6
Understanding Big Data Analysis with Machine Learning

In this chapter, we are going to learn about different machine learning techniques that can be used with R and Hadoop to perform Big Data analytics with the help of the following points:

- Introduction to machine learning
- Types of machine-learning algorithms
- Supervised machine-learning algorithms
- Unsupervised machine-learning algorithms
- Recommendation algorithms

Introduction to machine learning

Machine learning is a branch of artificial intelligence that allows us to make our application intelligent without being explicitly programmed. Machine learning concepts are used to enable applications to take a decision from the available datasets. A combination of machine learning and data mining can be used to develop spam mail detectors, self-driven cars, speech recognition, face recognition, and online transactional fraud-activity detection.

There are many popular organizations that are using machine-learning algorithms to make their service or product understand the need of their users and provide services as per their behavior. Google has its intelligent web search engine, which provides a number one search, spam classification in Google Mail, news labeling in Google News, and Amazon for recommender systems. There are many open source frameworks available for developing these types of applications/frameworks, such as R, Python, Apache Mahout, and Weka.

Types of machine-learning algorithms

There are three different types of machine-learning algorithms for intelligent system development:

- Supervised machine-learning algorithms
- Unsupervised machine-learning algorithms
- Recommender systems

In this chapter, we are going to discuss well-known business problems with classification, regression, and clustering, as well as how to perform these machine-learning techniques over Hadoop to overcome memory issues.

If you load a dataset that won't be able to fit into your machine memories and you try to run it, the predictive analysis will throw an error related to machine memory, such as **Error: cannot allocate vector of size 990.1 MB**. The solution is to increase the machine configuration or parallelize with commodity hardware.

Supervised machine-learning algorithms

In this section, we will be learning about supervised machine-learning algorithms. The algorithms are as follows:

- Linear regression
- Logistic regression

Linear regression

Linear regression is mainly used for predicting and forecasting values based on historical information. Regression is a supervised machine-learning technique to identify the linear relationship between target variables and explanatory variables. We can say it is used for predicting the target variable values in numeric form.

In the following section, we will be learning about linear regression with R and linear regression with R and Hadoop.

Here, the variables that are going to be predicted are considered as target variables and the variables that are going to help predict the target variables are called explanatory variables. With the linear relationship, we can identify the impact of a change in explanatory variables on the target variable.

In mathematics, regression can be formulated as follows:

$$y = ax + e$$

Other formulae include:

- The slope of the regression line is given by:

$$a = (N\Sigma xy - (\Sigma x)(\Sigma y)) / (N\Sigma x^2 - (\Sigma x)^2)$$

- The intercept point of regression is given by:

$$e = (\Sigma y - b(\Sigma x)) / N$$

Here, *x* and *y* are variables that form a dataset and *N* is the total numbers of values.

Suppose we have the data shown in the following table:

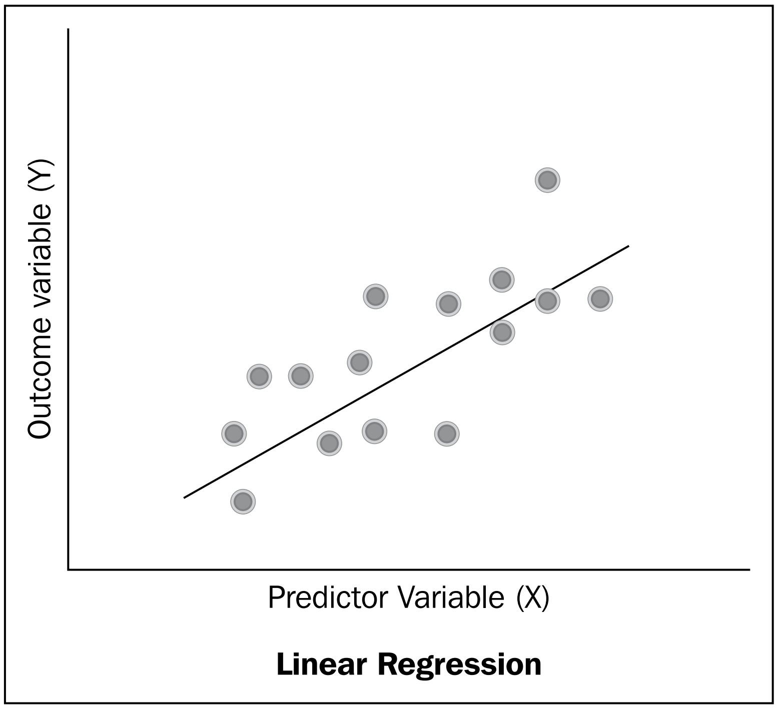
x	**y**
63	3.1
64	3.6
65	3.8
66	4

If we have a new value of *x*, we can get the value of *y* with it with the help of the regression formula.

Applications of linear regression include:

- Sales forecasting
- Predicting optimum product price
- Predicting the next online purchase from various sources and campaigns

Let's look at the statistical technique to implement the regression model for the provided dataset. Assume that we have been given n number of statistical data units.



Its formula is as follows:

$Y = e_0 + a_0x_0 + a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4$

Here, *Y* is the target variable (response variable), *xi* are explanatory variables, and e_0 is the sum of the squared error term, which can be considered as noise. To get a more accurate prediction, we need to reduce this error term as soon as possible with the help of the `call` function.

Linear regression with R

Now we will see how to perform linear regression in R. We can use the in-built `lm()` method to build a linear regression model with R.

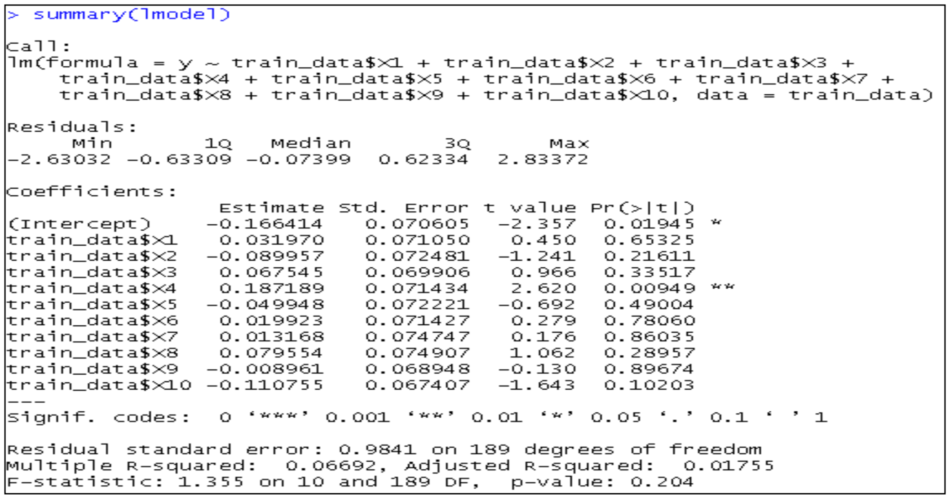
```
Model <-lm(target ~ ex_var1, data=train_dataset)
```

It will build a regression model based on the property of the provided dataset and store all of the variables' coefficients and model parameters used for predicting and identifying of data pattern from the model variable values.

```
# Defining data variables
X = matrix(rnorm(2000), ncol = 10)
y = as.matrix(rnorm(200))

# Bundling data variables into dataframe
train_data <- data.frame(X,y)

# Training model for generating prediction
lmodel<- lm(y~ train_data $X1 + train_data $X2 + train_data $X3 +
train_data $X4 + train_data $X5 + train_data $X6 + train_data $X7 +
train_data $X8 + train_data $X9 + train_data $X10,data= train_data)

summary(lmodel)
```

The following are the various model parameters that can be displayed with the preceding `summary` command:

- **RSS**: This is equal to ∑(yactual - y)^2.
- **Degrees of Freedom** (**DOF**): This is used for identifying the degree of fit for the prediction model, which should be as small as possible (logically, the value 0 means perfect prediction).
- **Residual standard error** (**RSS/DF**): This is used for identifying the goodness of fit for the prediction model, which should be as small as possible (logically, the value 0 means perfect prediction).
- **pr**: This is the probability for a variable to be included into the model; it should be less than 0.05 for a variable to be included.
- **t-value**: This is equal to 15.
- **f**: This is the statistic that checks whether R square is a value other than zero.



```
> summary(lmodel)

Call:
lm(formula = y ~ train_data$X1 + train_data$X2 + train_data$X3 +
    train_data$X4 + train_data$X5 + train_data$X6 + train_data$X7 +
    train_data$X8 + train_data$X9 + train_data$X10, data = train_data)

Residuals:
     Min       1Q   Median       3Q      Max
-2.63032 -0.63309 -0.07399  0.62334  2.83372

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)    -0.166414   0.070605  -2.357  0.01945 *
train_data$X1   0.031970   0.071050   0.450  0.65325
train_data$X2  -0.089957   0.072481  -1.241  0.21611
train_data$X3   0.067545   0.069906   0.966  0.33517
train_data$X4   0.187189   0.071434   2.620  0.00949 **
train_data$X5  -0.049948   0.072221  -0.692  0.49004
train_data$X6   0.019923   0.071427   0.279  0.78060
train_data$X7   0.013168   0.074747   0.176  0.86035
train_data$X8   0.079554   0.074907   1.062  0.28957
train_data$X9  -0.008961   0.068948  -0.130  0.89674
train_data$X10 -0.110755   0.067407  -1.643  0.10203
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.9841 on 189 degrees of freedom
Multiple R-squared:  0.06692,	Adjusted R-squared:  0.01755
F-statistic: 1.355 on 10 and 189 DF,  p-value: 0.204
```

Linear regression with R and Hadoop

Assume we have a large dataset. How will we perform regression data analysis now? In such cases, we can use R and Hadoop integration to perform parallel linear regression by implementing Mapper and Reducer. It will divide the dataset into chunks among the available nodes and then they will process the distributed data in parallel. It will not fire memory issues when we run with an R and Hadoop cluster because the large dataset is going to be distributed and processed with R among Hadoop computation nodes. Also, keep in mind that this implemented method does not provide higher prediction accuracy than the `lm()` model.

RHadoop is used here for integration of R and Hadoop, which is a trusted open source distribution of **Revolution Analytics**. For more information on RHadoop, visit `https://github.com/RevolutionAnalytics/RHadoop/wiki`. Among the packages of RHadoop, here we are using only the `rmr` and `rhdfs` libraries.

Let's see how to perform regression analysis with R and Hadoop data technologies.

```
# Defining the datasets with Big Data matrix X
X = matrix(rnorm(20000), ncol = 10)
X.index = to.dfs(cbind(1:nrow(X), X))
y = as.matrix(rnorm(2000))
```

Here, the `Sum()` function is re-usable as shown in the following code:

```
# Function defined to be used as reducers
Sum =
  function(., YY)
    keyval(1, list(Reduce('+', YY)))
```

The outline of the linear regression algorithm is as follows:

1. Calculating the `Xtx` value with MapReduce job1.
2. Calculating the `Xty` value with MapReduce job2.
3. Deriving the coefficient values with `Solve (Xtx, Xty)`.

Let's understand these steps one by one.

The first step is to calculate the `Xtx` value with MapReduce job 1.

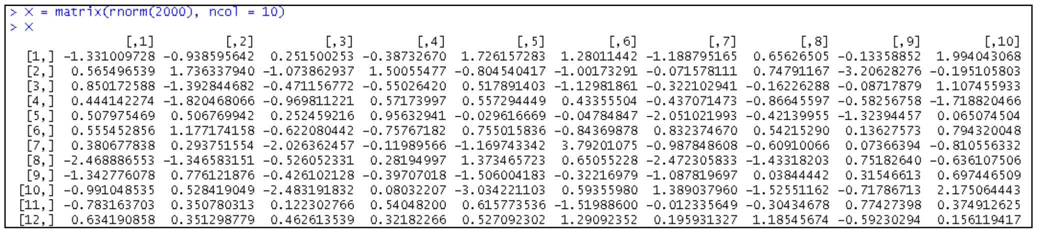
1. The big matrix is passed to the Mapper in chunks of complete rows. Smaller cross-products are computed for these submatrices and passed on to a single Reducer, which sums them together. Since we have a single key, a Combiner is mandatory and since the matrix sum is associative and commutative, we certainly can use it here.

```
# XtX =
  values(

# For loading hdfs data in to R
    from.dfs(

# MapReduce Job to produce XT*X
      mapreduce(
        input = X.index,

# Mapper – To calculate and emitting XT*X
        map =
          function(., Xi) {
            yi = y[Xi[,1],]
            Xi = Xi[,-1]
            keyval(1, list(t(Xi) %*% Xi))},

# Reducer – To reduce the Mapper output by performing sum
operation over them
        reduce = Sum,
        combine = TRUE)))[[1]]
```

2. When we have a large amount of data stored in **Hadoop Distributed File System** (**HDFS**), we need to pass its path value to the input parameters in the `MapReduce` method.
3. In the preceding code, we saw that `X` is the design matrix, which has been created with the following function:

```
X = matrix(rnorm(2000), ncol = 10)
```

4. Its output will look as shown in the following screenshot:



```
> X = matrix(rnorm(2000), ncol = 10)
> X
              [,1]         [,2]         [,3]        [,4]         [,5]        [,6]         [,7]        [,8]        [,9]        [,10]
 [1,] -1.331009728 -0.938595642  0.251500253 -0.38732670  1.726157283  1.28011442 -1.188795165  0.65626505 -0.13358852  1.994043068
 [2,]  0.565496539  1.736337940 -1.073862937  1.50055477 -0.804540417 -1.00173291 -0.071578111  0.74791167 -3.20628276 -0.195105803
 [3,]  0.850172588 -1.392844682 -0.471156772 -0.55026420  0.517891403 -1.12981861 -0.322102941 -0.16226288 -0.08717879  1.107455933
 [4,]  0.444142274 -1.820468066 -0.969811221  0.57173997  0.557294449  0.43355504 -0.437071473 -0.86645597 -0.58256758 -1.718820466
 [5,]  0.507975469  0.506769942  0.252459216  0.95632941 -0.029616669 -0.04784847 -2.051021993 -0.42139955 -1.32394457  0.065074504
 [6,]  0.555452856  1.177174158 -0.622080442 -0.75767182  0.755015836 -0.84369878  0.832374670  0.54215290  0.13627573  0.794320048
 [7,]  0.380677838  0.293751554 -2.026362457 -0.11989566 -1.169743342  3.79201075 -0.987848608 -0.60910066  0.07366394 -0.810556332
 [8,] -2.468886553 -1.346583151 -0.526052331  0.28194997  1.373465723  0.65055228 -2.472305833 -1.43318203  0.75182640 -0.636107506
 [9,] -1.342776078  0.776121876 -0.426102128 -0.39707018 -1.506004183 -0.32216979 -1.087819697  0.03844442  0.31546613  0.697446509
[10,] -0.991048535  0.528419049 -2.483191832  0.08032207 -3.034221103  0.59355980  1.389037960 -1.52551162 -0.71786713  2.175064443
[11,] -0.783163703  0.350780313  0.122302766  0.54048200  0.615773536 -1.51988600 -0.012335649 -0.30434678  0.77427398  0.374912625
[12,]  0.634190858  0.351298779  0.462613539  0.32182266  0.527092302  1.29092352  0.195931327  1.18545674 -0.59230294  0.156119417
```

So, here all the columns will be considered as explanatory variables and their standard errors can be calculated in a similar manner to how we calculated them with normal linear regression.

To calculate the `Xty` value with MapReduce job 2 is pretty much the same as for the vector `y`, which is available to the nodes according to normal scope rules.

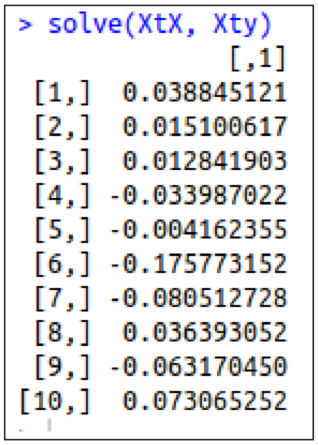
```
Xty = values(

# For loading hdfs data
from.dfs(

# MapReduce job to produce XT * y
      mapreduce(
       input = X.index,

# Mapper - To calculate and emitting XT*y
        map = function(., Xi) {
          yi = y[Xi[,1],]
          Xi = Xi[,-1]
          keyval(1, list(t(Xi) %*% yi))},

# Reducer - To reducer the Mapper output by performing # sum
operation over them
        reduce = Sum,
        combine = TRUE)))[[1]]
```

To derive the coefficient values with `solve (Xtx, Xty)`, use the following steps:

1. Finally, we just need to call the following line of code to get the coefficient values.

   ```
   solve(XtX, Xty)
   ```

2. The output of the preceding command will be as shown in the following screenshot:



```
> solve(XtX, Xty)
              [,1]
 [1,]  0.038845121
 [2,]  0.015100617
 [3,]  0.012841903
 [4,] -0.033987022
 [5,] -0.004162355
 [6,] -0.175773152
 [7,] -0.080512728
 [8,]  0.036393052
 [9,] -0.063170450
[10,]  0.073065252
```

Logistic regression

In statistics, logistic regression or logit regression is a type of probabilistic classification model. Logistic regression is used extensively in numerous disciplines, including the medical and social science fields. It can be binomial or multinomial.

Binary logistic regression deals with situations in which the outcome for a dependent variable can have two possible types. Multinomial logistic regression deals with situations where the outcome can have three or more possible types.

Logistic regression can be implemented using logistic functions, which are listed here.

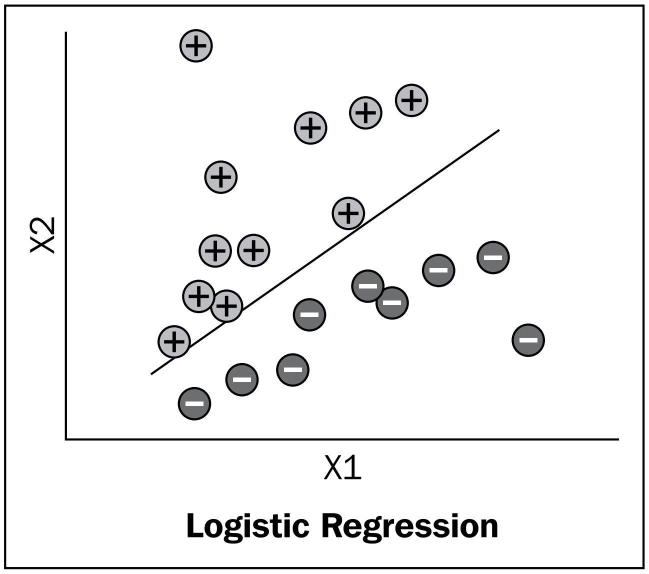
- To predict the log odds ratios, use the following formula:

  $\text{logit}(p) = \beta 0 + \beta 1 \times x1 + \beta 2 \times x2 + \ldots + \beta n \times xn$

- The probability formula is as follows:

  $p = e^{\text{logit}(p)} / 1 + e^{\text{logit}(p)}$

`logit(p)` is a linear function of the explanatory variable, X (x1,x2,x3..xn), which is similar to linear regression. So, the output of this function will be in the range 0 to 1. Based on the probability score, we can set its probability range from 0 to 1. In a majority of the cases, if the score is greater than 0.5, it will be considered as 1, otherwise 0. Also, we can say it provides a classification boundary to classify the outcome variable.



The preceding figure is of a training dataset. Based on the training dataset plot, we can say there is one classification boundary generated by the `glm` model in R.

Applications of logistic regression include:

- Predicting the likelihood of an online purchase
- Detecting the presence of diabetes

Logistic regression with R

To perform logistic regression with R, we will use the `iris` dataset and the `glm` model.

```
#loading iris dataset
data(iris)

# Setting up target variable
target <- data.frame(isSetosa=(iris$Species == 'setosa'))

# Adding target to iris and creating new dataset
inputdata <- cbind(target,iris)

# Defining the logistic regression formula
formula <- isSetosa ~ Sepal.Length + Sepal.Width + Petal.Length +
Petal.Width

# running Logistic model via glm()
logisticModel <- glm(formula, data=inputdata, family="binomial")
```

Logistic regression with R and Hadoop

To perform logistic regression with R and Hadoop, we will use RHadoop with `rmr2`.

The outline of the logistic regression algorithm is as follows:

- Defining the `lr.map` Mapper function
- Defining the `lr.reducer` Reducer function
- Defining the `logistic.regression` MapReduce function

Let's understand them one by one.

We will first define the logistic regression function with gradient decent. Multivariate regression can be performed by forming the nondependent variable into a matrix data format. For factorial variables, we can translate them to binary variables for fitting the model. This function will ask for `input`, `iterations`, `dims`, and `alpha` as input parameters.

- `lr.map`: This stands for the logistic regression Mapper, which will compute the contribution of subset points to the gradient.

  ```
  # Mapper - computes the contribution of a subset of points to the
  gradient.

  lr.map =
      function(., M) {
        Y = M[,1]
        X = M[,-1]
        keyval(
          1,
          Y * X *
            g(-Y * as.numeric(X %*% t(plane))))}
  ```

- `lr.reducer`: This stands for the logistic regression Reducer, which is performing just a big sum of all the values of key 1.

  ```
  # Reducer - Perform sum operation over Mapper output.

  lr.reduce =
      function(k, Z)
        keyval(k, t(as.matrix(apply(Z,2,sum))))
  ```

- `logistic.regression`: This will mainly define the `logistic.regression` MapReduce function with the following input parameters. Calling this function will start executing logistic regression of the MapReduce function.
  - `input`: This is an input dataset
  - `iterations`: This is the fixed number of iterations for calculating the gradient
  - `dims`: This is the dimension of input variables
  - `alpha`: This is the learning rate

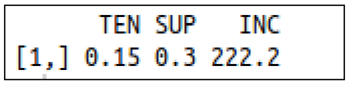
Let's see how to develop the logistic regression function.

```
# MapReduce job - Defining MapReduce function for executing logistic
regression

logistic.regression =
  function(input, iterations, dims, alpha){
  plane = t(rep(0, dims))
  g = function(z) 1/(1 + exp(-z))
  for (i in 1:iterations) {
    gradient =
      values(
        from.dfs(
          mapreduce(
            input,
            map = lr.map,
            reduce = lr.reduce,
            combine = T)))
    plane = plane + alpha * gradient }
  plane }
```

Let's run this logistic regression function as follows:

```
# Loading dataset
data(foodstamp)

# Storing data to hdfs
testdata <-  to.dfs(as.matrix(foodstamp))

# Running logistic regression with R and Hadoop
print(logistic.regression(testdata,10,3,0.05))
```

The output of the preceding command will be as follows:



```
      TEN SUP   INC
[1,] 0.15 0.3 222.2
```

Unsupervised machine learning algorithm

In machine learning, unsupervised learning is used for finding the hidden structure from the unlabeled dataset. Since the datasets are not labeled, there will be no error while evaluating for potential solutions.

Unsupervised machine learning includes several algorithms, some of which are as follows:

- Clustering
- Artificial neural networks
- Vector quantization

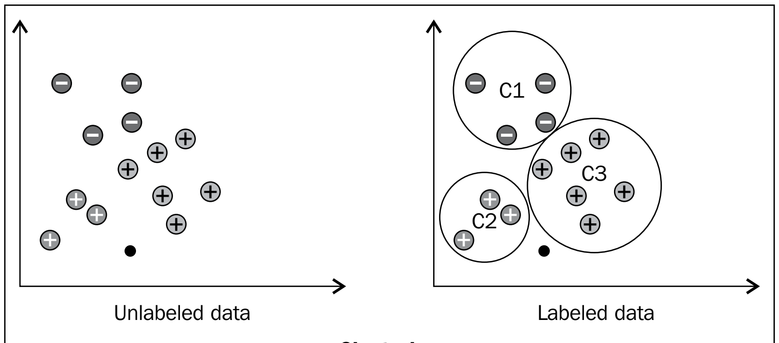
We will consider popular clustering algorithms here.

Clustering

Clustering is the task of grouping a set of object in such a way that similar objects with similar characteristics are grouped in the same category, but other objects are grouped in other categories. In clustering, the input datasets are not labeled; they need to be labeled based on the similarity of their data structure.

In unsupervised machine learning, the classification technique performs the same procedure to map the data to a category with the help of the provided set of input training datasets. The corresponding procedure is known as clustering (or cluster analysis), and involves grouping data into categories based on some measure of inherent similarity; for example, the distance between data points.

From the following figure, we can identify clustering as grouping objects based on their similarity:



Clustering

There are several clustering techniques available within R libraries, such as k-means, k-medoids, hierarchical, and density-based clustering. Among them, k-means is widely used as the clustering algorithm in data science. This algorithm asks for a number of clusters to be the input parameters from the user side.

Applications of clustering are as follows:

- Market segmentation
- Social network analysis
- Organizing computer network
- Astronomical data analysis

Clustering with R

We are considering the `k-means` method here for implementing the clustering model over the `iris` input dataset, which can be achieved by just calling its in-built R dataset – the `iris` data (for more information, visit `http://stat.ethz.ch/R-manual/R-devel/library/datasets/html/iris.html`). Here we will see how k-means clustering can be performed with R.

```
# Loading iris flower dataset
data("iris")
# generating clusters for iris dataset
kmeans <- kmeans(iris[, -5], 3, iter.max = 1000)

# comparing iris Species with generated cluster points
Comp <- table(iris[, 5], kmeans$cluster)
```

Deriving clusters for small datasets is quite simple, but deriving it for huge datasets requires the use of Hadoop for providing computation power.

Performing clustering with R and Hadoop

Since the k-means clustering algorithm is already developed in RHadoop, we are going to use and understand it. You can make changes in their Mappers and Reducers as per the input dataset format. As we are dealing with Hadoop, we need to develop the Mappers and Reducers to be run on nodes in a parallel manner.

The outline of the clustering algorithm is as follows:

- Defining the `dist.fun` distance function
- Defining the `k-means.map` k-means Mapper function
- Defining the `k-means.reduce` k-means Reducer function
- Defining the `k-means.mr` k-means MapReduce function
- Defining input data points to be provided to the clustering algorithms

Now we will run `k-means.mr` (the k-means MapReduce job) by providing the required parameters.

Let's understand them one by one.

- `dist.fun`: First, we will see the `dist.fun` function for calculating the distance between a matrix of center `C` and a matrix of point `P`, which is tested. It can produce 10^6 points and 10^2 centers in five dimensions in approximately 16 seconds.

```
# distance calculation function
dist.fun =
      function(C, P) {
        apply(
          C,
          1,
          function(x)
            colSums((t(P) - x)^2))}
```

- `k-means.map`: The Mapper of the k-means MapReduce algorithm will compute the distance between points and all the centers and return the closest center for each point. This Mapper will run in iterations based on the following code. With the first iteration, the cluster center will be assigned randomly and from the next iteration, it will calculate these cluster centers based on the minimum distance from all the points of the cluster.

```
# k-Means Mapper
  kmeans.map =
      function(., P) {
        nearest = {

# First interations- Assign random cluster centers
          if(is.null(C))
            sample(
              1:num.clusters,
              nrow(P),
              replace = T)
# Rest of the iterations, where the clusters are assigned # based
on the minimum distance from points
          else {
            D = dist.fun(C, P)
            nearest = max.col(-D)}}

       if(!(combine || in.memory.combine))
          keyval(nearest, P)
        else
          keyval(nearest, cbind(1, P))}
```

- `k-means.reduce`: The Reducer of the k-means MapReduce algorithm will compute the column average of matrix points as key.

```
# k-Means Reducer
kmeans.reduce = {

# calculating the column average for both of the
# conditions

      if (!(combine || in.memory.combine) )
        function(., P)
          t(as.matrix(apply(P, 2, mean)))
      else
        function(k, P)
          keyval(
            k,
            t(as.matrix(apply(P, 2, sum))))}
```

- `kmeans.mr`: Defining the k-means MapReduce function involves specifying several input parameters, which are as follows:
    - `P`: This denotes the input data points
    - `num.clusters`: This is the total number of clusters
    - `num.iter`: This is the total number of iterations to be processed with datasets
    - `combine`: This will decide whether the Combiner should be enabled or disabled (`TRUE` or `FALSE`)

```
# k-Means MapReduce - for
kmeans.mr =
  function(
    P,
    num.clusters,
    num.iter,
    combine,
    in.memory.combine) {
    C = NULL
    for(i in 1:num.iter ) {
      C =
        values(

# Loading hdfs dataset
              from.dfs(

# MapReduce job, with specification of input dataset,
# Mapper and Reducer
                mapreduce(
                  P,
                  map = kmeans.map,
                  reduce = kmeans.reduce)))
      if(combine || in.memory.combine)
        C = C[, -1]/C[, 1]
      if(nrow(C) < num.clusters) {
        C =
              rbind(
                C,
                matrix(
                  rnorm(
                    (num.clusters -
                       nrow(C)) * nrow(C)),
                  ncol = nrow(C)) %*% C) }}
        C}
```

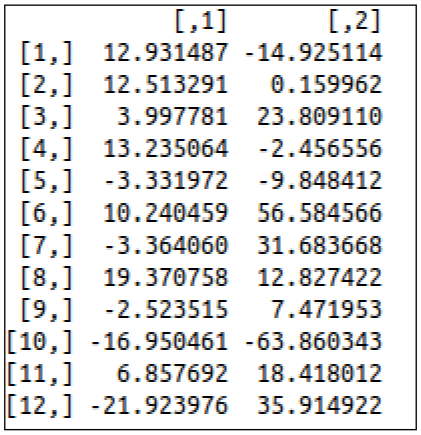
- Defining the input data points to be provided to the clustering algorithms:

```
# Input data points
P = do.call(
      rbind,
      rep(


        list(

# Generating Matrix of
            matrix(
# Generate random normalized data with sd = 10
              rnorm(10, sd = 10),
              ncol=2)),
        20)) +
    matrix(rnorm(200), ncol =2)
```

- Running `kmeans.mr` (the k-means MapReduce job) by providing it with the required parameters.

```
# Running kmeans.mr Hadoop MapReduce algorithms with providing the
required input parameters

kmeans.mr(
      to.dfs(P),
      num.clusters = 12,
      num.iter = 5,
      combine = FALSE,
      in.memory.combine = FALSE)
```

- The output of the preceding command is shown in the following screenshot:



```
            [,1]       [,2]
 [1,]  12.931487 -14.925114
 [2,]  12.513291   0.159962
 [3,]   3.997781  23.809110
 [4,]  13.235064  -2.456556
 [5,]  -3.331972  -9.848412
 [6,]  10.240459  56.584566
 [7,]  -3.364060  31.683668
 [8,]  19.370758  12.827422
 [9,]  -2.523515   7.471953
[10,] -16.950461 -63.860343
[11,]   6.857692  18.418012
[12,] -21.923976  35.914922
```

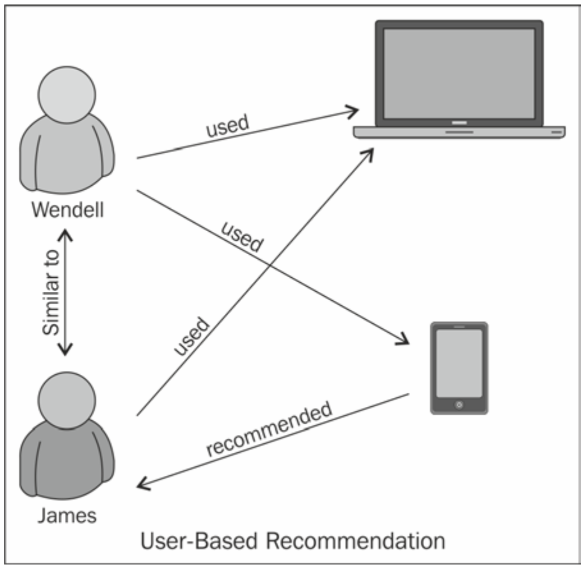
Recommendation algorithms

Recommendation is a machine-learning technique to predict what new items a user would like based on associations with the user's previous items. Recommendations are widely used in the field of e-commerce applications. Through this flexible data and behavior-driven algorithms, businesses can increase conversions by helping to ensure that relevant choices are automatically suggested to the right customers at the right time with cross-selling or up-selling.

For example, when a customer is looking for a Samsung Galaxy S IV/S4 mobile phone on Amazon, the store will also suggest other mobile phones similar to this one, presented in the **Customers Who Bought This Item Also Bought** window.

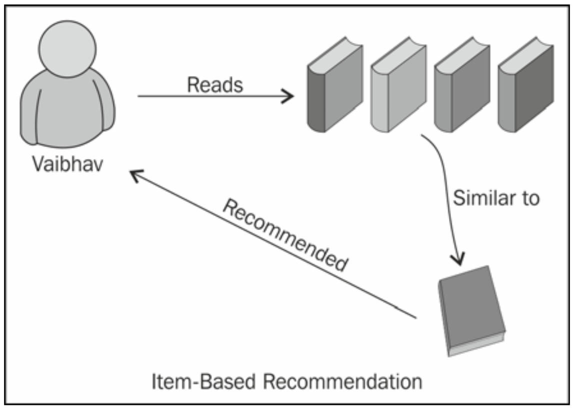
There are two different types of recommendations:

- **User-based recommendations**: In this type, users (customers) similar to current user (customer) are determined. Based on this user similarity, their interested/used items can be recommended to other users. Let's learn it through an example.



  Assume there are two users named Wendell and James; both have a similar interest because both are using an iPhone. Wendell had used two items, iPad and iPhone, so James will be recommended to use iPad. This is user-based recommendation.

- **Item-based recommendations**: In this type, items similar to the items that are being currently used by a user are determined. Based on the item-similarity score, the similar items will be presented to the users for cross-selling and up-selling type of recommendations. Let's learn it through an example.



For example, a user named Vaibhav likes and uses the following books:

- *Apache Mahout Cookbook*, *Piero Giacomelli*, *Packt Publishing*
- *Hadoop MapReduce Cookbook*, *Thilina Gunarathne* and *Srinath Perera*, *Packt Publishing*
- *Hadoop Real-World Solutions Cookbook*, *Brian Femiano*, *Jon Lentz*, and *Jonathan R. Owens*, *Packt Publishing*
- *Big Data For Dummies*, *Dr. Fern Halper*, *Judith Hurwitz*, *Marcia Kaufman*, and *Alan Nugent*, *John Wiley & Sons Publishers*

Based on the preceding information, the recommender system will predict which new books Vaibhav would like to read, as follows:

- *Big Data Analytics with R and Hadoop*, *Vignesh Prajapati*, *Packt Publishing*

Now we will see how to generate recommendations with R and Hadoop. But before going towards the R and Hadoop combination, let us first see how to generate it with R. This will clear the concepts to translate your generated recommender systems to MapReduce recommendation algorithms. In case of generating recommendations with R and Hadoop, we will use the RHadoop distribution of **Revolution Analytics**.

Steps to generate recommendations in R

To generate recommendations for users, we need to have datasets in a special format that can be read by the algorithm. Here, we will use the collaborative filtering algorithm for generating the recommendations rather than content-based algorithms. Hence, we will need the user's rating information for the available item sets. So, the `small.csv` dataset is given in the format `user ID, item ID, item's ratings`.

```
# user ID, item ID, item's rating
1,         101,     5.0
1,         102,     3.0
1,         103,     2.5
2,         101,     2.0
2,         102,     2.5
2,         103,     5.0
2,         104,     2.0
3,         101,     2.0
3,         104,     4.0
3,         105,     4.5
3,         107,     5.0
4,         101,     5.0
4,         103,     3.0
4,         104,     4.5
4,         106,     4.0
5,         101,     4.0
5,         102,     3.0
5,         103,     2.0
5,         104,     4.0
5,         105,     3.5
5,         106,     4.0
```

The preceding code and datasets are reproduced from the book *Mahout in Action*, *Robin Anil*, *Ellen Friedman*, *Ted Dunning*, and *Sean Owen*, *Manning Publications* and the website is `http://www.fens.me/`.

Recommendations can be derived from the matrix-factorization technique as follows:

```
Co-occurrence matrix * scoring matrix = Recommended Results
```

To generate the recommenders, we will follow the given steps:

1. Computing the co-occurrence matrix.
2. Establishing the user-scoring matrix.
3. Generating recommendations.

From the next section, we will see technical details for performing the preceding steps.

1. In the first section, computing the co-occurrence matrix, we will be able to identify the co-occurred item sets given in the dataset. In simple words, we can call it counting the pair of items from the given dataset.

```
# Quote plyr package
library (plyr)

# Read dataset
train <-read.csv (file = "small.csv", header = FALSE)
names (train) <-c ("user", "item", "pref")

# Calculated User Lists
usersUnique <-function () {
  users <-unique (train $ user)
  users [order (users)]
}

# Calculation Method Product List
itemsUnique <-function () {
  items <-unique (train $ item)
  items [order (items)]
}

# Derive unique User Lists
users <-usersUnique ()

# Product List
items <-itemsUnique ()


# Establish Product List Index
index <-function (x) which (items %in% x)
data<-ddply(train,.(user,item,pref),summarize,idx=index(item))

# Co-occurrence matrix
Co-occurrence <-function (data) {
  n <-length (items)
  co <-matrix (rep (0, n * n), nrow = n)
  for (u in users) {
    idx <-index (data $ item [which(data$user == u)])
    m <-merge (idx, idx)
    for (i in 1: nrow (m)) {
      co [m$x[i], m$y[i]] = co[m$x[i], m$y[i]]+1
    }
  }
  return (co)
}

# Generate co-occurrence matrix
co <-co-occurrence (data)
```

2. To establish the user-scoring matrix based on the user's rating information, the user-item rating matrix can be generated for users.

```
# Recommendation algorithm
recommend <-function (udata = udata, co = coMatrix, num = 0) {
  n <- length(items)

  # All of pref
  pref <- rep (0, n)
  pref[udata$idx] <-udata$pref

  # User Rating Matrix
  userx <- matrix(pref, nrow = n)

  # Scoring matrix co-occurrence matrix *
  r <- co %*% userx

  # Recommended Sort
  r[udata$idx] <-0
  idx <-order(r, decreasing = TRUE)
  topn <-data.frame (user = rep(udata$user[1], length(idx)), item
= items[idx], val = r[idx])

  # Recommended results take months before the num
  if (num> 0) {
    topn <-head (topn, num)
  }

  # Recommended results take months before the num
  if (num> 0) {
    topn <-head (topn, num)
  }

  # Back to results
  return (topn)
}
```

3. Finally, the recommendations as output can be generated by the product operations of both matrix items: co-occurrence matrix and user's scoring matrix.

```
# initializing dataframe for recommendations storage
recommendation<-data.frame()

# Generating recommendations for all of the users
for(i in 1:length(users)){
  udata<-data[which(data$user==users[i]),]
  recommendation<-rbind(recommendation,recommend(udata,co,0))
}
```

[Generating recommendations via **Myrrix** and R interface is quite easy. For more information, refer to `https://github.com/jwijffels/Myrrix-R-interface`.]

Generating recommendations with R and Hadoop

To generate recommendations with R and Hadoop, we need to develop an algorithm that will be able to run and perform data processing in a parallel manner. This can be implemented using Mappers and Reducers. A very interesting part of this section is how we can use R and Hadoop together to generate recommendations from big datasets.

So, here are the steps that are similar to generating recommendations with R, but translating them to the Mapper and Reducer paradigms is a little tricky:

1. Establishing the co-occurrence matrix items.
2. Establishing the user scoring matrix to articles.
3. Generating recommendations.

We will use the same concepts as our previous operation with R to generate recommendations with R and Hadoop. But in this case, we need to use a key-value paradigm as it's the base of parallel operations. Therefore, every function will be implemented by considering the key-value paradigm.

1. In the first section, establishment of the co-occurrence matrix items, we will establish co-occurrence items in steps: grouped by user, locate each user-selected items appearing alone counting, and counting in pairs.

   ```
   # Load rmr2 package
   library (rmr2)

   # Input Data File
   train <-read.csv (file = "small.csv", header = FALSE)
   names (train) <-c ("user", "item", "pref")

   # Use the hadoop rmr format, hadoop is the default setting.
   rmr.options (backend = 'hadoop')

   # The data set into HDFS
   train.hdfs = to.dfs (keyval (train$user, train))

   # see the data from hdfs
   from.dfs (train.hdfs)
   ```

   The key points to note are:

   - `train.mr`: This is the MapReduce job's key-value paradigm information
   - **key**: This is the list of items vector
   - **value**: This is the item combination vector

   ```
   # MapReduce job 1 for co-occurrence matrix items
   train.mr <-mapreduce (
     train.hdfs,
     map = function (k, v) {
       keyval (k, v$item)
     }

   # for identification of co-occurrence items
     , Reduce = function (k, v) {
       m <-merge (v, v)
       keyval (m$x, m$y)
     }
   )
   ```

The co-occurrence matrix items will be combined to count them.

To define a MapReduce job, `step2.mr` is used for calculating the frequency of the combinations of items.

- `Step2.mr`: This is the MapReduce job's key value paradigm information
- **key**: This is the list of items vector
- **value**: This is the co-occurrence matrix dataframe value (`item`, `item`, `Freq`)

```
# MapReduce function for calculating the frequency of the
combinations of the items.
step2.mr <-mapreduce (
  train.mr,

  map = function (k, v) {
    d <-data.frame (k, v)
    d2 <-ddply (d,. (k, v), count)

    key <- d2$k
    val <- d2
    keyval(key, val)
  }
)

# loading data from HDFS
from.dfs(step2.mr)
```

2. To establish the user-scoring matrix to articles, let us define the `Train2.mr` MapReduce job.

```
# MapReduce job for establish user scoring matrix to articles

train2.mr <-mapreduce (
  train.hdfs,
  map = function(k, v) {
      df <- v

# key as item
    key <-df $ item

# value as [item, user pref]
    val <-data.frame (item = df$item, user = df$user, pref =
df$pref)
# emitting (key, value)pairs
    keyval(key, val)
  }
)

# loading data from HDFS
from.dfs(train2.mr)
```

- `Train2.mr`: This is the MapReduce job's key value paradigm information
- **key**: This is the list of items
- **value**: This is the value of the user goods scoring matrix

The following is the consolidation and co-occurrence scoring matrix:

```
# Running equi joining two data – step2.mr and train2.mr
eq.hdfs <-equijoin (
  left.input = step2.mr,
  right.input = train2.mr,
  map.left = function (k, v) {
    keyval (k, v)
  },
  map.right = function (k, v) {
    keyval (k, v)
  },
  outer = c ("left")
)

# loading data from HDFS
from.dfs (eq.hdfs)
```

- `eq.hdfs`: This is the MapReduce job's key value paradigm information
- **key**: The key here is null
- **value**: This is the merged dataframe value

3. In the section of generating recommendations, we will obtain the recommended list of results.

```
# MapReduce job to obtain recommended list of result from
equijoined data
cal.mr <-mapreduce (
  input = eq.hdfs,

  map = function (k, v) {
    val <-v
    na <-is.na (v$user.r)
    if (length (which(na))> 0) val <-v [-which (is.na (v $
user.r)),]
    keyval (val$kl, val)
  }
  , Reduce = function (k, v) {
    val <-ddply (v,. (kl, vl, user.r), summarize, v = freq.l *
pref.r)
    keyval (val $ kl, val)
  }
)

# loading data from HDFS
from.dfs (cal.mr)
```

   ◦ `Cal.mr`: This is the MapReduce job's key value paradigm information
   ◦ **key**: This is the list of items
   ◦ **value**: This is the recommended result dataframe value

By defining the result for getting the list of recommended items with preference value, the sorting process will be applied on the recommendation result.

```
# MapReduce job for sorting the recommendation output
result.mr <-mapreduce (
  input = cal.mr,
  map = function (k, v) {
    keyval (v $ user.r, v)
  }
  , Reduce = function (k, v) {
    val <-ddply (v,. (user.r, vl), summarize, v = sum (v))
    val2 <-val [order (val$v, decreasing = TRUE),]
    names (val2) <-c ("user", "item", "pref")
    keyval (val2$user, val2)
  }
)
# loading data from HDFS
from.dfs (result.mr)
```

- `result.mr`: This is the MapReduce job's key value paradigm information
- **key**: This is the user ID
- **value**: This is the recommended outcome dataframe value

Here, we have designed the collaborative algorithms for generating item-based recommendation. Since we have tried to make it run on parallel nodes, we have focused on the Mapper and Reducer. They may not be optimal in some cases, but you can make them optimal by using the available code.

Summary

In this chapter, we learned how we can perform Big Data analytics with machine learning with the help of R and Hadoop technologies. In the next chapter, we will learn how to enrich datasets in R by integrating R to various external data sources.