Intro to Matrix Algebra

Gaston Sanchez

October 28, 2016

Matrix Algebra

Matrix algebra is fundamental for a good understanding of Multivariate Data Analysis methods, Statistical Learning methods, Machine Learning methods, as well as Data Mining techniques (keep in mind that there is a considerable amount of overlap in all these fields).

- Multivariate data is commonly represented in **tabular format** (rows and columns)
- Mathematically, a data table can be treated as a matrix
- Matrix algebra provides the analytical machinery and tools to manipulate and exploit values, information, and patterns of variability in data

The heart of any statitical mulativariate technique consists of a data matrix, or in some cases, matrices. The data matrix is a rectangular array of "numerical" entries whose informational content is to be summarized and displayed in some way.

Much of the multivariate analysis is concerned with studying certain aspects of the **association** among variables. The difference between the methods relies on how the analyst define "association", the number of variables, their type (e.g. quantitative, qualitative), as well as their measurement scales (e.g. raio-scaled, interval, ordinal, nominal, binary). Likewise, some methods focus not in the association between variables but in the association between individuals; in this case the idea is to study the ressemblance among inviduals using some type of similarity measure.

Preliminaries

Multivariate techniques start from a multivariate data matrix. The most common form for this matrix is a table that gives the results of a number of observations on a number of variables simultaneously. This is not the only way in which data matrices are found, but it is very common. We follow the standard convention that columns refer to variables, and rows to a set of observations. A more restrictive view of this format is the so-called *tidy* data:

- each variable forms a column
- each observation forms a row
- each type of observational unit forms a table

http://vita.had.co.nz/papers/tidy-data.pdf

Here's a list of assumptions and considerations for the topics and material covered in this unit (and other related units).

1. The point of departure is always a multivariate data matrix with a certain number, n, of rows for the individual observation units, and a certain number, p, of columns for the variables. This is a standard assumption, but it is possible to find matrices that don't represent observations-in-rows and varaibles-in-columns.

2. In most applications of multivariate analysis, we won't really be interested in variable means. They have their interest in each study, but multivariate analysis instead focuses on variances and covariances. Therefore, the data matrix will in general be transformed into a matrix where columns have zero means and where the numbers in the column represent *deviations from the mean.*

3. Such a matrix is the basis for the *variance-covariance matrix* (a.k.a. the covariance matrix) with p rows and p columns.

4. An often useful transformation is to *standardize* the data matrix: we first take deviations from the mean for each column, then divide the deviation from the mean by the standard deviation for the same column. The result is that values in a column will have zero mean and unit-variance.

5. The standardized data matrix is then the basis for calculating a *correlation matrix*, which is nothing but a variance-covariance matrix for standardized variables. In the diagonal of this matrix we find values equal to unity. In the off-diagonal cells we find correlations.

6. Very often we will need a variable that is a *linear combination* of the initial variables. The linear compound is simply a new variable whose values are obtained by a weighted sum of values of the original variables.

7. For some techniques of multivariate analysis, we need to be able to solve simultaneous equations. Doing so usually requires a computational routine called *matrix inversion.*

8. Multivariate analysis nearly always comes down to finding a *minimum* or a *maximum* of some sort. A typical example is to find a linear combination of some variables that has maximum correlation with some other variable (multiple correlation), or to find a linear combination of the observed scores that has maximum variance.

9. In addition, we will often need to find maxima (or minima) of functions where the procedure is limited by certain *side-conditions.* For instance, we are given two sets of variables, and are required to find a linear combination from the first set, and another from the second set, such that the value of the correlation between these two combinations is maximum.

10. Very often, a maximization procedure under certain side-conditions boils down to finding *eigenvectors* and *eigenvalues* of a given matrix.

Notation

I'll try to use bold capital letters for matrices, $\mathbf{A}$, and bold lower-case letters for vectors $\mathbf{x}$. Italics are used to represent scalars. Then, x_i is the i-th element of $\mathbf{x}$, A_{ij} is the element in the i-th row and j-th column of $\mathbf{A}$. By default, we'll consider a vector $\mathbf{x}$ to be a one-column matrix, and $\mathbf{x}^T$ to be a one-row matrix.

The **inner product** of two vectors $\mathbf{x}$ and $\mathbf{y}$ is:

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T\mathbf{y} = \sum_i x_i y_i$$

The **outer product** of two vectors $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{x}\mathbf{y}^T$ which comes from all pairwise products of the elements.

Matrices in R

In mathematics, a matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. In R, a matrix is a two-dimensional `array`: a two-dimensional collection of values of the same type (i.e. atomic). R stores matrices (and arrays in general) as vectors, therefore matrices are atomic structures. We can have `numeric`, `integer`, `character`, `logical`, or `complex` matrices. The main difference between an R `matrix` and a `vector` is that a vector does not have attribute `dim` (dimensions).

Matrices in R are stored column-major (i.e. by columns). This is like Fortran, Matlab, and Julia, but not like C or Python (e.g. numpy).

```r
# matrix (stored column-major)
M <- matrix(1:12, nrow = 4, ncol = 3)
M
```

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

The function `matrix()` takes a vector as input and produces an object of class `"matrix"`. Internally, an R matrix is a vector with `dim` attribute, and potentially, with `dimnames` attribute.

Basic functions in R for matrix objects

Functions	Description
`matrix()`	create a matrix
`dim()`	dimensions of a matrix
`nrow()`	number of rows
`ncol()`	number of columns
`as.matrix()`	converti into matrix
`is.matrix()`	test if object is a mtrix

Keep in mind that R can do some things that matrix algebra cannot: row-column naming, handling `NA`'s, and recycling.

```r
# matrix
A <- matrix(1:12, nrow = 4, ncol = 3)

# add row and column names
rownames(A) <- c("a", "b", "c", "d")
colnames(A) <- c("one", "two", "three")

A
```

```
##   one two three
## a   1   5     9
## b   2   6    10
## c   3   7    11
## d   4   8    12
```

In matrix algebra we refer to the elements of a matrix **A** by

$$a_{ij} = \{A\}_{ij}$$

In R, we use the bracekt notation, e.g. `A[i,j]`, to access and retrieve elements of a matrix. Because matrices are two-dimensional objects, inside the brackets we must specify two vectors separated by a comma. The first vector corresponds to the rows, and the second vector corresponds to the columns.

```
# element 1,1
A[1,1]
```

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

```
# first row
A[1, ]
```

```
##   one   two three
##     1     5     9
```

```
# first column
A[ ,1]
```

```
## a b c d
## 1 2 3 4
```

Notice that when you extract one column, e.g. `A[ ,2]`, by default the output is an R vector. To extract one column and keeping the output as a matrix we use the argument `drop = FALSE` inside the brackets:

```
# second column as a matrix
A[ , 2, drop = FALSE]
```

```
##   two
## a   5
## b   6
## c   7
## d   8
```

The dimension attributes (i.e. number of rows and columns) can be accessed with `dim()`, `nrow()`, and `ncol()`:

```
# dimensions
dim(A)
```

```
## [1] 4 3
```

```
# add row names
nrow(A)
```

```
## [1] 4
```

```
# add column names
ncol(A)
```

```
## [1] 3
```

As you know, R allows you to recycle vectors and handle missing values:

```
# vector
b <- 1:3

# recycling a vector into a matrix
B <- matrix(b, 4, 3, byrow=TRUE)
rownames(B) <- c('a', 'b', 'c', 'd')
colnames(B) <- c('Luke', 'Han', 'Leia')

B[6] <- NA

B
```

```
##   Luke Han Leia
## a    1   2    3
## b    1  NA    3
## c    1   2    3
## d    1   2    3
```

Matrices -vs- Vectors in R

It is important to distinguish vectors and matrices, especially in R:

- In matrix algebra we use the convention that vectors are column vectors (i.e. they are $n \times 1$ matrices).
- In R, a vector with n elements is not the same as an $n \times 1$ matrix, because an R matrix has the dimensions attribute, and an R vector does not.
- Vectors in R behave more like row vectors.
- However, depending on the type of functions you apply to vectors, sometimes R will handle vectors like if they were column vectors.
- Also, numbers in R are actually vectors with a single element.

From scalar to matrix

```
# scalar
x <- 1
```

```r
# dim
dim(x)
```

```
## NULL
```

```r
xx <- matrix(x, 1, 1)
xx
```

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

Matrices -vs- Dataframes in R

It is also important to distinguish an R `matrix` from an R `data.frame`. Both objects allows us to store data in a 2-dimensional object. In many cases, both R matrices and data.frames have similar behaviors. This is mostly the case when they are displayed on the screen. And in some cases it is hard to distinguish between a matrix and a data.frame.

Keep in mind that data.frames are actually lists. Internally, a data.frame is stored as an R list (typically a list of vectors). In other words,what you see as columns in a data.frame are actually vectors. Each column represents an element of a list. This provides a very flexible way to manipulate data.frames because you can handle them as list with `$` and `[[]]` operators. But you can also handle them as an array with the operator `[ , ]`.

An R matrix is internally stored as a vector. This is why matrices are atomic. The added attribute of a matrix is `dim`. Which means that you use the `[ , ]` operator to manipulate elements of a matrix.

Both matrices and data.frames have common methods:

- `[ , ]`
- `dim()`
- `nrow()`
- `ncol()`
- `colnames()`
- `apply()`

Knowing the differences between a `matrix` and a `data.frame` should help you avoid many common errors, and hopefully, save you from many headaches and frustrating moments.

Basic Matrix Operations

For ease of reference we first quickly go through the basic matrix operations in R. These basic operations are:

- transpose
- addition
- scalar multiplication
- matrix-vector multiplication

- matrix-matrix multiplication

Matrix Transpose: The transpose of a $n \times p$ matrix $\mathbf{X}$ is the $p \times n$ matrix $\mathbf{X}^T$. In R the transpose is given by the function `t()`

```r
# matrix X
X <- matrix(1:6, 2, 3)
X
```

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

```r
# transpose of X
t(X)
```

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

Matrix Addition: Matrix addition of two matrices $\mathbf{A} + \mathbf{B}$ is defined when $\mathbf{A}$ and $\mathbf{B}$ have the same dimensions:

```r
A <- matrix(1:6, 2, 3)
B <- matrix(7:9, 2, 3)
A + B
```

```
##      [,1] [,2] [,3]
## [1,]    8   12   13
## [2,]   10   11   15
```

Scalar Multiplication: We can multiply a matrix by a scalar using the usual product operator `*`, moreover it doesn't matter if we pre-multiply or post-multiply:

```r
X <- matrix(1:3, 3, 4)

# (pre)multiply X by 0.5
(1/2) * X
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.5  0.5  0.5  0.5
## [2,]  1.0  1.0  1.0  1.0
## [3,]  1.5  1.5  1.5  1.5
```

You can also postmultiply by a scalar (although this is not recommended because may confuse readers):

```r
X <- matrix(1:3, 3, 4)

# (post)multiply X by 0.5
X * (1/2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.5  0.5  0.5  0.5
## [2,]  1.0  1.0  1.0  1.0
## [3,]  1.5  1.5  1.5  1.5
```

Matrix-Matrix Multiplication: The matrix product operator in R is `%*%`. We can multiply matrices $\mathbf{A}$ and $\mathbf{B}$ if the number of columns of $\mathbf{A}$ is equal to the number of rows of $\mathbf{B}$

```r
A <- matrix(1:6, 2, 3)
B <- matrix(7:9, 3, 2)

A %*% B
```

```
##      [,1] [,2]
## [1,]   76   76
## [2,]  100  100
```

We can multiply matrices $\mathbf{A}$ and $\mathbf{B}$ if the number of columns of $\mathbf{A}$ is equal to the number of rows of $\mathbf{B}$

```r
A <- matrix(1:6, 2, 3)
B <- matrix(7:9, 3, 2)

B %*% A
```

```
##      [,1] [,2] [,3]
## [1,]   21   49   77
## [2,]   24   56   88
## [3,]   27   63   99
```

Cross-Products: A very common type of products in multivariate data analysis are $\mathbf{X}'\mathbf{X}$ and $\mathbf{X}\mathbf{X}'$, sometimes known as cross-products:

```r
# cross-product
t(A) %*% A
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```r
# cross-product
A %*% t(A)
```

```
##      [,1] [,2]
## [1,]   35   44
## [2,]   44   56
```

R provides functions `crossprod()` and `tcrossprod()` which are formally equivalent to:

- `crossprod(X, X)` $\equiv$ `t(X) %*% X`
- `tcrossprod(X, X)` $\equiv$ `X %*% t(X)`

However, `crossprod()` and `tcrossprod()` are usually faster than using `t()` and `%*%`

Vector-Matrix Multiplication: We can post-multiply an $n \times p$ matrix $\mathbf{X}$ with a vector $\mathbf{b}$ with p elements. This means making linear combinations (weighted sums) of the columns of $\mathbf{X}$:

```r
X <- matrix(1:12, 3, 4)
b <- seq(0.25, 1, by = 0.25)
X %*% b
```

```
##      [,1]
## [1,] 17.5
## [2,] 20.0
## [3,] 22.5
```

We can pre-multiply a vector $\mathbf{a}$ (with n elements) with an $n \times p$ matrix $\mathbf{X}$. This means making linear combinations (weighted sums) of the rows of $\mathbf{X}$:

```r
X <- matrix(1:12, 3, 4)
a <- 1:3
a %*% X
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   14   32   50   68
```

Notice that when we use the product operator `%*%`, R is smart enough to use the convention that vectors are $n \times 1$ matrices.

Notice also that if we ask for a vector-matrix multiplication, we can use both formulas:

- `a %*% X`
- `t(a) %*% X`

R will reformat the n vector as an $n \times 1$ matrix first.

Operations with Matrix Notation

The syntax and functions that R offers to handle matrix operations allow you to write code that is close to the formulas that use algebra notation. You should learn how to translate those formulas in R. Often, this is how you will typically start implementing a method or an algorithm that involves matrix algebra manipulations. Once you've written code that gets the job done, then you can start working out a solution that uses vectorized code, or write commands that are more efficient and/or compact.

In the following subsections we are going to see how to translate common matrix operations with R functions for matrices.

Sum of elements of a vector

The sum of all the values in an n-element vector $\mathbf{x}$ can be expressed in matrix algebra notation as:

$$\mathbf{x}^T \mathbf{1} = \sum_{i=1}^{n} x_i$$

where $\mathbf{1}$ is a vector of ones with the same length of $\mathbf{x}$

Using matrix algebra operations, the sum of elements of a vector can be expressed in R as:

```
x <- 1:4
ones <- rep(1, 4)

# scalar product
t(x) %*% ones
```

```
##      [,1]
## [1,]   10
```

and of course, you can also use vectorized code:

```
# vectorized sum
sum(x)
```

```
## [1] 10
```

Mean of elements of a vector

The mean value of the n-elements in a vector $\mathbf{x}$ can be expressed in matrix algebra notation as:

$$(1/n)\mathbf{x}^T \mathbf{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Using R matrix functions:

```
x <- 1:4
ones <- rep(1, 4)

# scalar product
(1/4) * t(x) %*% ones
```

```
##      [,1]
## [1,]  2.5
```

```
# vectorized mean
mean(x)
```

```
## [1] 2.5
```

Sum of squares of elements of a vector

The sum of squares of all the values in an n-element vector $\mathbf{x}$ can be expressed in matrix algebra notation as:

$$\mathbf{x}^T\mathbf{x} = \sum_{i=1}^{n} x_i^2$$

Using matrix functions in R, we have:

```r
x <- 1:4

# scalar product
t(x) %*% x
```

```
##      [,1]
## [1,]   30
```

```r
# vectorized sum of squares
sum(x^2)
```

```
## [1] 30
```

Trace of a square matrix

The trace of a $n \times n$ square matrix $\mathbf{A}$ is defined as:

$$tr(\mathbf{A}) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}$$

In R, you can calculate the trace as:

```r
A <- matrix(1:9, nrow = 3, ncol = 3)

# trace
sum(diag(A))
```

```
## [1] 15
```

An Overview of Multivariate Techniques

Multivariate analysis comes down to operations on columns (sometimes on rows, too) of the data matrix. What operations have to be performed depends on the specific model that inspires the analysis. Therefore, there is no cookbook in which one can look up the appropriate techniques for a given type of matrix. The analysis does not depend so much on the nature of the matrix as on the nature of the specific questions asked about the variables and their interrelations.

Measurement Scales, Transformation and Standardization

Variables can be classified into various types according to their scales or measurement. Categorical data can be measured on a *nominal* or *ordinal* scale. Nominal categories havo no ordering: for example, flavor (vanilla, chocolate, lemon). Ordinal categories do have an ordering (never, rarely, often, always). Continuous data can be measured on a *ratio* or *interval* scale. Count data have a special place as they can be considered both ordinal and ratio.

There is also compositional data: these are proportions that add up to 1. Compositional data are usually created from a set of counts or a set of ratio variables when their total is not as relevant as the composition formed by the parts. For example, when we count different species sampled at a particular site, it is likely that the total number is not so relevant, but rather the proportion that each species contributes to the overall count.

1. Variables can be either categorical or continuous, although all measurements are categorical in the sense of being discretized. Continuous variables are those that have very many categories, for example a count variable, or are discretized versions of a variable which could, at least theoretically, be measured on a continuous scale, for example a length or a concentration.
2. Categorical variables can be either ordinal or nominal, depending on whether the categories have an inherent ordering or not.
3. Continuous variables can be either ratio or interval, depending on whether we compare two observations on that variable multiplicatively (as a ratio) or additively (as a difference).
4. The logarithmic transformation is a very useful transformation for most positive ratio measurements, because multiplicative comparisons are converted to additive ones and because high values are pulled in, making the distribution of the variable more symmetric.
5. Categorical variables are usually coded as dummy variables in order to be able to judge the effect or relationship of individual categories.
6. Continuous variables can also be dummy coded but this loses a lot of information. A better option is to fuzzy code them into a small number of categories, which allows continuous variables to be analysed together with categorical ones more easily, especially in the case of structural multivariate methods.
7. In many multivariate methods, standardization is a major issue for consideration. Variances of the variables being analyzed need to be balanced in some way that gives each variable a fair chance of being involved in the determination of the latent structure. Results should not depend on the scale of measurement.

Converting Categorical Variables into Dummy Variables

R provides objects of class `"factor"` to handle categorical/qualitative data.

Many multivariate methods require transforming a categorical variable into binary indicators. For instance, say we have a factor `a`:

```
## [1] A B A B B
## Levels: A B
```

a can be “dummified” into a matrix with two binary indicators:

```
##      A B
## [1,] 1 0
## [2,] 0 1
## [3,] 1 0
## [4,] 0 1
## [5,] 0 1
```

It is common to form cross-tables from two factors:

```
g <- sample(1:3, size = 50, replace = TRUE)
h <- sample(letters[1:4], size = 50, replace = TRUE)
table(h, g)
```

```
##    g
## h   1 2 3
##   a 3 4 2
##   b 6 6 4
##   c 3 4 4
##   d 3 6 5
```

Your Turn: R provides the function `table()` to calculate simple cross-tables. But from the data-manipulation point of view, it is interesting to write your own code and function(s) which take two factors, convert them in a matrix of binary indicators, and then compute the cross-table with a matrix cross-product.

Standardization

Variables on different scales have natural variances which depend mostly on their scales. The fact that some variables can have high variances just because of the chosen scale of measurement causes problems when we look for structure amongst the variables. The variables with high variance will dominate our search because they appear to contain more information, while those with low variance are swamped because of their small differences between values.

The answer is clearly to balance out the variances so that each variable can play an equal role in our analysis—this is exactly what standardization tries to achieve. The simplest form of standardization is to make all variances in the data set exactly the same. For a bunch of continuous variables, for example, we would divide the values of each variable by its corresponding sample standard deviation so that each variable has variance (and also standard deviation) equal to 1. Often this is accompanied by centering the variable as well, that is, subtracting its mean, in which case we often refer to the standardized variable as a Z-score. This terminology originates in the standardization of a normally distributed variable X, which after subtracting its mean and dividing by its standard deviation is customarily denoted by the letter Z and called a standard normal variable, with mean 0 and variance 1.

Standardization can also be thought of as a form of weighting. That is, by dividing variables with large variances by their large standard deviations, we are actually multiplying them by small numbers and reducing their weight. The variables with small variances, on the other hand, are divided by smaller standard deviations and thus have their weight increased relative to the others.

Other forms of standardization are:

- by the range: each variable is linearly transformed to lie between 0 and 1, where 0 is its minimum and 1 its maximum value;
- by chosen percentiles: because the range is sensitive to outliers, we can "peg" the 0 and 1 values of the linearly transformed variable to, say, the 5th and 95th percentile of the sample distribution;
- by the mean: the values of a variable are divided by their mean, so that they have standard deviations equal to what is called their *coefficient of variation*.

When handling continuous and categorical variables jointly, where the continuous variables have been coded into fuzzy dummy variables and the categorical variables into (zero-one) dummies, we could standardize by calculating the collective variance of each set of dummies corresponding to one variable and then weighting the set accordingly. That is, we do not standardize individual dummy variables, which would be incorrect, but each group as a whole.

These are note the only possible standardizations. Certain techniques require a specific type of scaling. Also, some disciplines of research fields prefer a certain type of ad-hoc standardization.

Standardization in R

To standardize variables there are different functions:

- `scale()`
- `sweep()`
- some matrix multiplication

The most common function to mean-center and standardize variables is `scale()`

```
# some matrix
A <- matrix(runif(30), nrow = 10, ncol = 3)
apply(A, 2, mean)
```

```
## [1] 0.5007173 0.5033785 0.6010724
```

```
apply(A, 2, sd)
```

```
## [1] 0.3084997 0.2695682 0.2656843
```

```
# mean-centered
Ac <- scale(A, center = TRUE, scale = FALSE)
apply(Ac, 2, mean)
```

```
## [1]  2.220988e-17 -4.441163e-17 -4.441163e-17
```

```
apply(Ac, 2, sd)
```

```
## [1] 0.3084997 0.2695682 0.2656843
```

```
# mean-centered and standardized
As <- scale(A, center = TRUE, scale = FALSE)
apply(As, 2, mean)
```

```
## [1]  2.220988e-17 -4.441163e-17 -4.441163e-17
```

```r
apply(As, 2, sd)
```

```
## [1] 0.3084997 0.2695682 0.2656843
```

Your Turn: Find out how to use `scale()` to center the columns of a matrix in the following ways:

- median-center (i.e. subtracting the median of each variable).
- 25th percentile-center (i.e. subtracting the 25th-percentile of each variable).
- 75th percentile-center (i.e. subtracting the 75th-percentile of each variable).
- rescale the variables such that each column ranges from 0 to 1.

Cross Products

When handling data matrices, we will encounter various types of _association___ matrices. These matrices are the result of cross-products.

- minor product moment
- major product moment

Assuming the $\mathbf{A}$ is $n \times p$ and that $n > p$, the *minor product moment* of $\mathbf{A}$ is a $p \times p$ matrix $\mathbf{B}$:

$$\mathbf{B} = \mathbf{A}^T \mathbf{A}$$

It is called *minor* because $p < n$

The *major product moment* is an $n \times n$ matrix $\mathbf{B}$:

$$\mathbf{B} = \mathbf{A}\mathbf{A}^T$$

It is called *major* because $n > p$.

Based on the minor product moment, there are four types of cross-products:

- Raw cross-product matrix
- SSCP matrix
- Covariance matrix
- Correlation matrix

Consider the following data in matrix `M`:

```r
y <- c(1, 0, 1, 4, 3, 2, 5, 6, 9, 13, 15, 16)
x1 <- c(1, 2, 2, 3, 5, 5, 6, 7, 10, 11, 11, 12)
x2 <- c(1, 1, 2, 2, 4, 6, 5, 4, 8, 7, 9, 10)

A <- cbind(y, x1, x2)
rownames(A) <- letters[1:12]
A
```

```
##    y x1 x2
## a  1  1  1
## b  0  2  1
## c  1  2  2
## d  4  3  2
## e  3  5  4
## f  2  5  6
## g  5  6  5
## h  6  7  4
## i  9 10  8
## j 13 11  7
## k 15 11  9
## l 16 12 10
```

B is a symmetric matrix of order 3×3. The diagonal entries of this matrix denote the raw sums of squares of each variable, and the off-diagonal elements denote the raw sums of cross products.

```r
# raw sums of squares
B <- t(A) %*% A
B
```

```
##      y  x1  x2
## y  823 702 542
## x1 702 639 497
## x2 542 497 397
```

Mean-Centered (SSCP) Matrix

We can also express the sums of squares and cross products as deviations about the means of $\mathbf{y}$, $\mathbf{x_1}$ and $\mathbf{x_2}$. The mean-corrected sums of squares and cross-products matrix is often more simply called the **SSCP** (sums of squares and cross products) matrix and is expressed in matrix notation as:

$$\mathbf{S} = \mathbf{A}^T\mathbf{A} - \frac{1}{n}(\mathbf{A}^T\mathbf{1})(\mathbf{1}^T\mathbf{A})$$

where $\mathbf{1}$ denotes a 12×1 unit vector and n denotes the number of observations. The last term on the previous equation represents the correction term and is a generalization of the usual scalar formula for computing sums of squares about the mean:

$$\sum(x_i - \bar{x})^2 = \sum x_i^2 - \frac{1}{n}\left(\sum x_i\right)^2$$

In R:

```r
n <- nrow(A)
ones <- rep(1, n)

S <- t(A) %*% A - (1/n) * (t(A) %*% ones) %*% (t(ones) %*% A)
```

Covaraince and Correlation Matrices

The covariance matrix **C** is obtained from the mean-centered SSCP matrix by simply dividing each entry of **S** by the scalar n. That is:

$$\mathbf{C} = \frac{1}{n}\mathbf{S}$$

```
C <- (1/n) * S
C
```

```
##           y      x1        x2
## y  29.52083 19.4375 14.437500
## x1 19.43750 14.1875 10.687500
## x2 14.43750 10.6875  8.909722
```

In many statistical software, like R, the actual formula to compute the variance and covariance involves dividin by $n-1$ insted of n.

Of course, you can also use the function `cov()`:

```
cov(A)
```

```
##           y       x1        x2
## y  32.20455 21.20455 15.750000
## x1 21.20455 15.47727 11.659091
## x2 15.75000 11.65909  9.719697
```

Notices that `cov(A)` is different from `C = (1/n) * S`, this is because the `cov(A)` divides by $n-1$:

```
(1 / (n - 1)) * S
```

```
##           y       x1        x2
## y  32.20455 21.20455 15.750000
## x1 21.20455 15.47727 11.659091
## x2 15.75000 11.65909  9.719697
```

The correlatoin matrix **R** is related to **S**, the SSCP matrix, and **C**, the covariance matrix. If we take the square roots of the three variables $\mathbf{y}$, $\mathbf{x_1}$ and $\mathbf{x_2}$, and enter the reciprocals of these square roots in a diagonal matrix, we have:

```
# mean center vectors
y_centered <- y - mean(y)
x1_centered <- x1 - mean(x1)
x2_centered <- x2 - mean(x2)
```

```
sqr_y <- 1 / sqrt(sum(y_centered * y_centered))
sqr_x1 <- 1 / sqrt(sum(x1_centered * x1_centered))
sqr_x2 <- 1 / sqrt(sum(x2_centered * x2_centered))

D <- diag(c(sqr_y, sqr_x1, sqr_x2))
D
```

```
##            [,1]       [,2]       [,3]
## [1,] 0.05313064 0.00000000 0.00000000
## [2,] 0.00000000 0.07664017 0.00000000
## [3,] 0.00000000 0.00000000 0.09671132
```

Then, by pre and postmultiplying **S** by **D** we can obtain the correlation matrix **R**

$$\mathbf{R} = \mathbf{DSD}$$

```
R <- D %*% S %*% D
R
```

```
##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.9497803 0.8902164
## [2,] 0.9497803 1.0000000 0.9505853
## [3,] 0.8902164 0.9505853 1.0000000
```

Consider the following matrix:

```
set.seed(9867)
M <- matrix(10 * round(runif(30), 2), nrow = 10, ncol = 3)
```

Vector with column means

```
# vector of ones
ones = rep(1, nrow(M))

# for loop
xbar = c(0, 0, 0)
for (j in 1:ncol(M)) {
  xbar[j] = mean(M[ ,j])
}
xbar
```

```
## [1] 6.62 4.67 5.39
```

```
# using apply()
apply(M, 2, mean)
```

```
## [1] 6.62 4.67 5.39
```

```
# vectorized function
colMeans(M)
```

```
## [1] 6.62 4.67 5.39
```

Mean-centered Data

```
# using apply
apply(M, 2, function(x) x - mean(x))
```

```
##        [,1]  [,2]  [,3]
##  [1,] -3.72 -2.47  3.81
##  [2,] -2.12  3.13 -1.19
##  [3,]  2.88 -2.07 -5.29
##  [4,] -0.62  1.93 -1.69
##  [5,]  0.28  4.83  4.11
##  [6,]  1.88  0.03  0.91
##  [7,]  1.88 -4.67  3.91
##  [8,]  1.38  4.23  3.61
##  [9,] -5.02 -1.17 -2.79
## [10,]  3.18 -3.77 -5.39
```

```
# using sweep
xbar = colMeans(M)
sweep(M, MARGIN = 2, STATS = xbar, FUN = "-")
```

```
##        [,1]  [,2]  [,3]
##  [1,] -3.72 -2.47  3.81
##  [2,] -2.12  3.13 -1.19
##  [3,]  2.88 -2.07 -5.29
##  [4,] -0.62  1.93 -1.69
##  [5,]  0.28  4.83  4.11
##  [6,]  1.88  0.03  0.91
##  [7,]  1.88 -4.67  3.91
##  [8,]  1.38  4.23  3.61
##  [9,] -5.02 -1.17 -2.79
## [10,]  3.18 -3.77 -5.39
```

```
# using scale
scale(M, center = TRUE, scale = FALSE)
```

```
##        [,1]  [,2]  [,3]
##  [1,] -3.72 -2.47  3.81
##  [2,] -2.12  3.13 -1.19
##  [3,]  2.88 -2.07 -5.29
##  [4,] -0.62  1.93 -1.69
##  [5,]  0.28  4.83  4.11
##  [6,]  1.88  0.03  0.91
##  [7,]  1.88 -4.67  3.91
##  [8,]  1.38  4.23  3.61
##  [9,] -5.02 -1.17 -2.79
## [10,]  3.18 -3.77 -5.39
## attr(,"scaled:center")
## [1] 6.62 4.67 5.39
```

Challenge: mean-centered data

Imagine a BIG matrix that does not fit into memory. For illustrations purposes, consider a "big" matrix 1000000 rows and 5 columns.

- read line by line
- accumulate the values, and possibly keep track of number of lines
- when reaching end-of-file, then divide by number of read lines

```
BIG = matrix(runif(300), 10, 3)

xsum = c(0, 0, 0)
for (i in 1:nrow(BIG)) {
  xsum = xsum + BIG[i, ]
}
xsum / nrow(BIG)
```

```
## [1] 0.6136184 0.5125541 0.4305178
```
