as follows:

* **Petal.Length <2.45** as root node
* **Petal.Length <4.85**, **Sepal.Length <5.15,** and **Petal.Width <1.75** are called internal nodes
* Final nodes having the class of the flowers are called terminal nodes
* The lines connecting the nodes are called the branches of the tree

While predicting responses on new data using the previously built model, each new data point is taken through each node, a question is asked, and a logical path is taken to reach its logical class, as shown in the following figure:



See the decision tree implementation in R on the iris dataset using the tree package available from **Comprehensive R Archive Network** (**CRAN**).

The summary of the mode is given here. It tells us that the misclassification rate is 0.0381, indicating that the model is accurate:

library(tree)

data(iris)

sample = iris[sample(nrow(iris)),]

train = sample[1:105,]

test = sample[106:150,]

model = tree(Species~.,train)

summary(model)

**Classification tree**:

tree(formula = Species ~ ., data = train, x = TRUE, y = TRUE)

Variables actually used in tree construction:

[1] "Petal.Length" "Sepal.Length" "Petal.Width"

Number of terminal nodes: 5

Residual mean deviance: 0.1332 = 13.32 / 100

Misclassification error rate: 0.0381 = 4 / 105 '

//plotting the decision tree

plot(model)text(model)

pred = predict(model,test[,-5],type="class")

> pred

[1] setosa setosa virginica setosa setosa setosa versicolor

[8] virginica virginica setosa versicolor versicolor virginica versicolor

[15] virginica virginica setosa virginica virginica versicolor virginica

[22] versicolor setosa virginica setosa versicolor virginica setosa

[29] versicolor versicolor versicolor virginica setosa virginica virginica

[36] versicolor setosa versicolor setosa versicolor versicolor setosa

[43] versicolor setosa setosa

Levels: setosa versicolor virginica

# Ensemble methods

In data mining, we use ensemble methods, which means using multiple learning algorithms to obtain better predictive results than applying any single learning algorithm on any statistical problem. This section will provide an overview of popular ensemble methods such as bagging, boosting, and random forests

## Bagging

Bagging is also known as Bootstrap aggregating. It is designed to improve the stability and accuracy of machine-learning algorithms. It helps avoid over fitting and reduces variance. This is mostly used with decision trees.

Bagging involves randomly generating Bootstrap samples from the dataset and trains the models individually. Predictions are then made by aggregating or averaging all the response variables:

* For example, consider a dataset (Xi, Yi), where i=1 …n, contains n data points.
* Now, randomly select B samples with replacements from the original dataset using Bootstrap technique.
* Next, train the B samples with regression/classification models independently. Then, predictions are made on the test set by averaging the responses from all the B models generated in the case of regression. Alternatively, the most often occurring class among B samples is generated in the case of classification.

## Random forests

Random forests are improvised supervised algorithms than bootstrap aggregation or bagging methods, though they are built on a similar approach. Unlike selecting all the variables in all the B samples generated using the Bootstrap technique in bagging, we select only a few predictor variables randomly from the total variables for each of the B samples. Then, these samples are trained with the models. Predictions are made by averaging the result of each model. The number of predictors in each sample is decided using the formula m = √p, where p is the total variable count in the original dataset.

Here are some key notes:

* This approach removes the condition of dependency of strong predictors in the dataset as we intentionally select fewer variables than all the variables for every iteration
* This approach also de-correlates variables, resulting in less variability in the model and, hence, more reliability

Refer to the R implementation of random forests on the iris dataset using the randomForest package available from CRAN:

#randomForest

library(randomForest)

data(iris)

sample = iris[sample(nrow(iris)),]

train = sample[1:105,]

test = sample[106:150,]

model =randomForest(Species~.,data=train,mtry=2,importance =TRUE,proximity=TRUE)

model

**Call**:

randomForest(formula = Species ~ ., data = train, mtry = 2, importance = TRUE, proximity = TRUE)

Type of random forest: classification

Number of trees: 500

No. of variables tried at each split: 2

OOB estimate of error rate: 5.71%

Confusion matrix:

setosa versicolor virginica class.error

setosa 40 0 0 0.00000000

versicolor 0 28 3 0.09677419

virginica 0 3 31 0.08823529

pred = predict(model,newdata=test[,-5])

pred

pred

119 77 88 90 51 20 96

virginica versicolor versicolor versicolor versicolor setosa versicolor

1 3 118 127 6 102 5

setosa setosa virginica virginica setosa virginica setosa

91 8 23 133 17 78 52

versicolor setosa setosa virginica setosa virginica versicolor

63 82 84 116 70 50 129

versicolor versicolor virginica virginica versicolor setosa virginica

150 34 9 120 41 26 121

virginica setosa setosa virginica setosa setosa virginica

145 138 94 4 104 81 122

virginica virginica versicolor setosa virginica versicolor virginica

18 105 100

setosa virginica versicolor

Levels: setosa versicolor virginica

## Boosting

Unlike with bagging, where multiple copies of Bootstrap samples are created, a new model is fitted foreach copy of the dataset, and all the individual models are combined to create a single predictive model, each new model is built using information from previously built models. Boosting can be understood as an iterative method involving two steps:

* A new model is built on the residuals of previous models instead of the response variable
* Now, the residuals are calculated from this model and updated to the residuals used in the previous step

The preceding two steps are repeated for multiple iterations, allowing each new model to learn from its previous mistakes, thereby improving the model accuracy:

#Boosting in R

library(gbm)

data(iris)

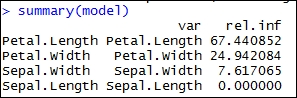
sample = iris[sample(nrow(iris)),]

train = sample[1:105,]

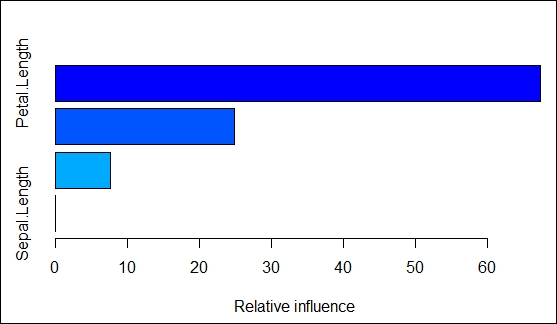
test = sample[106:150,]

model = gbm(Species~.,data=train,distribution="multinomial",n.trees=5000,interaction.depth=4)

summary(model)



The output of the preceding code is as follows:



In the following code snippet, the output value for the predict() function is used in the apply()function to pick the response with the highest probability among each row in the pred matrix. Theresultant output from the apply() function is the prediction for the response variable:

//the preceding summary states the relative importance of the variables of the model.

pred = predict(model,newdata=test[,-5],n.trees=5000)

pred[1:5,,]

setosa versicolor virginica

[1,] 5.630363 -2.947531 -5.172975

[2,] 5.640313 -3.533578 -5.103582

[3,] -5.249303 3.742753 -3.374590

[4,] -5.271020 4.047366 -3.770332

[5,] -5.249324 3.819050 -3.439450

//pick the response with the highest probability from the resulting pred matrix, by doing apply(.., 1, which.max) on the vector output from prediction.

p.pred <- apply(pred,1,which.max)

p.pred

[1] 1 1 3 3 2 2 3 1 3 1 3 2 2 1 2 3 2 2 3 3 1 1 3 1 3 3 3 1 1 2 2 2 2 2 2 2 1 1 3 1 2

[42] 1 3 2 3

**Evaluating data-mining algorithms**

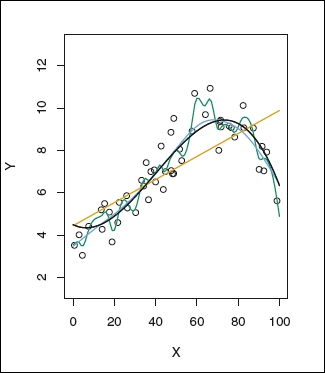
In the previous sections, we have seen various data-mining techniques used in recommender systems. In this section, you will learn how to evaluate models built using data-mining techniques. The ultimate goal for any data analytics model is to perform well on future data. This objective could be achieved only if we build a model that is efficient and robust during the development stage.

While evaluating any model, the most important things we need to consider are as follows:

* Whether the model is over fitting or under fitting
* How well the model fits the future data or test data

Under fitting, also known as bias, is a scenario when the model doesn't even perform well on training data. This means that we fit a less robust model to the data. For example, say the data is distributed non-linearly and we are fitting the data with a linear model. From the following image, we see that data is non-linearly distributed. Assume that we have fitted a linear model (orange line). In this case, during the model building stage itself, the predictive power will be low.

Over fitting is a scenario when the model performs well on training data, but does really bad on test data. This scenario arises when the model memorizes the data pattern rather than learning from data. For example, say the data is distributed in a non-linear pattern, and we have fitted a complex model, shown using the green line. In this case, we observe that the model is fitted very close to the data distribution, taking care of all the ups and downs. In this case, the model is most likely to fail on previously unseen data.



The preceding image shows simple, complex, and appropriate fitted models' training data. The green fit represents overfitting, the orange line represents underfitting, the black and blue lines represent theappropriate model, which is a trade-off between underfit and overfit.

Any fitted model is evaluated to avoid previously mentioned scenarios using cross validation, regularization, pruning, model comparisons, ROC curves, confusion matrices, and so on .

**Cross validation**: This is a very popular technique for model evaluation for almost all models. In this technique, we divide the data into two datasets: a training dataset and a test dataset. The model is built using the training dataset and evaluated using the test dataset. This process is repeated many times. The test errors are calculated for every iteration. The averaged test error is calculated to generalize the model accuracy at the end of all the iterations.

**Regularization**: In this technique, the data variables are penalized to reduce the complexity of the model with the objective to minimize the cost function. There are two most popular regularization techniques: ridge regression and lasso regression. In both techniques, we try to reduce the variable co-efficient to zero. Thus, a smaller number of variables will fit the data optimally.

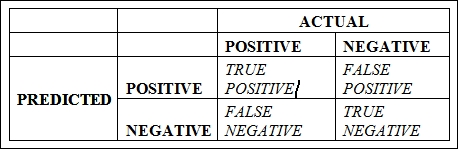
**Confusion matrix**: This technique is popularly used in evaluating a classification model. We build a confusion matrix using the results of the model. We calculate precision and recall/sensitivity/specificity to evaluate the model.

**Precision**: This is the probability whether the truly classified records are relevant.

**Recall/Sensitivity**: This is the probability whether the relevant records are truly classified.

**Specificity**: Also known as true negative rate, this is the proportion of truly classified wrong records.

A confusion matrix shown in the following image is constructed using the results of classification models discussed in the previous section:



Let's understand the confusion matrix:

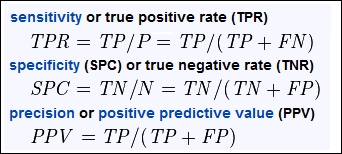
**TRUE POSITVE (TP)**: This is a count of all the responses where the actual response is negative and the model predicted is positive

**FALSE POSITIVE (FP)**: This is a count of all the responses where the actual response is negative, but the model predicted is positive. It is, in general, a **FALSE ALARM**.

**FALSE NEGATIVE (FN)**: This is a count of all the responses where the actual response is positive, but the model predicted is negative. It is, in general, **A MISS**.

**TRUE NEGATIVE (TN)**: This is a count of all the responses where the actual response is negative, and the model predicted is negative.

Mathematically, precision and recall/specificity is calculated as follows:



**Model comparison**: A classification problem can be solved using one or more statistical models. For example, a classification problem can be solved using logistic regression, a decision tree, ensemblemethods, and SVM. How do you choose which model fits the data well? A number of approaches are available for a suitable model selection, such as **Akaike information criteria** (**AIC**), **Bayesianinformation criteria** (**BIC**), and Adjusted R^2, Cᵨ. For each model, AIC / BIC / Adjusted R^2 is calculated. The model with least of these values is selected as the best model.

# Chapter 3. Recommender Systems

This chapter shows some popular recommendation techniques. In addition, we will implement some of them in R.

We will deal with the following techniques:

* **Collaborative filtering**: This is the branch of techniques that we will explore in detail. The algorithms are based on information about similar users or similar items. The two sub-branches are as follows:
  + **Item-based collaborative filtering**: This recommends to a user the items that are most similar to the user's purchases
  + **User-based collaborative filtering**: This recommends to a user the items that are the most preferred by similar users
* **Content-based filtering**: This is for each user; it defines a user profile and identify the items that match it.
* **Hybrid filtering**: This combines different techniques.
* **Knowledge-based filtering**: This is uses explicit knowledge about users and items.

# R package for recommendation – recommenderlab

In this chapter, we will build recommender systems using recommenderlab, which is an R package for collaborative filtering. This section will present a quick overview of this package. First, let's install it, if we haven't done so already:

if(!"recommenderlab" %in% rownames(installed.packages())){install.packages("recommenderlab")}

Now, we can load the package. Then, using the help function, we can take a look at its documentation:

library("recommenderlab")

help(package = "recommenderlab")

When we run the preceding command in RStudio, a help file containing some links and a list of functions will open.

The examples that you will see in this chapter contain some random components. In order to be able to reproduce the code obtaining the same output, we need to run this line:

set.seed(1)

We are now ready to start exploring recommenderlab.

## Datasets

Like many other R packages, recommenderlab contains some datasets that can be used to play around with the functions:

data\_package <- data(package = "recommenderlab")

data\_package$results[, "Item"]

### JESTER5K, MSWEB, AND MOVIELENSE

In our examples, we will use the MovieLense dataset; the data is about movies. The table contains the ratings that the users give to movies. Let's load the data and take a look at it:

data(MovieLense)

MovieLense

## 943 x 1664 rating matrix of class 'realRatingMatrix' with 99392 ratings.

Each row of MovieLense corresponds to a user, and each column corresponds to a movie. There are more than 943 x 1664 = 1,500,000 combinations between a user and a movie. Therefore, storing the complete matrix would require more than 1,500,000 cells. However, not every user has watched every movie. Therefore, there are fewer than 100,000 ratings, and the matrix is sparse. The recommenderlab package allows us to store it in a compact way.

## The class for rating matrices

In this section, we will explore MovieLense in detail:

class(MovieLense)

## [1] "realRatingMatrix"

## attr(,"package")

## [1] "recommenderlab"

The realRatingMatrix class is defined by recommenderlab, and ojectsojectsb contains sparse rating matrices. Let's take a look at the methods that we can apply on the objects of this class:

methods(class = class(MovieLense))

|  |  |  |
| --- | --- | --- |
| [ | dimnames<- | Recommender |
| binarize | dissimilarity | removeKnownRatings |
| calcPredictionAccuracy | evaluationScheme | rowCounts |
| calcPredictionAccuracy | getData.frame | rowMeans |
| colCounts | getList | rowSds |
| colMeans | getNormalize | rowSums |
| colSds | getRatings | sample |
| colSums | getTopNLists | show |
| denormalize | image | similarity |
| dim | normalize |  |
| dimnames | nratings |  |

Some methods that are applicable to matrices have been redefined in a more optimized way. For instance, we can use dim to extract the number of rows and columns, and colSums to compute the sum of each column. In addition, there are new methods that are specific for recommendation systems.

Usually, rating matrices are sparse matrices. For this reason, the realRatingMatrix class supports a compact storage of sparse matrices. Let's compare the size of MovieLense with the corresponding R matrix:

object.size(MovieLense)

## 1388448 bytes

object.size(as(MovieLense, "matrix"))

## 12740464 bytes

We can compute how many times the recommenderlab matrix is more compact:

object.size(as(MovieLense, "matrix")) / object.size(MovieLense)

## 9.17604692433566 bytes

As expected, MovieLense occupies much less space than the equivalent standard R matrix. The rate is about 1:9, and the reason is the sparsity of MovieLense. A standard R matrix object stores all the missing values as 0s, so it stores 15 times more cells.

## Computing the similarity matrix

Collaborative filtering algorithms are based on measuring the similarity between users or between items. For this purpose, recommenderlab contains the similarity function. The supportedmethods to compute similarities are cosine, pearson, and jaccard.

For instance, we might want to determine how similar the first five users are with each other. Let's compute this using the cosine distance:

similarity\_users <- similarity(MovieLense[1:4, ], method = "cosine", which = "users")

The similarity\_users object contains all the dissimilarities. Let's explore it:

class(similarity\_users)

## [1] "dist"

As expected, similarity\_users is an object containing distances. Since dist is a base R class, we can use it in different ways. For instance, we could use hclust to build a hierarchic clustering model.

We can also convert similarity\_users into a matrix and visualize it:

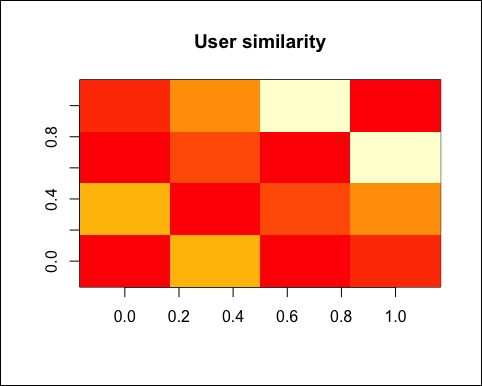
as.matrix(similarity\_users)

| **1** | **2** | **3** | **4** |
| --- | --- | --- | --- |
| 0 | 0.1689 | 0.03827 | 0.06635 |
| 0.1689 | 0 | 0.09707 | 0.1531 |
| 0.03827 | 0.09707 | 0 | 0.3334 |
| 0.06635 | 0.1531 | 0.3334 | 0 |

Using image, we can visualize the matrix. Each row and each column corresponds to a user, and each cell corresponds to the similarity between two users:

image(as.matrix(similarity\_users), main = "User similarity")

The more red the cell is, the more similar two users are. Note that the diagonal is red, since it's comparing each user with itself:



Using the same approach, we can compute and visualize the similarity between the first four items:

similarity\_items <- similarity(MovieLense[, 1:4], method = "cosine", which = "items")

as.matrix(similarity\_items)

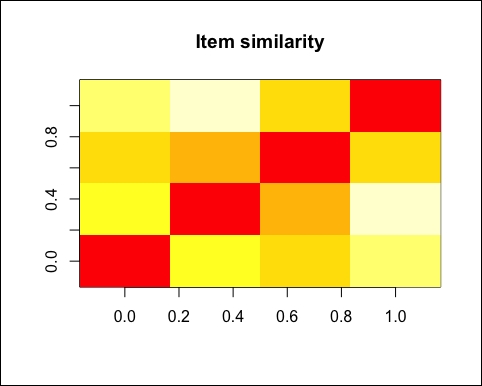
|  | **Toy Story (1995)** | **GoldenEye (1995)** |
| --- | --- | --- |
| **Toy Story (1995)** | 0 | 0.4024 |
| **GoldenEye (1995)** | 0.4024 | 0 |
| **Four Rooms (1995)** | 0.3302 | 0.2731 |
| **Get Shorty (1995)** | 0.4549 | 0.5026 |

The table continues as follows:

|  | **Four Rooms (1995)** | **Get Shorty (1995)** |
| --- | --- | --- |
| **Toy Story (1995)** | 0.3302 | 0.4549 |
| **GoldenEye (1995)** | 0.2731 | 0.5026 |
| **Four Rooms (1995)** | 0 | 0.3249 |
| **Get Shorty (1995)** | 0.3249 | 0 |

Similar to the preceding screenshot, we can visualize the matrix using this image:

image(as.matrix(similarity\_items), main = "Item similarity")



The similarity is the base of collaborative filtering models.

## Recommendation models

The recommenderlab package contains some options for the recommendation algorithm. We can display the model applicable to the realRatingMatrix objects using recommenderRegistry$get\_entries:

recommender\_models <- recommenderRegistry$get\_entries(dataType = "realRatingMatrix")

The recommender\_models object contains some information about the models. First, let's see which models we have:

names(recommender\_models)

| **Models** |
| --- |
| IBCF\_realRatingMatrix |
| PCA\_realRatingMatrix |
| POPULAR\_realRatingMatrix |
| RANDOM\_realRatingMatrix |
| SVD\_realRatingMatrix |
| UBCF\_realRatingMatrix |

Let's take a look at their descriptions:

lapply(recommender\_models, "[[", "description")

## $IBCF\_realRatingMatrix

## [1] "Recommender based on item-based collaborative filtering (real data)."

##

## $PCA\_realRatingMatrix

## [1] "Recommender based on PCA approximation (real data)."

##

## $POPULAR\_realRatingMatrix## [1] "Recommender based on item popularity (real data)."

##

## $RANDOM\_realRatingMatrix

## [1] "Produce random recommendations (real ratings)."

##

## $SVD\_realRatingMatrix

## [1] "Recommender based on SVD approximation (real data)."

##

## $UBCF\_realRatingMatrix

## [1] "Recommender based on user-based collaborative filtering (real data)."

Out of them, we will use IBCF and UBCF.

The recommender\_models object also contains some other information, such as its parameters:

recommender\_models$IBCF\_realRatingMatrix$parameters

| **Parameter** | **Default** |
| --- | --- |
| k | 30 |
| method | Cosine |
| normalize | center |
| normalize\_sim\_matrix | FALSE |
| alpha | 0.5 |
| na\_as\_zero | FALSE |

For a more detailed description of the package and some use cases, you can take a look at the package vignette. You can find all the material by typing help(package = "recommenderlab").

The recommenderlab package is a good and flexible package to perform recommendation. If we combine its models with other R tools, we will have a powerful framework to explore the data and build recommendation models.

In the next section, we will explore a dataset of recommenderlab using some of its tools.

# Data exploration

In this section, we will explore the MovieLense dataset. For this purpose, we will use recommenderlab to build recommender systems and ggplot2 to visualize their results. Let's load the packages and the data:

library("recommenderlab")

library("ggplot2")

data(MovieLense)

class(MovieLense)

## [1] "realRatingMatrix"

## attr(,"package")

## [1] "recommenderlab"

MovieLense is a realRatingMatrix object containing a dataset about movie ratings. Each row corresponds to a user, each column to a movie, and each value to a rating.

## Exploring the nature of the data

Let's take a quick look at MovieLense. As explained in the previous section, there are some generic methods that can be applied to realRatingMatrix objects. We can extract their size using dim:

dim(MovieLense)

## [1] 943 1664

There are 943 users and 1664 movies. Since realRatingMatrix is an S4 class, the components of the objects are contained in MovieLense slots. We can see all the slots using slotNames, which displays all the data stored within an object:

slotNames(MovieLense)

## [1] "data" "normalize"

MovieLense contains a data slot. Let's take a look at it.

class(MovieLense@data)

## [1] "dgCMatrix"

## attr(,"package")

## [1] "Matrix"

dim(MovieLense@data)

## [1] 943 1664

MovieLense@data belongs to the dgCMatrix class that inherits from Matrix. In order to perform custom data exploration, we might need to access this slot.

## Exploring the values of the rating

Starting from the slot data, we can explore the matrix. Let's take a look at the ratings. We can convert the matrix into a vector and explore its values:

vector\_ratings <- as.vector(MovieLense@data)

unique(vector\_ratings)

## [1] 5 4 0 3 1 2

The ratings are integers in the range 0-5. Let's count the occurrences of each of them.

table\_ratings <- table(vector\_ratings)

table\_ratings

| **Rating** | **Occurrences** |
| --- | --- |
| 0 | 1469760 |
| 1 | 6059 |
| 2 | 11307 |
| 3 | 27002 |
| 4 | 33947 |
| 5 | 21077 |

According to the documentation, a rating equal to 0 represents a missing value, so we can remove them from vector\_ratings:

vector\_ratings <- vector\_ratings[vector\_ratings != 0]

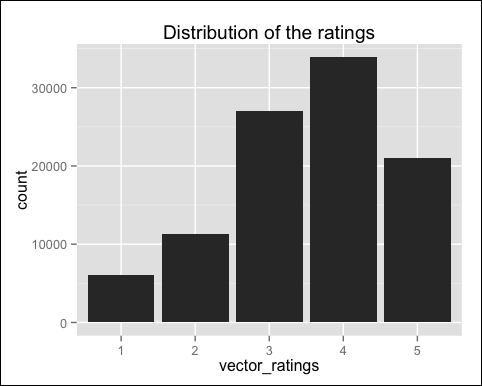
Now, we can build a frequency plot of the ratings. In order to visualize a bar plot with frequencies, we can use ggplot2. Let's convert them into categories using factor and build a quick chart:

vector\_ratings <- factor(vector\_ratings)

Let's visualize their distribution using qplot:

qplot(vector\_ratings) + ggtitle("Distribution of the ratings")

The following image shows the distribution of the ratings:



Most of the ratings are above **2,** and the most common is **4**.

## Exploring which movies have been viewed

Starting with MovieLense, we can easily extract quick results using methods such as the followingones:

* colCounts: This is the number of non-missing values for each column
* colMeans: This is the average value for each column

For instance, which are the most viewed movies? We can use colCounts for this purpose. First, let's count the views for each movie:

views\_per\_movie <- colCounts(MovieLense)

Then, we can sort the movies by number of views:

table\_views <- data.frame(

movie = names(views\_per\_movie),

views = views\_per\_movie

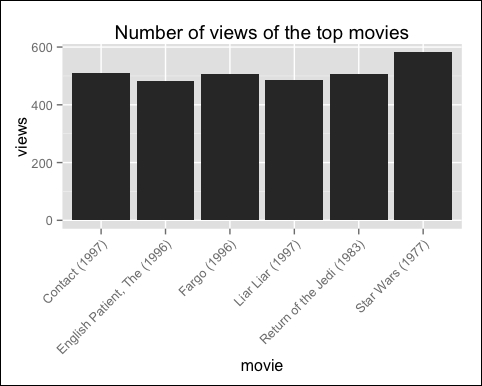
)

table\_views <- table\_views[order(table\_views$views, decreasing = TRUE), ]

Now, we can visualize the first six rows and build a histogram:

ggplot(table\_views[1:6, ], aes(x = movie, y = views)) + geom\_bar(stat="identity") + theme(axis.text.x = element\_text(angle = 45, hjust = 1)) + ggtitle("Number of views of the top movies")

The following image shows the number of views of the top movies:



In the preceding chart, you can notice that **Star Wars (1977)** is the most viewed movie, exceeding the others by about 100 views.

## Exploring the average ratings

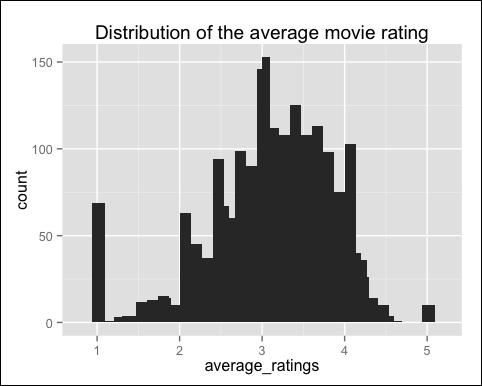
We can identify the top-rated movies by computing the average rating of each of them. For this purpose, we can use colMeans; it automatically ignores the 0s, since they represent missing values. Let's take a look at the distribution of the average movie rating:

average\_ratings <- colMeans(MovieLense)

Let's build the chart using qplot:

qplot(average\_ratings) + stat\_bin(binwidth = 0.1) + ggtitle("Distribution of the average movie rating")

The following image shows the distribution of the average movie rating:



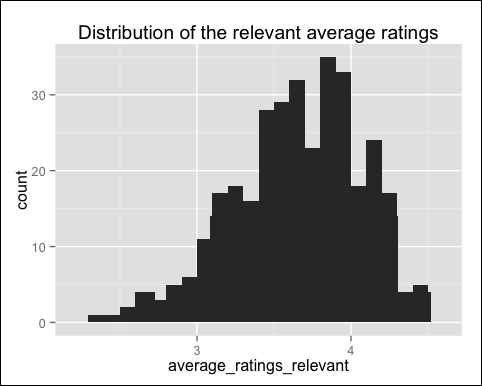
The highest value is around 3, and there are a few movies whose rating is either 1 or 5. Probably, thereason is that these movies received a rating from a few people only, so we shouldn't take them into account. We can remove the movies whose number of views is below a defined threshold, for instance, below 100:

average\_ratings\_relevant <- average\_ratings[views\_per\_movie > 100]

Let's build the chart:

qplot(average\_ratings\_relevant) + stat\_bin(binwidth = 0.1) + ggtitle(paste("Distribution of the relevant average ratings"))

The following image shows the distribution of the relevant average ratings:



All the rankings are between 2.3 and 4.5. As expected, we removed the extremes. The highest value changes, and now, it is around 4.

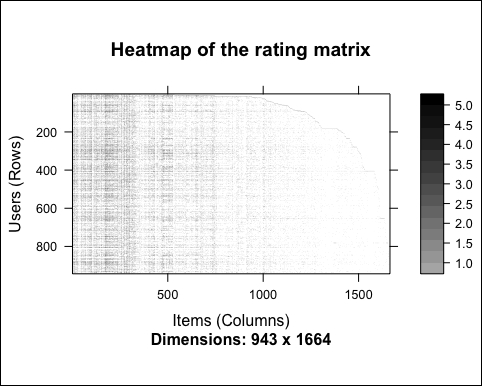
## Visualizing the matrix

We can visualize the matrix by building a heat map whose colors represent the ratings. Each row of the matrix corresponds to a user, each column to a movie, and each cell to its rating. For this purpose, we can use the generic method: image. The recommenderlab package redefined the method image for realRatingMatrix objects.

Let's build the heatmap using image:

image(MovieLense, main = "Heatmap of the rating matrix")

The following image displays the heatmap of the rating matrix:



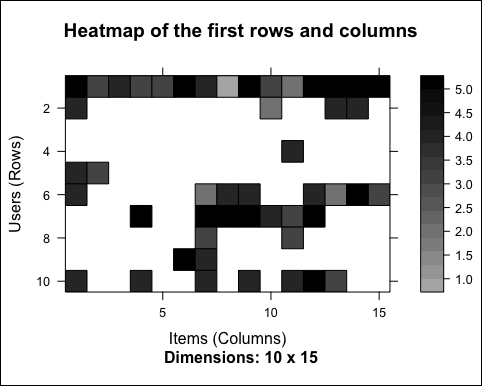
We can notice a white area in the top-right region. The reason is that the row and columns are sorted.

Since there are too many users and items, this chart is hard to read. We can build another chart zooming in on the first rows and columns.

Let's build the heat map using image:

image(MovieLense[1:10, 1:15], main = "Heatmap of the first rows and columns")

The following image shows the heatmap of the first rows and columns:



Some users saw more movies than the others. However, this chart is just displaying some random users and items. What if, instead, we select the most relevant users and items? This means visualizing only the users who have seen many movies and the movies that have been seen by many users. To identify and select the most relevant users and movies, follow these steps:

1. Determine the minimum number of movies per user.
2. Determine the minimum number of users per movie.
3. Select the users and movies matching these criteria.

For instance, we can visualize the top percentile of users and movies. In order to do this, we use the quantile function:

min\_n\_movies <- quantile(rowCounts(MovieLense), 0.99)

min\_n\_users <- quantile(colCounts(MovieLense), 0.99)

min\_n\_movies

## 99%

## 440.96

min\_n\_users

## 99%

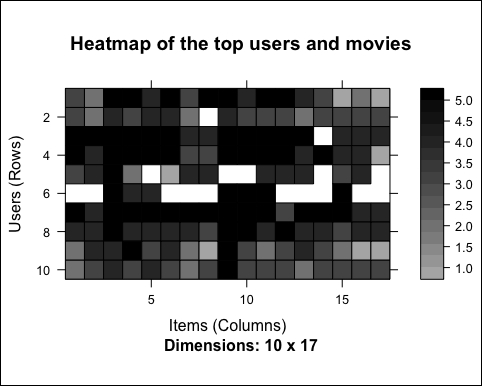
## 371.07

Now, we can visualize the rows and columns matching the criteria.

Let's build the heat map using image:

image(MovieLense[rowCounts(MovieLense) > min\_n\_movies, colCounts(MovieLense) > min\_n\_users], main = "Heatmap of the top users and movies")

The following image displays the heatmap of the top users and movies:



Let's take account of the users having watched more movies. Most of them have seen all the top movies, and this is not surprising. We can notice some columns that are darker than the others. These columns represent the highest-rated movies. Conversely, darker rows represent users giving higher ratings. Because of this, we might need to normalize the data.

In this section, we have explored the data. In the next section, we will process and transform it in order to have the inputs for the recommendation models.

# Data preparation

This section will show you how to prepare the data to be used in recommender models. Follow these steps:

1. Select the relevant data.
2. Normalize the data.

## Selecting the most relevant data

When we explored the data, we noticed that the table contains:

* Movies that have been viewed only a few times. Their ratings might be biased because of lack of data.
* Users who rated only a few movies. Their ratings might be biased.

We need to determine the minimum number of users per movie and vice versa. The correct solution comes from an iteration of the entire process of preparing the data, building a recommendation model, and validating it. Since we are implementing the model for the first time, we can use a rule of thumb. After having built the models, we can come back and modify the data preparation.

We will define ratings\_movies containing the matrix that we will use. It takes account of:

* Users who have rated at least 50 movies
* Movies that have been watched at least 100 times

The preceding points are defined in the following code:

ratings\_movies <- MovieLense[rowCounts(MovieLense) > 50, colCounts(MovieLense) > 100] ratings\_movies

## 560 x 332 rating matrix of class 'realRatingMatrix' with 55298 ratings.

The ratings\_movies object contains about half of the users and a fifth of the movies in comparison with MovieLense.

## Exploring the most relevant data

Using the same approach as we did in the previous section, let's visualize the top 2 percent of users and movies in the new matrix:

# visualize the top matrix

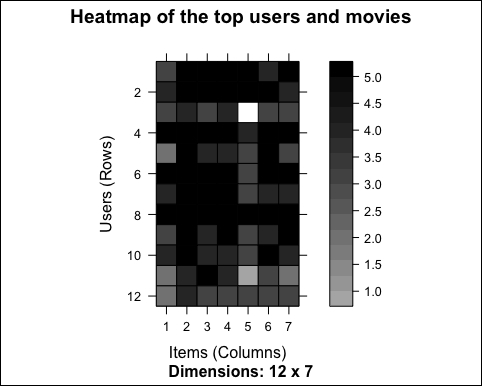
min\_movies <- quantile(rowCounts(ratings\_movies), 0.98)

min\_users <- quantile(colCounts(ratings\_movies), 0.98)

Let's build the heatmap:

image(ratings\_movies[rowCounts(ratings\_movies) > min\_movies, colCounts(ratings\_movies) > min\_users], main = "Heatmap of the top users and movies")

The following image displays the heatmap of the top users and movies:



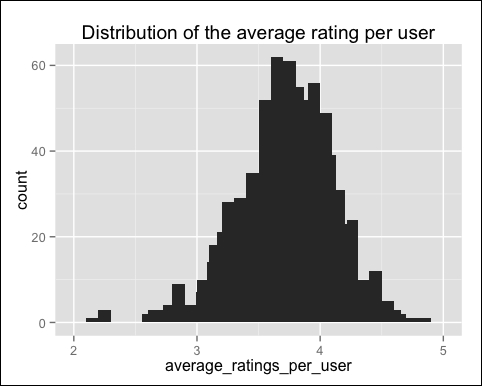
As we already noticed, some rows are darker than the others. This might mean that some users give higher ratings to all the movies. However, we have visualized the top movies only. In order to have an overview of all the users, let's take a look at the distribution of the average rating by user:

average\_ratings\_per\_user <- rowMeans(ratings\_movies)

Let's visualize the distribution:

qplot(average\_ratings\_per\_user) + stat\_bin(binwidth = 0.1) + ggtitle("Distribution of the average rating per user")

The following image shows the distribution of the average rating per user:



As suspected, the average rating varies a lot across different users.

## Normalizing the data

Having users who give high (or low) ratings to all their movies might bias the results. We can remove this effect by normalizing the data in such a way that the average rating of each user is 0. The prebuilt normalize function does it automatically:

ratings\_movies\_norm <- normalize(ratings\_movies)

Let's take a look at the average rating by users:

sum(rowMeans(ratings\_movies\_norm) > 0.00001)

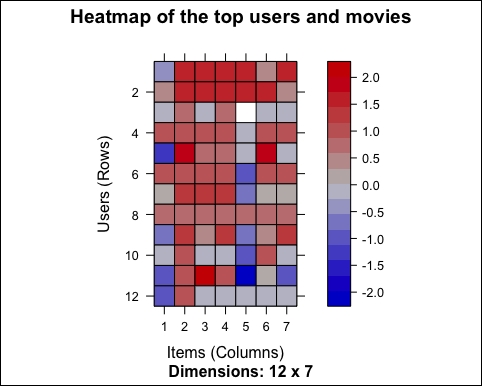
## [1] 0

As expected, the mean rating of each user is 0 (apart from the approximation error). We can visualize the new matrix using image. Let's build the heat map:

# visualize the normalized matrix

image(ratings\_movies\_norm[rowCounts(ratings\_movies\_norm) > min\_movies, colCounts(ratings\_movies\_norm) > min\_users], main = "Heatmap of the top users and movies")

The following image shows the heatmap of the top users and movies:



The first difference that we can notice is the colors, and this is because the data is continuous. Previously, the rating was an integer between 1 and 5. After the normalization, the rating can be any number between -5 and 5.

There are still some lines that are more blue and some that are more red. The reason is that we arevisualizing only the top movies. We already checked that the average rating is 0 for each user.

## Binarizing the data

Some recommendation models work on binary data, so we might want to binarize our data, that is, define a table containing only 0s and 1s. The 0s will be either treated as missing values or as bad ratings.

In our case, we can either:

* Define a matrix having 1 if the user rated the movie, and 0 otherwise. In this case, we are losing the information about the rating.
* Define a matrix having 1 if the rating is above or equal to a definite threshold (for example, 3), and 0 otherwise. In this case, giving a bad rating to a movie is equivalent to not having rated it.

Depending on the context, one choice is more appropriate than the other.

The function to binarize the data is binarize. Let's apply it to our data. First, let's define a matrix equal to 1 if the movie has been watched, that is if its rating is at least 1:

ratings\_movies\_watched <- binarize(ratings\_movies, minRating = 1)

Let's take a look at the results. In this case, we will have black-and-white charts so that we can visualize a larger portion of users and movies, for example, 5 percent. Similarly, let's select this 5 percent using quantile. The row and column counts are the same as the original matrix, so we can still apply rowCounts and colCounts on ratings\_movies:

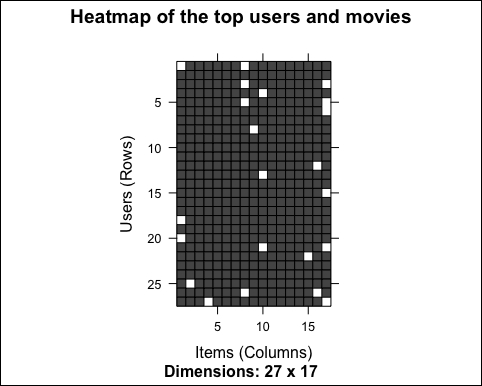
min\_movies\_binary <- quantile(rowCounts(ratings\_movies), 0.95)

min\_users\_binary <- quantile(colCounts(ratings\_movies), 0.95)

Let's build the heat map:

image(ratings\_movies\_watched[rowCounts(ratings\_movies) > min\_movies\_binary,colCounts(ratings\_movies) > min\_users\_binary], main = "Heatmap of the top users and movies")

The following image shows the heat map of the top users and movies:



Only a few cells contain unwatched movies. This is just because we selected the top users and movies.

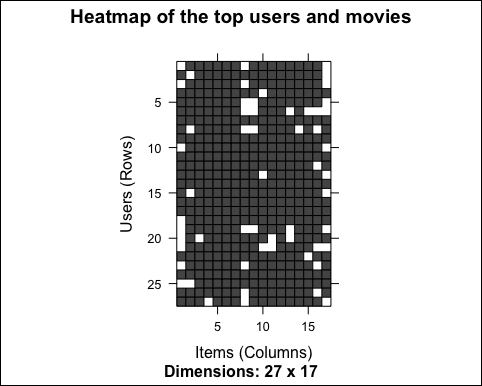
Let's use the same approach to compute and visualize the other binary matrix The cells having a rating above the threshold will have their value equal to 1 and the other cells will be 0s:

ratings\_movies\_good <- binarize(ratings\_movies, minRating = 3)

Let's build the heat map:

image(ratings\_movies\_good[rowCounts(ratings\_movies) > min\_movies\_binary, colCounts(ratings\_movies) > min\_users\_binary], main = "Heatmap of the top users and movies")

The following image shows the heatmap of the top users and movies:



As expected, we have more white cells now. Depending on the model, we can leave the ratings matrix as it is or normalize/binarize it.

In this section, we prepared the data to perform recommendations. In the upcoming sections, we will build collaborative filtering models.

# Item-based collaborative filtering

Collaborative filtering is a branch of recommendation that takes account of the information about different users. The word "collaborative" refers to the fact that users collaborate with each other to recommend items. In fact, the algorithms take account of user purchases and preferences. The starting point is a rating matrix in which rows correspond to users and columns correspond to items.

This section will show you an example of item-based collaborative filtering. Given a new user, the algorithm considers the user's purchases and recommends similar items. The core algorithm is based on these steps:

1. For each two items, measure how similar they are in terms of having received similar ratings by similar users
2. For each item, identify the k-most similar items
3. For each user, identify the items that are most similar to the user's purchases

In this chapter, we will see the overall approach to building an IBCF model. In addition, the upcoming sections will show its details.

## Defining the training and test sets

We will build the model using a part of the MovieLense dataset (the training set) and apply it on the other part (the test set). Since it's not a topic of this chapter, we will not evaluate the model, but will only recommend movies to the users in the test set.

The two sets are as follows:

* **Training set**: This set includes users from which the model learns
* **Test set**: This set includes users to whom we recommend movies

The algorithm automatically normalizes the data, so we can use ratings\_movies that contains relevant users and movies of MovieLense. We defined ratings\_movies in the previous section as the subset of MovieLense users who have rated at least 50 movies and movies that have been rated at least 100 times.

First, we randomly define the which\_train vector that is TRUE for users in the training set and FALSE for the others. We will set the probability in the training set as 80 percent:

which\_train <- sample(x = c(TRUE, FALSE), size = nrow(ratings\_movies), replace = TRUE, prob = c(0.8, 0.2))

head(which\_train)

## [1] TRUE TRUE TRUE FALSE TRUE FALSE

Let's define the training and the test sets:

recc\_data\_train <- ratings\_movies[which\_train, ]

recc\_data\_test <- ratings\_movies[!which\_train, ]

If we want to recommend items to each user, we could just use the k-fold:

* Split the users randomly into five groups
* Use a group as a test set and the other groups as training sets
* Repeat it for each group

This is a sample code:

which\_set <- sample(x = 1:5, size = nrow(ratings\_movies), replace = TRUE)

for(i\_model in 1:5) {

which\_train <- which\_set == i\_model

recc\_data\_train <- ratings\_movies[which\_train, ]

recc\_data\_test <- ratings\_movies[!which\_train, ]

# build the recommender

}

In order to show how this package works, we split the data into training and test sets manually. You can also do this automatically in recommenderlab using the evaluationScheme function. This function also contains some tools to evaluate models that we will use in the [Chapter 4](https://www.safaribooksonline.com/library/view/building-a-recommendation/9781783554492/ch04.html), Evaluating the Recommender Systems, which is about model evaluation.

Now, we have the inputs to build and validate the model.

## Building the recommendation model

The function to build models is recommender and its inputs are as follows:

* **Data**: This is the training set
* **Method**: This is the name of the technique
* **Parameters**: These are some optional parameters of the technique

The model is called IBCF, which stands for item-based collaborative filtering. Let's take a look at its parameters:

recommender\_models <- recommenderRegistry$get\_entries(dataType = "realRatingMatrix")

recommender\_models$IBCF\_realRatingMatrix$parameters

| **Parameters** | **Default** |
| --- | --- |
| k | 30 |
| method | Cosine |
| normalize | center |
| normalize\_sim\_matrix | FALSE |
| alpha | 0.5 |
| na\_as\_zero | FALSE |
| minRating | NA |

Some relevant parameters are as follows:

* k: In the first step, the algorithm computes the similarities among each pair of items. Then, for each item, it identifies its k most similar items and stores it.
* method: This is the similarity function. By default, it is Cosine. Another popular option is pearson.

At the moment, we can just set them to their defaults. In order to show how to change parameters, we are setting k = 30, which is the default. We are now ready to build a recommender model:

recc\_model <- Recommender(data = recc\_data\_train, method = "IBCF", parameter = list(k = 30))recc\_model

## Recommender of type 'IBCF' for 'realRatingMatrix' ## learned using 111 users.

class(recc\_model)

## [1] "Recommender"

## attr(,"package")

## [1] "recommenderlab"

The recc\_model class is an object of the Recommender class containing the model.

## Exploring the recommender model

Using getModel, we can extract some details about the model, such as its description and parameters:

model\_details <- getModel(recc\_model)

model\_details$description

## [1] "IBCF: Reduced similarity matrix"

model\_details$k

## [1] 30

The model\_details$sim component contains the similarity matrix. Let's check its structure:

class(model\_details$sim)

## [1] "dgCMatrix"

## attr(,"package")

## [1] "Matrix"

dim(model\_details$sim)

## [1] 332 332

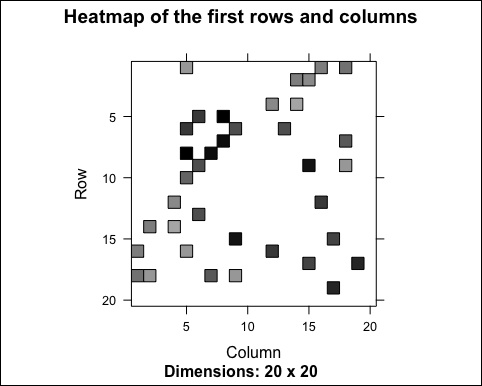
As expected, model\_details$sim is a square matrix whose size is equal to the number of items. We can explore a part of it using image:

n\_items\_top <- 20

Let's build the heat map:

image(model\_details$sim[1:n\_items\_top, 1:n\_items\_top], main = "Heatmap of the first rows and columns")

The following image displays heatmap of the first rows and columns:



Most of the values are equal to 0. The reason is that each row contains only k elements. Let's check it:

model\_details$k

## [1] 30

row\_sums <- rowSums(model\_details$sim > 0)

table(row\_sums)

## row\_sums

## 30

## 332

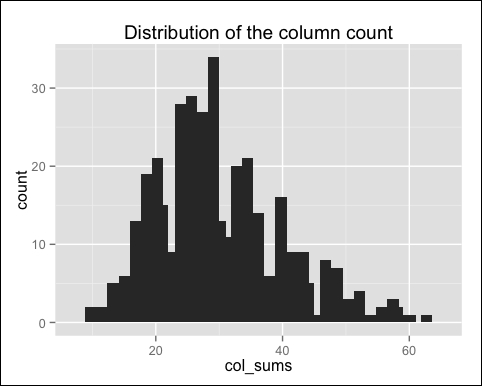
As expected, each row has 30 elements greater than 0. However, the matrix is not supposed to be symmetric. In fact, the number of non-null elements for each column depends on how many times the corresponding movie was included in the top k of another movie. Let's check the distribution of the number of elements by column:

col\_sums <- colSums(model\_details$sim > 0)

Let's build the distribution chart:

qplot(col\_sums) + stat\_bin(binwidth = 1) + ggtitle("Distribution of the column count")

The following image displays the distribution of the column count:



As expected, there are a few movies that are similar to many others. Let's see which are the movies with the most elements:

which\_max <- order(col\_sums, decreasing = TRUE)[1:6]

rownames(model\_details$sim)[which\_max]

| **Movie** | **col\_sum** |
| --- | --- |
| Sling Blade (1996) | 62 |
| Usual Suspects, The (1995) | 60 |
| Fargo (1996) | 58 |
| Vertigo (1958) | 58 |
| Stargate (1994) | 57 |
| The Godfather (1972) | 55 |

## Applying the recommender model on the test set

Now, we are able to recommend movies to the users in the test set. We will define n\_recommended that specifies the number of items to recommend to each user. This section will show you the most popular approach to computing a weighted sum:

n\_recommended <- 6

For each user, the algorithm extracts its rated movies. For each movie, it identifies all its similar items, starting from the similarity matrix. Then, the algorithm ranks each similar item in this way:

* Extract the user rating of each purchase associated with this item. The rating is used as a weight.
* Extract the similarity of the item with each purchase associated with this item.
* Multiply each weight with the related similarity.
* Sum everything up.

Then, the algorithm identifies the top n recommendations:

recc\_predicted <- predict(object = recc\_model, newdata = recc\_data\_test, n = n\_recommended)

recc\_predicted

## Recommendations as 'topNList' with n = 6 for 449 users.

The recc\_predicted object contains the recommendations. Let's take a look at its structure:

class(recc\_predicted)

## [1] "topNList"## attr(,"package")## [1] "recommenderlab"

slotNames(recc\_predicted)

## [1] "items" "itemLabels" "n"

The slots are:

* items: This is the list with the indices of the recommended items for each user
* itemLabels: This is the name of the items
* n: This is the number of recommendations

For instance, these are the recommendations for the first user:

recc\_predicted@items[[1]]

## [1] 201 182 254 274 193 297

We can extract the recommended movies from recc\_predicted@item labels:

recc\_user\_1 <- recc\_predicted@items[[1]]movies\_user\_1 <- recc\_predicted@itemLabels[recc\_user\_1]

movies\_user\_1

| **Index** | **Movie** |
| --- | --- |
| 201 | Schindler's List (1993) |
| 182 | Secrets and Lies (1996) |
| 254 | Trainspotting (1996) |
| 274 | The Deer Hunter (1978) |
| 193 | L.A. Confidential (1997) |
| 297 | The Manchurian Candidate (1962) |

We can define a matrix with the recommendations for each user:

recc\_matrix <- sapply(recc\_predicted@items, function(x){

colnames(ratings\_movies)[x]

})

dim(recc\_matrix)

## [1] 6 449

Let's visualize the recommendations for the first four users:

recc\_matrix[, 1:4]

|  |  |
| --- | --- |
| Schindler's List (1993) | Babe (1995) |
| Secrets and Lies (1996) | The Usual Suspects (1995) |
| Trainspotting (1996) | Taxi Driver (1976) |
| The Deer Hunter (1978) | Blade Runner (1982) |
| L.A. Confidential (1997) | Welcome to the Dollhouse (1995) |
| Manchurian Candidate, The (1962) | The Silence of the Lambs (1991) |
| Batman Forever (1995) | Strictly Ballroom (1992) |
| Stargate (1994) | L.A. Confidential (1997) |
| Star Trek III: The Search for Spock (1984) | Cold Comfort Farm (1995) |
| Tin Cup (1996) | 12 Angry Men (1957) |
| Courage Under Fire (1996) | Vertigo (1958) |
| Dumbo (1941) | A Room with a View (1986) |

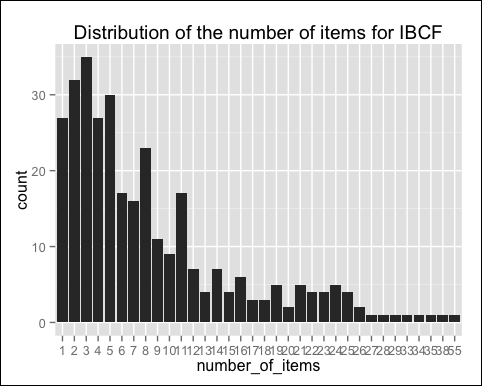
Now, we can identify the most recommended movies. For this purpose, we will define a vector with all the recommendations, and we will build a frequency plot:

number\_of\_items <- factor(table(recc\_matrix))chart\_title <- "Distribution of the number of items for IBCF"

Let's build the distribution chart:

qplot(number\_of\_items) + ggtitle(chart\_title)

The following image shows the distribution of the number of items for IBCF:



Most of the movies have been recommended only a few times, and a few movies have been recommended many times. Let's see which are the most popular movies:

number\_of\_items\_sorted <- sort(number\_of\_items, decreasing = TRUE)

number\_of\_items\_top <- head(number\_of\_items\_sorted, n = 4)

table\_top <- data.frame(names(number\_of\_items\_top), number\_of\_items\_top)

table\_top

|  | **names.number\_of\_items\_top** |
| --- | --- |
| Mr. Smith Goes to Washington (1939) | Mr. Smith Goes to Washington (1939) |
| Babe (1995) | Babe (1995) |
| The Maltese Falcon (1941) | The Maltese Falcon (1941) |
| L.A. Confidential (1997) | L.A. Confidential (1997) |

The preceding table continues as follows:

|  | **number\_of\_items\_top** |
| --- | --- |
| Mr. Smith Goes to Washington (1939) | 55 |
| Babe (1995) | 38 |
| The Maltese Falcon (1941) | 35 |
| L.A. Confidential (1997) | 34 |

IBCF recommends items on the basis of the similarity matrix. It's an eager-learning model, that is, once it's built, it doesn't need to access the initial data. For each item, the model stores the k-most similar, so the amount of information is small once the model is built. This is an advantage in the presence of lots of data.

In addition, this algorithm is efficient and scalable, so it works well with big rating matrices. Its accuracy is rather good, compared with other recommendation models.

In the next section, we will explore another branch of techniques: user-based collaborative filtering.

# Item-based collaborative filtering

Collaborative filtering is a branch of recommendation that takes account of the information about different users. The word "collaborative" refers to the fact that users collaborate with each other to recommend items. In fact, the algorithms take account of user purchases and preferences. The starting point is a rating matrix in which rows correspond to users and columns correspond to items.

This section will show you an example of item-based collaborative filtering. Given a new user, the algorithm considers the user's purchases and recommends similar items. The core algorithm is based on these steps:

1. For each two items, measure how similar they are in terms of having received similar ratings by similar users
2. For each item, identify the k-most similar items
3. For each user, identify the items that are most similar to the user's purchases

In this chapter, we will see the overall approach to building an IBCF model. In addition, the upcoming sections will show its details.

## Defining the training and test sets

We will build the model using a part of the MovieLense dataset (the training set) and apply it on the other part (the test set). Since it's not a topic of this chapter, we will not evaluate the model, but will only recommend movies to the users in the test set.

The two sets are as follows:

* **Training set**: This set includes users from which the model learns
* **Test set**: This set includes users to whom we recommend movies

The algorithm automatically normalizes the data, so we can use ratings\_movies that contains relevant users and movies of MovieLense. We defined ratings\_movies in the previous section as the subset of MovieLense users who have rated at least 50 movies and movies that have been rated at least 100 times.

First, we randomly define the which\_train vector that is TRUE for users in the training set and FALSE for the others. We will set the probability in the training set as 80 percent:

which\_train <- sample(x = c(TRUE, FALSE), size = nrow(ratings\_movies), replace = TRUE, prob = c(0.8, 0.2))

head(which\_train)

## [1] TRUE TRUE TRUE FALSE TRUE FALSE

Let's define the training and the test sets:

recc\_data\_train <- ratings\_movies[which\_train, ]

recc\_data\_test <- ratings\_movies[!which\_train, ]

If we want to recommend items to each user, we could just use the k-fold:

* Split the users randomly into five groups
* Use a group as a test set and the other groups as training sets
* Repeat it for each group

This is a sample code:

which\_set <- sample(x = 1:5, size = nrow(ratings\_movies), replace = TRUE)

for(i\_model in 1:5) {

which\_train <- which\_set == i\_model

recc\_data\_train <- ratings\_movies[which\_train, ]

recc\_data\_test <- ratings\_movies[!which\_train, ]

# build the recommender

}

In order to show how this package works, we split the data into training and test sets manually. You can also do this automatically in recommenderlab using the evaluationScheme function. This function also contains some tools to evaluate models that we will use in the [Chapter 4](https://www.safaribooksonline.com/library/view/building-a-recommendation/9781783554492/ch04.html), Evaluating the Recommender Systems, which is about model evaluation.

Now, we have the inputs to build and validate the model.

## Building the recommendation model

The function to build models is recommender and its inputs are as follows:

* **Data**: This is the training set
* **Method**: This is the name of the technique
* **Parameters**: These are some optional parameters of the technique

The model is called IBCF, which stands for item-based collaborative filtering. Let's take a look at its parameters:

recommender\_models <- recommenderRegistry$get\_entries(dataType = "realRatingMatrix")

recommender\_models$IBCF\_realRatingMatrix$parameters

| **Parameters** | **Default** |
| --- | --- |
| k | 30 |
| method | Cosine |
| normalize | center |
| normalize\_sim\_matrix | FALSE |
| alpha | 0.5 |
| na\_as\_zero | FALSE |
| minRating | NA |

Some relevant parameters are as follows:

* k: In the first step, the algorithm computes the similarities among each pair of items. Then, for each item, it identifies its k most similar items and stores it.
* method: This is the similarity function. By default, it is Cosine. Another popular option is pearson.

At the moment, we can just set them to their defaults. In order to show how to change parameters, we are setting k = 30, which is the default. We are now ready to build a recommender model:

recc\_model <- Recommender(data = recc\_data\_train, method = "IBCF", parameter = list(k = 30))recc\_model

## Recommender of type 'IBCF' for 'realRatingMatrix' ## learned using 111 users.

class(recc\_model)

## [1] "Recommender"

## attr(,"package")

## [1] "recommenderlab"

The recc\_model class is an object of the Recommender class containing the model.

## Exploring the recommender model

Using getModel, we can extract some details about the model, such as its description and parameters:

model\_details <- getModel(recc\_model)

model\_details$description

## [1] "IBCF: Reduced similarity matrix"

model\_details$k

## [1] 30

The model\_details$sim component contains the similarity matrix. Let's check its structure:

class(model\_details$sim)

## [1] "dgCMatrix"

## attr(,"package")

## [1] "Matrix"

dim(model\_details$sim)

## [1] 332 332

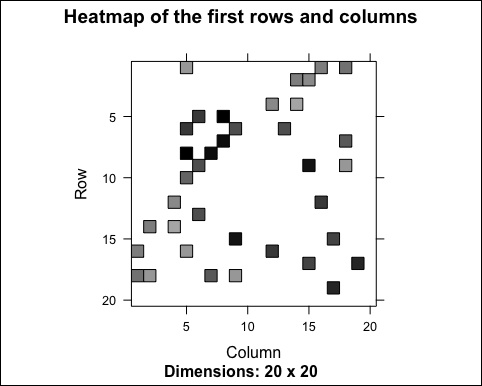
As expected, model\_details$sim is a square matrix whose size is equal to the number of items. We can explore a part of it using image:

n\_items\_top <- 20

Let's build the heat map:

image(model\_details$sim[1:n\_items\_top, 1:n\_items\_top], main = "Heatmap of the first rows and columns")

The following image displays heatmap of the first rows and columns:



Most of the values are equal to 0. The reason is that each row contains only k elements. Let's check it:

model\_details$k

## [1] 30

row\_sums <- rowSums(model\_details$sim > 0)

table(row\_sums)

## row\_sums

## 30

## 332

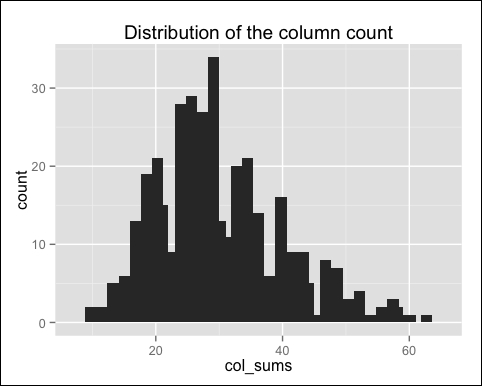
As expected, each row has 30 elements greater than 0. However, the matrix is not supposed to be symmetric. In fact, the number of non-null elements for each column depends on how many times the corresponding movie was included in the top k of another movie. Let's check the distribution of the number of elements by column:

col\_sums <- colSums(model\_details$sim > 0)

Let's build the distribution chart:

qplot(col\_sums) + stat\_bin(binwidth = 1) + ggtitle("Distribution of the column count")

The following image displays the distribution of the column count:



As expected, there are a few movies that are similar to many others. Let's see which are the movies with the most elements:

which\_max <- order(col\_sums, decreasing = TRUE)[1:6]

rownames(model\_details$sim)[which\_max]

| **Movie** | **col\_sum** |
| --- | --- |
| Sling Blade (1996) | 62 |
| Usual Suspects, The (1995) | 60 |
| Fargo (1996) | 58 |
| Vertigo (1958) | 58 |
| Stargate (1994) | 57 |
| The Godfather (1972) | 55 |

## Applying the recommender model on the test set

Now, we are able to recommend movies to the users in the test set. We will define n\_recommendedthat specifies the number of items to recommend to each user. This section will show you the most popular approach to computing a weighted sum:

n\_recommended <- 6

For each user, the algorithm extracts its rated movies. For each movie, it identifies all its similar items, starting from the similarity matrix. Then, the algorithm ranks each similar item in this way:

* Extract the user rating of each purchase associated with this item. The rating is used as a weight.
* Extract the similarity of the item with each purchase associated with this item.
* Multiply each weight with the related similarity.
* Sum everything up.

Then, the algorithm identifies the top n recommendations:

recc\_predicted <- predict(object = recc\_model, newdata = recc\_data\_test, n = n\_recommended)

recc\_predicted

## Recommendations as 'topNList' with n = 6 for 449 users.

The recc\_predicted object contains the recommendations. Let's take a look at its structure:

class(recc\_predicted)

## [1] "topNList"## attr(,"package")## [1] "recommenderlab"

slotNames(recc\_predicted)

## [1] "items" "itemLabels" "n"

The slots are:

* items: This is the list with the indices of the recommended items for each user
* itemLabels: This is the name of the items
* n: This is the number of recommendations

For instance, these are the recommendations for the first user:

recc\_predicted@items[[1]]

## [1] 201 182 254 274 193 297

We can extract the recommended movies from recc\_predicted@item labels:

recc\_user\_1 <- recc\_predicted@items[[1]]movies\_user\_1 <- recc\_predicted@itemLabels[recc\_user\_1]

movies\_user\_1

| **Index** | **Movie** |
| --- | --- |
| 201 | Schindler's List (1993) |
| 182 | Secrets and Lies (1996) |
| 254 | Trainspotting (1996) |
| 274 | The Deer Hunter (1978) |
| 193 | L.A. Confidential (1997) |
| 297 | The Manchurian Candidate (1962) |

We can define a matrix with the recommendations for each user:

recc\_matrix <- sapply(recc\_predicted@items, function(x){

colnames(ratings\_movies)[x]

})

dim(recc\_matrix)

## [1] 6 449

Let's visualize the recommendations for the first four users:

recc\_matrix[, 1:4]

|  |  |
| --- | --- |
| Schindler's List (1993) | Babe (1995) |
| Secrets and Lies (1996) | The Usual Suspects (1995) |
| Trainspotting (1996) | Taxi Driver (1976) |
| The Deer Hunter (1978) | Blade Runner (1982) |
| L.A. Confidential (1997) | Welcome to the Dollhouse (1995) |
| Manchurian Candidate, The (1962) | The Silence of the Lambs (1991) |
| Batman Forever (1995) | Strictly Ballroom (1992) |
| Stargate (1994) | L.A. Confidential (1997) |
| Star Trek III: The Search for Spock (1984) | Cold Comfort Farm (1995) |
| Tin Cup (1996) | 12 Angry Men (1957) |
| Courage Under Fire (1996) | Vertigo (1958) |
| Dumbo (1941) | A Room with a View (1986) |

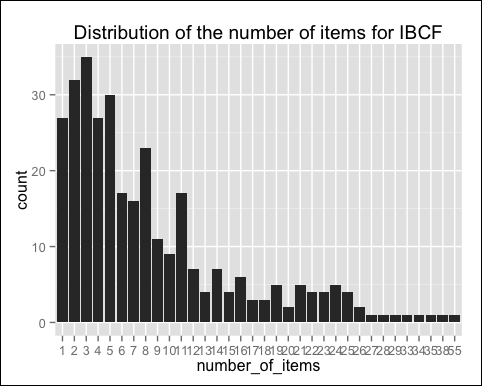
Now, we can identify the most recommended movies. For this purpose, we will define a vector with all the recommendations, and we will build a frequency plot:

number\_of\_items <- factor(table(recc\_matrix))chart\_title <- "Distribution of the number of items for IBCF"

Let's build the distribution chart:

qplot(number\_of\_items) + ggtitle(chart\_title)

The following image shows the distribution of the number of items for IBCF:



Most of the movies have been recommended only a few times, and a few movies have been recommended many times. Let's see which are the most popular movies:

number\_of\_items\_sorted <- sort(number\_of\_items, decreasing = TRUE)

number\_of\_items\_top <- head(number\_of\_items\_sorted, n = 4)

table\_top <- data.frame(names(number\_of\_items\_top), number\_of\_items\_top)

table\_top

|  | **names.number\_of\_items\_top** |
| --- | --- |
| Mr. Smith Goes to Washington (1939) | Mr. Smith Goes to Washington (1939) |
| Babe (1995) | Babe (1995) |
| The Maltese Falcon (1941) | The Maltese Falcon (1941) |
| L.A. Confidential (1997) | L.A. Confidential (1997) |

The preceding table continues as follows:

|  | **number\_of\_items\_top** |
| --- | --- |
| Mr. Smith Goes to Washington (1939) | 55 |
| Babe (1995) | 38 |
| The Maltese Falcon (1941) | 35 |
| L.A. Confidential (1997) | 34 |

IBCF recommends items on the basis of the similarity matrix. It's an eager-learning model, that is, once it's built, it doesn't need to access the initial data. For each item, the model stores the k-most similar, so the amount of information is small once the model is built. This is an advantage in the presence of lots of data.

In addition, this algorithm is efficient and scalable, so it works well with big rating matrices. Its accuracy is rather good, compared with other recommendation models.

In the next section, we will explore another branch of techniques: user-based collaborative filtering.