Chapter 5
Linear Algebra & Matrix Computing

Linear algebra is a branch of mathematics that studies linear associations using vectors, vector-spaces, linear equations, linear transformations, and matrices. It is generally challenging to visualize complex data, e.g., large vectors, tensors, and tables in n-dimensional Euclidian spaces ($n > 3$). Linear algebra allows us to mathematically represent, computationally model, statistically analyze, synthetically simulate, and visually summarize such complex data.

Virtually all natural processes permit first-order linear approximations. This is useful because linear equations are easy to write, interpret, and solve. These first order approximations may be useful to practically assess the process, determine general trends, identify potential patterns, and suggest associations in the data.

Linear equations represent the simplest type of models for many processes. Higher order models may include additional non-linear terms, e.g., Taylor-series expansions. Linear algebra provides the foundation for linear representation, analytics, computatiponal solutions, inference, and visualization of first-order affine models. Linear algebra is a small part of the larger mathematics field of *functional analysis*, which is actually the infinite-dimensional version of linear algebra.

Specifically, *linear algebra* allows us to **computationally** manipulate, model, solve, and interpret complex systems of equations representing large numbers of dimensions and variables. Arbitrarily large problems can be mathematically transformed into simple matrix equations of the form $Ax = b$ or $Ax = \lambda x$.

In this chapter, we review the fundamentals of linear algebra, matrix manipulation and their applications to represent, model, and analyse real data. Specifically, we will cover (1) construction of matrices and matrix operations, (2) general matrix algebra notations, (3) eigenvalues and eigenvectors of linear operators, (4) least squares estimation, and (5) linear regression and variance-covariance matrices.

5.1 Matrices (Second Order Tensors)

5.1.1 Create Matrices

The easiest way to create a matrix is by using the `matrix()` function, which organizes the elements of a vector into specified positions into a matrix.

```
seq1<-seq(1:6)
m1<-matrix(seq1, nrow=2, ncol=3)
m1

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

m2<-diag(seq1)
m2

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    0    0    0    0
## [2,]    0    2    0    0    0    0
## [3,]    0    0    3    0    0    0
## [4,]    0    0    0    4    0    0
## [5,]    0    0    0    0    5    0
## [6,]    0    0    0    0    0    6

m3<-matrix(rnorm(20), nrow=5)
m3

##            [,1]        [,2]       [,3]       [,4]
## [1,]  0.4877535  0.22081284 -0.6067573 -0.8982306
## [2,] -0.1672924 -1.49020015  0.3038424 -0.1875045
## [3,] -0.4771204 -0.39004837  1.1160825 -0.6948070
## [4,] -0.9274687  0.08378863  0.3846627  0.2386284
## [5,]  0.8672767 -0.86752831  1.5536853  0.3222158
```

The function `diag()` is very useful. When the object is a vector, it creates a diagonal matrix with the vector in the principal diagonal.

```
diag(c(1, 2, 3))

##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

When the object is a matrix, `diag()` returns its principal diagonal.

```
diag(m1)

## [1] 1 4
```

When the object is a scalar, `diag(k)` returns a $k \times k$ identity matrix.

```
diag(4)

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

5.1.2 *Adding Columns and Rows*

Function `cbind()` and `rbind()` are used throughout the textbook.

```
c1<-1:5
m4<-cbind(m3, c1)
m4

##                                                    c1
## [1,]  0.4877535  0.22081284 -0.6067573 -0.8982306  1
## [2,] -0.1672924 -1.49020015  0.3038424 -0.1875045  2
## [3,] -0.4771204 -0.39004837  1.1160825 -0.6948070  3
## [4,] -0.9274687  0.08378863  0.3846627  0.2386284  4
## [5,]  0.8672767 -0.86752831  1.5536853  0.3222158  5

r1<-1:4
m5<-rbind(m3, r1)
m5

##          [,1]        [,2]       [,3]       [,4]
##     0.4877535  0.22081284 -0.6067573 -0.8982306
##    -0.1672924 -1.49020015  0.3038424 -0.1875045
##    -0.4771204 -0.39004837  1.1160825 -0.6948070
##    -0.9274687  0.08378863  0.3846627  0.2386284
##     0.8672767 -0.86752831  1.5536853  0.3222158
## r1  1.0000000  2.00000000  3.0000000  4.0000000
```

Note that `m5` has a row name `r1` in the fourth row. We can remove row/column names by naming them as `NULL`.

```
dimnames(m5)<-list(NULL, NULL)
m5

##            [,1]        [,2]       [,3]       [,4]
## [1,]  0.4877535  0.22081284 -0.6067573 -0.8982306
## [2,] -0.1672924 -1.49020015  0.3038424 -0.1875045
## [3,] -0.4771204 -0.39004837  1.1160825 -0.6948070
## [4,] -0.9274687  0.08378863  0.3846627  0.2386284
## [5,]  0.8672767 -0.86752831  1.5536853  0.3222158
## [6,]  1.0000000  2.00000000  3.0000000  4.0000000
```

5.2 Matrix Subscripts

Each element in a matrix has a location. `A[i, j]` means the *i*th row and *j*th column in a matrix *A*. We can also access specific rows or columns using matrix subscripts.

```
m6<-matrix(1:12, nrow=3)
m6

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

m6[1, 2]

## [1] 4

m6[1, ]

## [1]  1  4  7 10

m6[, 2]

## [1] 4 5 6

m6[, c(2, 3)]

##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
## [3,]    6    9
```

5.3 Matrix Operations

5.3.1 *Addition*

Elements in the same position are added to represent the result at the same location.

```
m7<-matrix(1:6, nrow=2)
m7

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

m8<-matrix(2:7, nrow = 2)
m8

##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7

m7+m8

##      [,1] [,2] [,3]
## [1,]    3    7   11
## [2,]    5    9   13
```

5.3.2 *Subtraction*

Similar to addition, matrix subtraction reflects differences between elements in same position.

```
m8-m7

##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1

m8-1

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

5.3.3 *Multiplication*

Multiplicative oparations are different than additive operations. We can do elementwise multiplication or matrix multiplication. For matrix multiplication, the matrix dimensions have to match. That is, the number of columns in the first matrix must equal to the number of rows in the second matrix.

Elementwise Multiplication

Multiplication between elements in same position.

```
m8*m7

##      [,1] [,2] [,3]
## [1,]    2   12   30
## [2,]    6   20   42
```

Matrix Multiplication

The resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

```
dim(m8)

## [1] 2 3

m9<-matrix(3:8, nrow=3)
m9

##      [,1] [,2]
## [1,]    3    6
## [2,]    4    7
## [3,]    5    8

dim(m9)

## [1] 3 2

m8%*%m9

##      [,1] [,2]
## [1,]   52   88
## [2,]   64  109
```

We made a 2×2 matrix resulting from multiplying two matrices $2 \times 3 * 3 \times 2$.

The process of multiplying two vectors is called **outer product**. Assume we have two vectors u and v, in matrix multiplication their outer product is represented mathematically as uv^T. In R, the operator for outer product is `%o%`.

```
u<-c(1, 2, 3, 4, 5)
v<-c(4, 5, 6, 7, 8)
u%o%v

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4    5    6    7    8
## [2,]    8   10   12   14   16
## [3,]   12   15   18   21   24
## [4,]   16   20   24   28   32
## [5,]   20   25   30   35   40

u%*%t(v)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4    5    6    7    8
## [2,]    8   10   12   14   16
## [3,]   12   15   18   21   24
## [4,]   16   20   24   28   32
## [5,]   20   25   30   35   40
```

What are the differences between $u \% * \% t(v)$, $u \% * \% t(v)$, $u * t(v)$, and $u * v$?

5.3.4 Element-wise Division

Elementwise division is defined similarly to alement-wize multipliaiton, however, this is different than multiplicative inversion.

```
m8/m7

##      [,1]     [,2]     [,3]
## [1,]  2.0 1.333333 1.200000
## [2,]  1.5 1.250000 1.166667

m8/2

##      [,1] [,2] [,3]
## [1,]  1.0  2.0  3.0
## [2,]  1.5  2.5  3.5
```

5.3.5 Transpose

The transpose of a matrix is a new matrix created by swapping the columns and the rows of the original matrix. Do this in a simple function `t()`.

```
m8

##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7

t(m8)

##      [,1] [,2]
## [1,]    2    3
## [2,]    4    5
## [3,]    6    7
```

Notice that the [1, 2] element in `m8` is the [2, 1] element in the transpose matrix `t(m8)`.

5.3.6 Multiplicative Inverse

The inverse of a matrix (A^{-1}) is its multiplicative inverse. That is, multiplying the original matrix (A) by it's inverse (A^{-1}) yields an identity matrix that has 1's on the diagonal and 0's off the diagonal.

$$AA^{-1} = I$$

Assume we have the following 2×2 matrix:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Its matrix inverse is

$$\frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

For higher dimensions, the formula for computing the inverse matrix is more complex. In R, we can use the `solve()` function to calculate the matrix inverse, if it exists.

```
m10<-matrix(1:4, nrow=2)
m10

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

solve(m10)

##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5

m10%*%solve(m10)

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Note that only some matrices have inverses. These are square matrices, i.e., they have the same number of rows and columns, and are non-singular.

Another function that can help us compute the inverse of a matrix is the `ginv()` function under the `MASS` package, which reports the Moore-Penrose Generalized Inverse of a matrix.

```
require(MASS)

## Loading required package: MASS

ginv(m10)

##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

Also, the samae function `solve()` can be used to solve matrix equations. `solve(A, b)` returns vector x in the equation $b = Ax$ (i.e., $x = A^{-1}b$).

```
s1<-diag(c(2, 4, 6, 8))
s2<-c(1, 2, 3, 4)
solve(s1, s2)

## [1] 0.5 0.5 0.5 0.5
```

The following Table 5.1 summarizes some basic matrix operation functions.

Table 5.1 Basic matrix operators in R

Expression	Explanation
`t(x)`	Transpose
`diag(x)`	Diagonal
`%*%`	Matrix multiplication
`solve(a, b)`	Solves `a %*% x` = `b` for `x`
`solve(a)`	Matrix inverse of `a`
`rowsum(x)`	Sum of rows for a matrix-like object. `rowSums(x)` is a faster version
`colSums(x)`, `colSums(x)`	Id. for columns
`rowMeans(x)`	Fast version of row means
`colMeans(x)`	Id. for columns

```
mat1 <- cbind(c(1, -1/5), c(-1/3, 1))
mat1.inv <- solve(mat1)

mat1.identity <- mat1.inv %*% mat1
mat1.identity

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1

b <- c(1, 2)
x <- solve (mat1, b)
x

## [1] 1.785714 2.357143
```

5.4 Matrix Algebra Notation

Let's introduce the basic matrix notation. The product AB between matrices A and B is defined only if the number of columns in A equals the number of rows in B. That is, we can multiply an $m \times n$ matrix A by an $n \times k$ matrix B and the result will be $AB_{m \times k}$ matrix. Each element of the product matrix, $(AB_{i,\,j})$, represents the product of the ith row in A and the jth column in B, which are of the same size n. Matrix multiplication is `row-by-column`.

5.4.1 Linear Models

Linear algebra notation simplifies the mathematical descriptions and manipulations of linear models, as well as coding in R.

The main point is to show how we can write linear models using matrix notation. Later, we'll explain how this is useful for solving the least squares matrix equation. Let's start by defining the notation and matrix multiplication.

5.4.2 Solving Systems of Equations

Linear algebra notation enables the mathematical analysis and the analytical solution of systems of linear equations:

$$\begin{aligned} a+b+2c &= 6 \\ 3a-2b+c &= 2\,. \\ 2a+b-c &= 3 \end{aligned}$$

It provides a generic machinery for solving these problems.

$$\underbrace{\begin{pmatrix} 1 & 1 & 2 \\ 3 & -2 & 1 \\ 2 & 1 & -1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 6 \\ 2 \\ 3 \end{pmatrix}}_{b}.$$

That is: $Ax = b$, which yields a solution vector x:

$$x = \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1 & 1 & 2 \\ 3 & -2 & 1 \\ 2 & 1 & -1 \end{pmatrix}^{-1} \begin{pmatrix} 6 \\ 2 \\ 3 \end{pmatrix}.$$

In other words, $A^{-1}Ax = x = A^{-1}b$.

Notice that this parallels the solution of simple (univariate) linear equations like:

$$\underbrace{2}_{(\text{design matrix})\ A} \quad \underbrace{x}_{\text{unknown } x} \quad - \quad \underbrace{3}_{\text{simple constant term}} \quad = \underbrace{5}_{b}\,.$$

The constant term, -3, can be simply joined with the right-hand-size, b, to form a new term $b^{'} = 5 + 3 = 8$. Thus, the shifting factor is mostly ignored in linear models, or linear equations, to simplify the equation to:

$$\underbrace{2}_{(\text{design matrix})\ A} \quad \underbrace{x}_{\text{unknown } x} \quad = \underbrace{5+3}_{b'} = \underbrace{8}_{b'}\,.$$

This (simple) linear equation is solved by multiplying both sides by the inverse (reciprocal) of the x multiplier, 2:

$$\frac{1}{2}2x = \frac{1}{2}8.$$

Thus, the unique solution is:

$$x = \frac{1}{2}8 = 4.$$

So, let's use exactly the same protocol to solve the corresponding matrix equation (linear equations, $Ax = b$) using R (the unknown is x, and the design matrix A and the constant vector b are known):

$$\underbrace{\begin{pmatrix} 1 & 1 & 2 \\ 3 & -2 & 1 \\ 2 & 1 & -1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 6 \\ 2 \\ 3 \end{pmatrix}}_{b}.$$

```
A_matrix_values <- c(1, 1, 2, 3, -2, 1, 2, 1, -1)
A <- t(matrix(A_matrix_values, nrow=3, ncol=3))
b <- c(6, 2, 3)
# to solve Ax = b, x=A^{-1}*b
x <- solve (A, b)
# Ax = b ==> x = A^{-1} * b
x

## [1] 1.35 1.75 1.45

# Check the Solution x=(1.35 1.75 1.45)
LHS <- A %*% x
round (LHS-b)

##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
```

How about if we want to triple-check the accuracy of the `solve` method to provide accurate solutions to matrix-based systems of linear equations?

We can generate the solution (x) to the equation $Ax = b$ using first principles:

$$x = A^{-1}b.$$

```
A.inverse <- solve(A) # the inverse matrix A^{-1}
x1 <- A.inverse %*% b
# check if X and x1 are the same
x; x1

## [1] 1.35 1.75 1.45

##      [,1]
## [1,] 1.35
## [2,] 1.75
## [3,] 1.45

round(x-x1,6)

##              [,1]
## [1,] 0
## [2,] 0
## [3,] 0
```

5.4.3 The Identity Matrix

The identity matrix is the matrix analog to the multiplicative numeric identity, i.e., the number 1. Multiplying the identity matrix by any other matrix (B) does not change the matrix B. For this to happen, the multiplicative identity matrix must look like:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

The identity matrix is always a square matrix with diagonal elements 1 and 0 at the off-diagonal elements.

If you follow the matrix multiplication rule above, you notice this works out:

$$\mathbf{X}\times\mathbf{I} = \begin{pmatrix} x_{1,1} & \dots & x_{1,p} \\ & \vdots & \\ x_{n,1} & \dots & x_{n,p} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ & & & \vdots & & \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix} = \begin{pmatrix} x_{1,1} & \dots & x_{1,p} \\ & \vdots & \\ x_{n,1} & \dots & x_{n,p} \end{pmatrix}.$$

In R, you can form an identity matrix as follows:

```r
n <- 3 #pick dimensions
I <- diag(n); I

##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1

A %*% I; I %*% A

##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    1   -2    1
## [3,]    2    1   -1

##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    1   -2    1
## [3,]    2    1   -1
```

5.5 Scalars, Vectors and Matrices

Let's look at this notation deeper. In the baseball player data, there are three quantitative variables: `Heights`, `Weight`, and `Age`. Suppose the variable `Weight` is represented as a `response` $Y_1, \ldots, Y_n$ random vector.

We can examine players' `Weight` as a function of `Age` and `Height`.

```
# Data: https://umich.instructure.com/courses/38100/files/folder/data
(01a_data.txt)
data <- read.table('https://umich.instructure.com/files/330381/download?down
load_frd=1', as.is=T, header=T)
attach(data)
head(data)
```

```
##                Name Team       Position Height Weight   Age
## 1   Adam_Donachie  BAL        Catcher     74    180 22.99
## 2       Paul_Bako  BAL        Catcher     74    215 34.69
## 3 Ramon_Hernandez  BAL        Catcher     72    210 30.78
## 4    Kevin_Millar  BAL  First_Baseman     72    210 35.43
## 5     Chris_Gomez  BAL  First_Baseman     73    188 35.71
## 6   Brian_Roberts  BAL Second_Baseman     69    176 29.39
```

We can also use vector notation. We usually use bold to distinguish vectors from the individual elements:

$$\mathbf{Y} = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix}.$$

The default representation of data vectors is as columns, i.e., we have dimension $n \times 1$, as opposed to $1 \times n$ rows.

Similarly, we can use math notation to represent the covariates or predictors: `Age` and `Height`. In a case with two predictors, we can represent them like this:

$$\mathbf{X}_1 = \begin{pmatrix} x_{1,1} \\ \vdots \\ x_{n,1} \end{pmatrix} \text{ and } \mathbf{X}_2 = \begin{pmatrix} x_{1,2} \\ \vdots \\ x_{n,2} \end{pmatrix}.$$

Note that for the baseball players example, $x_{1,\ 1} = Age_1$ and $x_{i,\ 1} = Age_i$ with Age_i represent the `Age` of the ith player, and similarly, $x_{i,\ 2} = Height_i$, represents the height of the ith player. These vectors are also thought of as $n \times 1$ matrices.

It is convenient to represent both covariates as a matrix:

$$\mathbf{X} = [\mathbf{X}_1 \mathbf{X}_2] = \begin{pmatrix} x_{1,1} & x_{1,2} \\ & \vdots \\ x_{n,1} & x_{n,2} \end{pmatrix}.$$

This matrix is of dimension $n \times 2$ and can be create in R this way:

```
X <- cbind(Age, Height)
head(X)

##        Age Height
## [1,] 22.99     74
## [2,] 34.69     74
## [3,] 30.78     72
## [4,] 35.43     72
## [5,] 35.71     73
## [6,] 29.39     69

dim(X)

## [1] 1034    2
```

We can also use this notation to denote an arbitrary number of covariates (k) with the following $n \times k$ matrix:

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & \dots & x_{1,k} \\ x_{2,1} & \dots & x_{2,k} \\ & \vdots & \\ x_{n,1} & \dots & x_{n,k} \end{pmatrix}.$$

You can simulate such a matrix in R now using `matrix`, instead of `cbind`:

```
n <- 1034; k <- 5
X <- matrix(1:(n*k), n, k)
head(X)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1 1035 2069 3103 4137
## [2,]    2 1036 2070 3104 4138
## [3,]    3 1037 2071 3105 4139
## [4,]    4 1038 2072 3106 4140
## [5,]    5 1039 2073 3107 4141
## [6,]    6 1040 2074 3108 4142

dim(X)

## [1] 1034    5
```

By default, the matrices are filled in a column-by-column order; however using the `byrow=TRUE` argument allows us to change that order to row-by-row:

```
n <- 1034; k <- 5
X <- matrix(1:(n*k), n, k, byrow=TRUE)
head(X)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
## [6,]   26   27   28   29   30

dim(X)

## [1] 1034    5
```

A scalar is just a univariate number, which is different from vectors and matrices, that is usually denoted by lower case letters.

5.5.1 *Sample Statistics (Mean, Variance)*

Mean

To compute the sample average and variance of a dataset, we use the formulas:

$$\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$$

and

$$\operatorname{var}(Y) = \frac{1}{n-1}\sum_{i=1}^{n}\left(Y_i - \bar{Y}\right)^2,$$

which can be represented with matrix multiplication.

Define a $n \times 1$ matrix made of 1's:

$$A = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}.$$

This implies that:

$$\frac{1}{n}\mathbf{A}^\top Y = \frac{1}{n}\begin{pmatrix} 1 & 1 & \dots & 1 \end{pmatrix}\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \frac{1}{n}\sum_{i=1}^{n} Y_i = \bar{Y}.$$

Note that we multiply matrices by scalars, like $\frac{1}{n}$, using the traditional multiplication operator `*`, whereas we multiply two matrices using this operator `%*%`:

```
# Using the Baseball dataset
y <- data$Height
print(mean(y))

## [1] 73.69729

n <- length(y)
Y<- matrix(y, n, 1)
A <- matrix(1, n, 1)
barY=t(A)%*%Y / n
```

```
print(barY)

##          [,1]
## [1,] 73.69729

# double-check the result
mean(data$Height)

## [1] 73.69729
```

Note: Multiplying the transpose of a matrix with another matrix is very common in statistical modeling and computing. Thus, there is an R function for this operation, `crossprod()`:

```
barY=crossprod(A, Y) / n
print(barY)

##          [,1]
## [1,] 73.69729
```

Variance

For the variance, we note that if:

$$\mathbf{Y}' \equiv \begin{pmatrix} Y_1 - \bar{Y} \\ \vdots \\ Y_n - \bar{Y} \end{pmatrix}, \frac{1}{n-1}\mathbf{Y}'^{\top}\mathbf{Y}' = \frac{1}{n-1}\sum_{i=1}^{n}\left(Y_i - \bar{Y}\right)^2.$$

A `crossprod` with only one matrix input computes: $Y'^{\top}Y'$ Thus, to compute the variance, we can simply type:

```
Y1 <- y - barY
crossprod(Y1)/(n-1)  # Y1.man <- (1/(n-1))* t(Y1) %*% Y1

##          [,1]
## [1,] 5.316798
```

Applications of Matrix Algebra: Linear Modeling

Let's use these matrices:

$$\mathbf{Y} = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix}, \mathbf{X} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ & \vdots \\ 1 & x_n \end{pmatrix}, \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} \text{ and } \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}.$$

Then, we can write a simple linear model:

$$Y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, i = 1, \ldots, n$$

as:

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

or simply:

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

which is a brief way to write the same model equation.

The optimal solution is achieved when all residuals (ϵ_i) are as small as possible (indicating a good model fit). This corresponds to the least squares (LS) solution to this matrix equation ($Y = X\beta + \epsilon$), which can be obtained by minimizing the residual square error:

$$< \epsilon^T, \epsilon >= (Y - X\beta)^T \times (Y - X\beta).$$

This can be achieved using the following cross-product:

$$\hat{\beta} = (\mathbf{Y} - \mathbf{X}\boldsymbol{\beta})^\top (\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}).$$

We can determine the values of β by minimizing this expression, using calculus to find the minimum of the cost (objective) function, more about optimization is in Chap. 22.

Finding Function Extrema (Min/Max) Using Calculus

There are a series of rules that permit us to solve partial derivative equations in matrix notation. By setting the derivative of a cost function to zero and solving for the unknown parameter β, we obtain a candidate solution(s). The derivative of the above equation is:

$$2\mathbf{X}^\top \left(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\beta}}\right) = 0$$
$$\mathbf{X}^\top \mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{Y}$$
$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{Y},$$

which represents the desired solution. Hat notation (^) is used to denote estimates. For instance, the solution for the unknown β parameters is denoted by the (data-driven) estimate $\hat{\beta}$.

The least squares minimization works because minimizing a function corresponds to finding the roots of its (first) derivative. In the ordinary least squares (OLS), we square the residuals:

$$(\mathbf{Y} - \mathbf{X}\boldsymbol{\beta})^{\top}(\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}).$$

Notice that the minima of $f(x)$ and $f^2(x)$ are achieved at the same roots of $f'(x)$, as the derivative of $f^2(x)$ is $2f(x)f'(x)$.

5.5.2 Least Square Estimation

```
#x=cbind(data$Height, data$Age)
x=data$Height
y=data$Weight
X <- cbind(1, x)
beta_hat <- solve( t(X) %*% X ) %*% t(X) %*% y
###or
beta_hat <- solve( crossprod(X) ) %*% crossprod( X, y )
```

Now we can see the results of this by computing the estimated $\hat{\beta}_0 + \hat{\beta}_1 x$ for any value of x (Fig. 5.1):

```
newx <- seq(min(x), max(x), len=100)
X <- cbind(1, newx)
fitted <- X%*%beta_hat
plot(x, y, xlab="MLB Player's Height", ylab="Playeer's Weight")
lines(newx, fitted, col=2)
```

$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^{\top}\mathbf{X}\right)^{-1}\mathbf{X}^{\top}\mathbf{Y}$ is one of the most widely used results in data analytics. One of the advantages of this approach is that we can use it in many different situations.

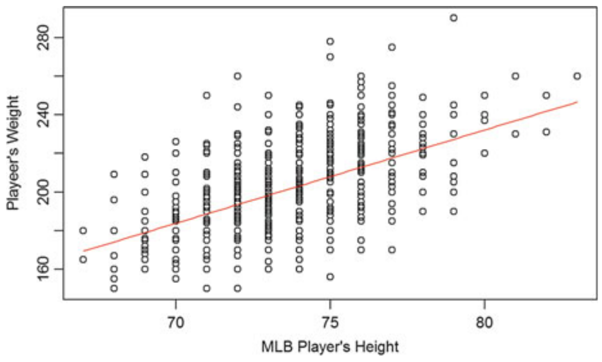

Fig. 5.1 A linear model of player's weight as a function of their height overlayed on the paired scatterplot

The R **lm** Function

R has a very convenient function that fits these models. We will learn more about this function later, but here is a preview:

```
# X <- cbind(data$Height, data$Age) # more complicated model
X <- data$Height    # simple model
y <- data$Weight
fit <- lm(y ~ X);
fit
```

Note that we obtain the same estimates of the solution using either the built-in lm() function or using first-principles.

5.6 Eigenvalues and Eigenvectors

Eigen-spectrum decomposition of linear operators (matrices) into *eigenvalues* and *eigenvectors* enables us to easily understand linear transformations. The eigenvectors represent the "axes" (directions) along which a linear transformation acts by *stretching, compressing*, or *flipping*. The eigenvalues represent the amounts of this linear transformation into the specified eigenvector direction. In higher dimensions, there are more directions along which we need to understand the behavior of the linear transformation. The eigen-spectrum makes it easier to understand the linear transformation, especially when many (perhaps all) of the eigenvectors are linearly independent (orthogonal).

For a matrix **A**, if we have $A\vec{v} = \lambda\vec{v}$ then we say $\vec{v}$ (a non-zero vector) is a right eigenvector of the matrix A, and the scale factor λ is the eigenvalue corresponding to that eigenvector.

With some calculations we know that $A\vec{v} = \lambda\vec{v}$ is the same as $(\lambda I_n - A)\vec{v} = \vec{0}$. Here I_n is the $n \times n$ identity matrix. So, when we solve this equation, we get our eigenvalues and eigenvectors. Of course, as this is a very common operation, we don't need to do that by hand – the `eigen()` function in R help us with this calculation.

```
m11<-diag(nrow = 2, ncol=2)
m11

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1

eigen(m11)

## $values
## [1] 1 1
##
## $vectors
##      [,1] [,2]
## [1,]    0   -1
## [2,]    1    0
```

We can use R to prove that $(\lambda I_n - A)\vec{v} = \vec{0}$.

```
(eigen(m11)$values*diag(2)-m11)%*%eigen(m11)$vectors
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

As we mentioned earlier, `diag(n)` creates an $n \times n$ identity matrix. Thus, `diag(2)` is the I_2 matrix in the equation. The output zero matrix proves that the equation $(\lambda I_n - A)\vec{v} = \vec{0}$ holds true.

Some of the many interesting applications of the eigen-spectrum are shown below.

5.7 Other Important Functions

Other useful matrix operation are listed in the following Table 5.2.

5.8 Matrix Notation (Another View)

Some flexible matrix operations can help us save time calculating row or column averages. For example, column averages can be calculated by the following matrix operation.

Table 5.2 Other matrix operators and operands

Functions	Math expression or explanation
`crossprod(A, B)`	$\boldsymbol{A}^T\boldsymbol{B}$ where $\boldsymbol{A}$, $\boldsymbol{B}$ are matrices
`y<-svd(A)`	The Singular Value Decomposition output has the following components
`-y$d`	Vector containing the singular values of A
`-y$u`	Matrix with columns contain the left singular vectors of A
`-y$v`	Matrix with columns contain the right singular vectors of A
`k <- qr(A)`	The output has the following components
`-k$qr`	Has an upper triangle that contains the decomposition and a lower triangle that contains information on the Q decomposition.
`-k$rank`	Is the rank of A
`-k$qraux`	A vector which contains additional information on Q
`-k$pivot`	Contains information on the pivoting strategy used.
`rowMeans(A)/colMeans(A)`	Returns vector of row/column means
`rowSums(A)/colSums(A)`	Returns vector of row/column sums

$$AX = \begin{pmatrix} \frac{1}{N} & \frac{1}{N} & \dots & \frac{1}{N} \end{pmatrix} \begin{pmatrix} X_{1,1} & \dots & & X_{1,p} \\ X_{2,1} & \dots & & X_{2,p} \\ \dots & \dots & \dots & \dots \\ X_{N,1} & \dots & & X_{N,p} \end{pmatrix} = \begin{pmatrix} \bar{X}_1 & \bar{X}_2 & \dots & \bar{X}_N \end{pmatrix}.$$

While row averages can be calculated by the next operation:

$$XB = \begin{pmatrix} X_{1,1} & \dots & & X_{1,p} \\ X_{2,1} & \dots & & X_{2,p} \\ \dots & \dots & \dots & \dots \\ X_{N,1} & \dots & & X_{N,p} \end{pmatrix} \begin{pmatrix} \frac{1}{p} \\ \frac{1}{p} \\ \dots \\ \frac{1}{p} \end{pmatrix} = \begin{pmatrix} \bar{X}_1 \\ \bar{X}_2 \\ \dots \\ \bar{X}_q \end{pmatrix}.$$

We see that fast calculations can be done by multiplying a matrix in the front or at the back of the original feature matrix. In general, multiplying a vector in front can give us the following equation.

$$AX = \begin{pmatrix} a_1 & a_2 & \dots & a_N \end{pmatrix} \begin{pmatrix} X_{1,1} & \dots & & X_{1,p} \\ X_{2,1} & \dots & & X_{2,p} \\ \dots & \dots & \dots & \dots \\ X_{N,1} & \dots & & X_{N,p} \end{pmatrix}$$
$$= \begin{pmatrix} \sum_{i=1}^{N} a_i \bar{X}_{i,1} & \sum_{i=1}^{N} a_i \bar{X}_{i,2} & \dots & \sum_{i=1}^{N} a_i \bar{X}_{i,N} \end{pmatrix}.$$

Now let's do an example to practice matrix notation. We will use genetic expression data including 8,793 different genes and 208 subjects. These gene expression data represents a microarray experiment—GSE5859—comparing Gene Expression Profiles from Lymphoblastoid cells. Specifically, the data compares the expression level of genes in lymphoblasts from individuals in three HapMap populations {CEU, CHB, JPT}. The study found that more than 1,000 genes were significantly different ($a < 0.05$) in mean expression level between the {CEU} and {CHB + JPT} samples.

The gene expression profiles data has two components:

- The gene expression intensities (exprs_GSE5859.csv) where the rows represent features on the microarray (e.g., genes), and columns represent different microarray samples,
- Meta-data about each of the samples (exprs_MetaData_GSE5859.csv) where rows represent samples, and columns represent meta-data (e.g., sex, age, treatment status, the date the sample processing).

```
gene<-read.csv("https://umich.instructure.com/files/2001417/download?downloa
d_frd=1", header = T) # exprs_GSE5859.csv
info<-read.csv("https://umich.instructure.com/files/2001418/download?downloa
d_frd=1", header=T)  # exprs_MetaData_GSE5859.csv
```

Like the `lapply()` function that we will discuss in Chap. 7, the `sapply()` function can be used to calculate column and row averages. Let's compare the output by using `sapply` for first-principles matrix calculations.

```
colmeans<-sapply(gene[, -1], mean)
gene1<-as.matrix(gene[, -1])
# can also use built in funcitons
# colMeans <- colMeans(gene1)
colmeans.matrix<-crossprod(rep(1/nrow(gene1), nrow(gene1)), gene1)
colmeans[1:15]

##  GSM25581.CEL.gz  GSM25681.CEL.gz GSM136524.CEL.gz GSM136707.CEL.gz
##         5.703998         5.721779         5.726300         5.743632
##  GSM25553.CEL.gz GSM136676.CEL.gz GSM136711.CEL.gz GSM136542.CEL.gz
##         5.835499         5.742565         5.751601         5.732211
## GSM136535.CEL.gz  GSM25399.CEL.gz  GSM25552.CEL.gz  GSM25542.CEL.gz
##         5.741741         5.618825         5.805147         5.733117
## GSM136544.CEL.gz  GSM25662.CEL.gz GSM136563.CEL.gz
##         5.733175         5.716855         5.750600

colmeans.matrix[1:15]

##  [1] 5.703998 5.721779 5.726300 5.743632 5.835499 5.742565 5.751601
##  [8] 5.732211 5.741741 5.618825 5.805147 5.733117 5.733175 5.716855
## [15] 5.750600
```

The same output is reported. Here, we use `rep(1/nrow(gene1), nrow(gene1))` to create the vector

$$\left(\frac{1}{N}\ \frac{1}{N}\ \cdots\ \frac{1}{N}\right)$$

needed to obtain the column averages. We may visualize the column means using a histogram (Fig. 5.2).

```
colmeans<-as.matrix(colmeans)
hist(colmeans)
```

The histogram shows that the distribution is mostly symmetric and bell shaped. We can address harder problems using matrix notation. For example, let's calculate the differences between genders for each gene. First, we need to get the gender information for each subject.

```
gender<-info[, c(3, 4)]
rownames(gender)<-gender$filename
```

Then, we have to reorder the columns to make then consistent with the feature matrix `gene1`.

```
gender<-gender[colnames(gene1), ]
```

After that, we will construct the design the matrix and multiply it by the feature matrix. The plan is to multiply the following two matrices.

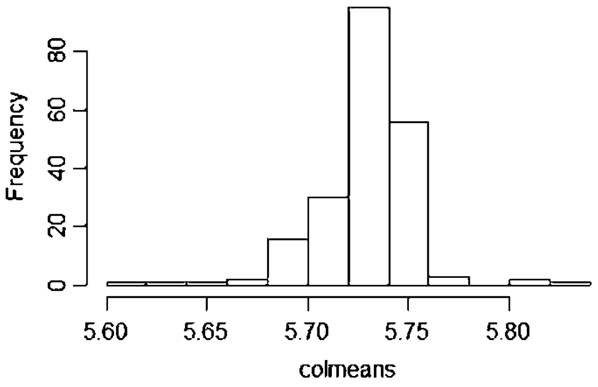

Fig. 5.2 Histogram of the column means of the gene expression data

$$\begin{pmatrix} X_{1,1} & \dots & & X_{1,p} \\ X_{2,1} & \dots & & X_{2,p} \\ \dots & \dots & \dots & \dots \\ X_{N,1} & \dots & & X_{N,p} \end{pmatrix} \begin{pmatrix} \frac{1}{p} & a_1 \\ \frac{1}{p} & a_2 \\ \dots & \dots \\ \frac{1}{p} & a_p \end{pmatrix} = \begin{pmatrix} \bar{X}_1 & gender.diff_1 \\ \bar{X}_2 & gender.diff_2 \\ \dots & \dots \\ \bar{X}_q & gender.diff_N \end{pmatrix}.$$

where $a_i = -1/N_F$ if the subject is female and $a_i = 1/N_M$ if the subject is male. Thus, we gave each female and male the same weight before the subtraction. We average each gender and get their difference. $\bar{X}_i$ represents the average across both genders and *gender. diff*$_i$ represents the gender difference for the *i*th gene.

```
table(gender$sex)

##
##   F   M
##  86 122

gender$vector<-ifelse(gender$sex=="F", -1/86, 1/122)
vec1<-as.matrix(data.frame(rowavg=rep(1/ncol(gene1), ncol(gene1)),
gender.diff=gender$vector))
gender.matrix<-gene1%*%vec1
gender.matrix[1:15, ]

##         rowavg  gender.diff
##  [1,] 6.383263 -0.003209464
##  [2,] 7.091630 -0.031320597
##  [3,] 5.477032  0.064806978
##  [4,] 7.584042 -0.001300152
##  [5,] 3.197687  0.015265502
##  [6,] 7.338204  0.078434938
##  [7,] 4.232132  0.008437864
##  [8,] 3.716460  0.018235650
##  [9,] 2.810554 -0.038698101
## [10,] 5.208787  0.020219666
## [11,] 6.498989  0.025979654
## [12,] 5.292992 -0.029988980
## [13,] 7.069081  0.038575442
## [14,] 5.952406  0.030352616
## [15,] 7.247116  0.046020066
```

5.9 Multivariate Linear Regression

As we mentioned earlier, the formula for multivariate linear regression can be written as

$$Y_i = \beta_0 + X_{i,1}\beta_1 + \cdots + X_{i,p}\beta_p + \epsilon_i, i = 1, \ldots, N.$$

We can rewrite this in a matrix form.

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \ldots \\ Y_N \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \ldots \\ 1 \end{pmatrix}\beta_0 + \begin{pmatrix} X_{1,1} \\ X_{2,1} \\ \ldots \\ X_{N,1} \end{pmatrix}\beta_1 + \ldots + \begin{pmatrix} X_{1,p} \\ X_{2,p} \\ \ldots \\ X_{N,p} \end{pmatrix}\beta_p + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \ldots \\ \epsilon_N \end{pmatrix}.$$

Which is the same as $Y = X\beta + \epsilon$ or

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \ldots \\ Y_N \end{pmatrix} = \begin{pmatrix} 1 & X_{1,1} & \ldots & X_{1,p} \\ 1 & X_{2,1} & \ldots & X_{2,p} \\ \ldots & \ldots & \ldots & \ldots \\ 1 & X_{N,1} & \ldots & X_{N,p} \end{pmatrix}\begin{pmatrix} \beta_o \\ \beta_1 \\ \ldots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \ldots \\ \epsilon_N \end{pmatrix}.$$

$Y = X\beta + \epsilon$ implies that $X^T Y \sim X^T(X\beta) = (X^T X)\beta$, and thus the solution for β is obtained by multiplying both hand sides by $(X^T X)^{-1}$:

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T Y.$$

Matrix calculation would be faster than fitting a regression model. Let's apply this to the Lahman baseball data representing yearly stats and standings. Let's download the baseball.data (https://umich.instructure.com/files/2018445/download?download_frd=1) and put it in the R working directory. We can use the `load()` function to load a local RData object. For this example, we subset the dataset by `G==162` and `yearID<2002`. Also, we create a new feature named `Singles` that is equal to `H(Hits by batters) - X2B(Doubles) - X3B(Tripples) - HR (Homeruns by batters)`. Finally, we only pick some features: `R` (Runs scored), `Singles`, `HR` (Homeruns by batters) and `BB` (Walks by batters).

```r
#If you downloaded the .RData locally first, then you can easily load it
into the R workspace by:
# load("Teams.RData")

# Alternatively you can also download the data in CSV format from
http://umich.instructure.com/courses/38100/files/folder/data (teamsData.csv)
Teams <- read.csv('https://umich.instructure.com/files/2798317/download?down
load_frd=1', header=T)

dat<-Teams[Teams$G==162&Teams$yearID<2002, ]
dat$Singles<-dat$H-dat$X2B-dat$X3B-dat$HR
dat<-dat[, c("R", "Singles", "HR", "BB")]
head(dat)
```

```
##        R Singles  HR  BB
## 439  505     997  11 344
## 1367 683     989  90 580
## 1368 744     902 189 681
## 1378 652     948 156 516
## 1380 707    1017  92 620
## 1381 632    1020 126 504
```

Now let's do a simple example. We will use runs scored (R) as the response variable and batters walks (BB) as the independent variable. Also, we need to add a column of 1's to the X matrix.

```
Y<-dat$R
X<-cbind(rep(1, n=nrow(dat)), dat$BB)
X[1:10, ]
```

```
##       [,1] [,2]
##  [1,]    1  344
##  [2,]    1  580
##  [3,]    1  681
##  [4,]    1  516
##  [5,]    1  620
##  [6,]    1  504
##  [7,]    1  498
##  [8,]    1  502
##  [9,]    1  493
## [10,]    1  556
```

Let's solve for the effect-sizes (the beta coefficients) by

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T Y.$$

```
beta<-solve(t(X)%*%X)%*%t(X)%*%Y
beta
```

```
##             [,1]
## [1,] 326.8241628
## [2,]   0.7126402
```

To examine this manual calculation, we refit the linear equation using the `lm()` function. After comparing the time used for computations, we may notice that matrix calculation are more time efficient.

```
fit<-lm(R~BB, data=dat)
# fit<-lm(R~., data=dat)
# '.' indicates all other variables, very useful when fitting models
with many predictors
fit
```

```
##
## Call:
## lm(formula = R ~ BB, data = dat)
##
```

```
## Coefficients:
## (Intercept)           BB
##    326.8242       0.7126

summary(fit)

## Call:
## lm(formula = R ~ BB, data = dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -187.788  -53.977   -2.995   55.649  258.614
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 326.82416   22.44340   14.56   <2e-16 ***
## BB            0.71264    0.04157   17.14   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 76.95 on 661 degrees of freedom
## Multiple R-squared:  0.3078, Adjusted R-squared:  0.3068
## F-statistic:   294 on 1 and 661 DF,  p-value: < 2.2e-16

system.time(fit<-lm(R~BB, data=dat))

##    user  system elapsed
##       0       0       0

system.time(beta<-solve(t(X)%*%X)%*%t(X)%*%Y)

##    user  system elapsed
##       0       0       0
```

We can visualize the relationship between `R` and `BB` by drawing a scatter plot (Fig. 5.3).

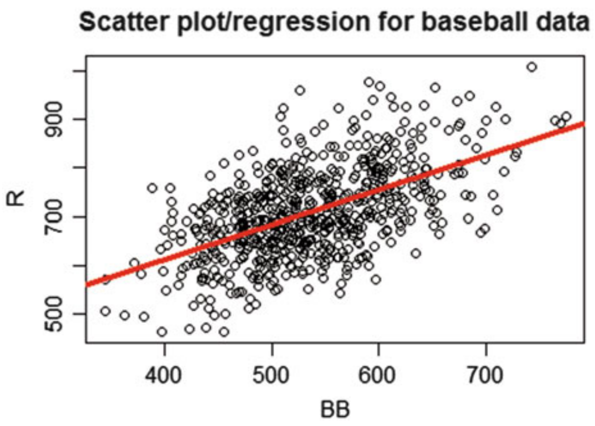

Fig. 5.3 Scatterplot and model of walks (BB) and runs (R) by batters, using the MLB dataset

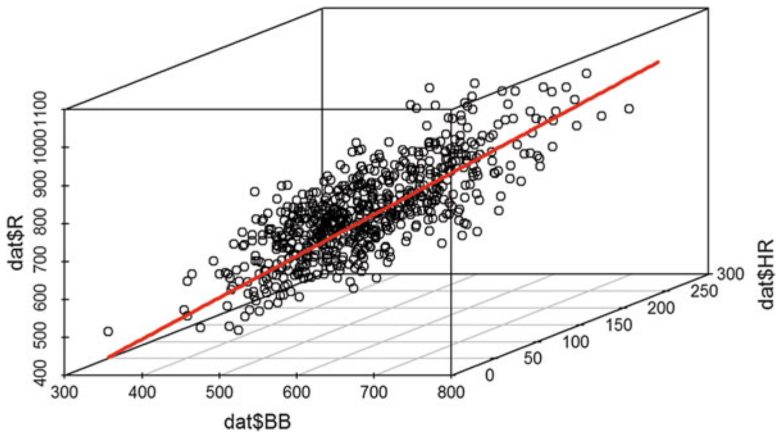

Fig. 5.4 3D Scatterplot of walks (BB), homeruns (HR), and runs (R) by batters, using the baseball dataset

```
plot(dat$BB, dat$R, xlab = "BB", ylab = "R", main = "Scatter plot/regression
for baseball data")
abline(beta[1, 1], beta[2, 1], lwd=4, col="red")
```

On Fig. 5.3, the red line is our regression line calculated by matrix calculation. Matrix calculation can still work if we have multiple independent variables. Next, we will add another variable, HR, to the model, Fig. 5.4.

```
X<-cbind(rep(1, n=nrow(dat)), dat$BB, dat$HR)
beta<-solve(t(X)%*%X)%*%t(X)%*%Y
beta

##             [,1]
## [1,] 287.7226756
## [2,]   0.3897178
## [3,]   1.5220448

#install.packages("scatterplot3d")
library(scatterplot3d)
scatterplot3d(dat$BB, dat$HR, dat$R)
```

5.10 Sample Covariance Matrix

We can also obtain the covariance matrix for our features using matrix operations. Suppose

$$X_{N\times K} = \begin{pmatrix} X_{1,1} & \dots & X_{1,K} \\ X_{2,1} & \dots & X_{2,K} \\ \dots & \dots & \dots \\ X_{N,1} & \dots & X_{N,K} \end{pmatrix} = [X_1, X_2, \dots, X_N]^T.$$

Then the covariance matrix is:

$$\Sigma = \left(\Sigma_{i,j}\right),$$

where $\Sigma_{i,\,j} = Cov(X_i, X_j) = E((X_i - \mu_i)(X_j - \mu_j))$, $1 \leq i, j, \leq N$.

The sample covariance matrix is:

$$\Sigma_{i,j} = \frac{1}{N-1}\sum_{m=1}^{N}\left(x_{m,\,i} - \bar{x}_i\right)\left(x_{m,j} - \bar{x}_j\right),$$

where

$$\bar{x}_i = \frac{1}{N}\sum_{m=1}^{N} x_{m,\,i}, \quad i = 1, \ldots, K.$$

In general,

$$\Sigma = \frac{1}{n-1}\left(X - \bar{X}\right)^T\left(X - \bar{X}\right).$$

Assume we want to get the sample covariance matrix of the following 5 $\times$ 3 feature matrix x.

```
x<-matrix(c(4.0, 4.2, 3.9, 4.3, 4.1, 2.0, 2.1, 2.0, 2.1, 2.2, 0.60, 0.59,
0.58, 0.62, 0.63), ncol=3)
x

##      [,1] [,2] [,3]
## [1,]  4.0  2.0 0.60
## [2,]  4.2  2.1 0.59
## [3,]  3.9  2.0 0.58
## [4,]  4.3  2.1 0.62
## [5,]  4.1  2.2 0.63
```

Notice that we have *three* features and *five* observations in this matrix. Let's get the column means first.

```
vec2<-matrix(c(1/5, 1/5, 1/5, 1/5, 1/5), ncol=5)
#column means
x.bar<-vec2%*%x
x.bar

##      [,1] [,2]  [,3]
## [1,]  4.1 2.08 0.604
```

Then, we repeat each column mean *5* times to match the layout of feature matrix. Finally, we are able to plug everything in the formula above.

```
x.bar<-matrix(rep(x.bar, each=5), nrow=5)
S<-1/4*t(x-x.bar)%*%(x-x.bar)
S

##         [,1]    [,2]    [,3]
## [1,] 0.02500 0.00750 0.00175
## [2,] 0.00750 0.00700 0.00135
## [3,] 0.00175 0.00135 0.00043
```

In the covariance matrix, `S[i, i]` is the variance of the *i*th feature and `S[i, j]` is the covariance of *i*th and *j*th features.

Compare this to the automated calculation of the variance-covariance matrix.

```
autoCov <- cov(x)
autoCov

##         [,1]    [,2]    [,3]
## [1,] 0.02500 0.00750 0.00175
## [2,] 0.00750 0.00700 0.00135
## [3,] 0.00175 0.00135 0.00043
```

5.11 Assignments: 5. Linear Algebra & Matrix Computing

5.11.1 *How Is Matrix Multiplication Defined?*

Validate that $(A_{k,n} \times B_{n,m})^T = \left(B^T_{m,n}\right) \times \left(A^T_{n,k}\right)$, by using math notation, as well as by using `R` functions.

5.11.2 *Scalar Versus Matrix Multiplication*

Demonstrate the differences between the scalar multiplication ($*$) and matrix multiplication ($\% * \%$) for numbers, vectors, and matrices (second-order tensors).

5.11.3 *Matrix Equations*

Write a simple matrix solver ($b = Ax$, i.e., $x = A^{-1}b$) and validate its acuracy using the R command `solve(A,b)`. Solve this equation:

$$\begin{aligned} 2a - b + 2c &= 5 \\ -a - 2b + c &= 3\,. \\ a + b - c &= 2 \end{aligned}$$

5.11.4 Least Square Estimation

Use the SOCR Knee Pain dataset, extract the `RB` = `Right-Back` locations (x, y), and fit in a linear model for vertical locations (y) in terms of the horizontal locations (x). Display the linear model on top of the scatter plot of the paired data. Comment on the model you obtain.

5.11.5 Matrix Manipulation

Create a matrix A with elements `seq(1, 15, length = 6)` and argument `nrow = 3`. Then, add a row to this matrix and add two columns to A to obtain a matrix $C_{4,\ 4}$. Next, generate a diagonal matrix D with $dim = 4$ and elements `rnorm (4,0,1)`. Apply elementwise addition, subtraction, multiplication, and division to the matrix C and D; apply matrix multiplication to D and C; obtain the inverse of the C and compare it with the generalized inverse, `MASS::ginv()`.

5.11.6 Matrix Transpose

Validate the multiplication transposition formula, $(A_{k,n} \cdot B_{n,m})^T = B_{n,m}^T \cdot A_{k,n}^T$, by using math notation, as well as computationally using R and some example matrices. E.g. you can try

```
A = matrix(1:6,nrow=3); B = matrix(2:7, nrow = 2)
```

5.11.7 Sample Statistics

Use the SOCR Data Iris Sepal Petal Classes and extract the rows of `setosa` flowers. Compute the sample mean and variance of each variables; then calculate sample covariance and correlation between sepal width and sepal height.

5.11.8 Least Square Estimation

Use the SOCR Knee Pain dataset, extract the `RB` = `Right-Back` locations (x, y), and fit in a linear model for vertical location (y) in terms of the horizontal location (x). Display the linear model on top of the scatter plot of the paired data. Comment on the model you obtained.

5.11.9 Eigenvalues and Eigenvectors

Generate a random matrix with `A = matrix(rnorm(9,0,1),nrow = 3)`, compute eigenvalues and eigenvectors for A; then try to solve this equation *det* $(A - \lambda I) = 0$, where λ is a vector of length 3. Compare λ and the eigenvalues you solved above.

Example of manual and automated calculations of eigen-spectra (eigenvalues and eigenvectors):

```
# A <- matrix(rnorm(9,0,1),nrow = 3); A

# define a random design matrix, may generate complex solutions
A <- matrix(c(0,1/4,1/4,3/4,0,1/4,1/4,3/4,1/2),3,3,byrow=T); A
eigen_spectrum <- eigen(A); eigen_spectrum

# compute the eigen spectrum (eigen-values, $l$, and eigen-vectors, $v$),

# $ A \times v = l \times v$.
B <- A-eigen(A)$values*diag(3); B

# compute B = (A - eigen_value \times I)
det(A-eigen(A)\$values*diag(3))

# verrify that the det(A-eigen(A)\$values*diag(3)) is not trivial ($0$)
A%*%eigen(A)$vector - eigen(A)$value*diag(3)

# validate that $ A \times v = l \times v$.
all.equal(A,  eigen(A)\$vector %*% diag(eigen(A)\$values) %*%
solve(eigen(A)$vector))  # compare A = v*l*inv(v)
all.equal(diag(3),  A%*%eigen(A)$vector - eigen(A)$values * eigen(A)$vector)
# The last line compares I == AV - lambda\*v, mind the $*$ and

# $%*%$ scalar and matrix operators
```

References

http://www.statmethods.net/advstats/matrix.html

Vinod, Hrishikesh D. (2011) *Hands-On Matrix Algebra Using R: Active and Motivated Learning with Applications*, World Scientific Publishing, ISBN 981310080X, 9789813100800.

Gentle, James E. (2007) *Matrix Algebra: Theory, Computations, and Applications in Statistics*, Springer, ISBN 0387708723, 9780387708720.