Data Structures

Jim Harner

10/4/2020

1.4 Data Structures

Material in this section is partially based on Hadley's Wickham's Advanced R Data Structures chapter.

The canonical data structure in R is the vector. Vectors are of two types:

- atomic vectors
- lists

Other data structures can be constructed from these basic types. A `matrix`, or more generally an `array`, is constructed from an atomic vector by specifying a `dim` attribute. A `data.frame` is a special type of `list`.

Data structures in R are organized by:

- Dimensionality
- Homogeneity.

Atomic vectors and R objects made from vectors by specifying a `dim` attribute are homogeneous, i.e., their elements are of the same type (see below). Lists and data frames are heterogeneous, i.e., they can be composed of mixed types, as explained below.

Dimension	Homogeneous	Heterogeneous
1-dim	atomic vector	list
2-dim	matrix	data frame
n-dim	array	

These data structures are now discussed including their metadata as specified in the data's attributes.

1.4.1 Atomic vectors

The common types of atomic vectors are:

- numeric (double)
- character
- integer
- logical

Atomic vectors are created by the R function `c()`, i.e., combine.

We now consider the `puromucin` data in which we predict `velocity` in terms of `conc`.

```
conc <- c(.02, .02, .06, .06, .11, .11, .22, .22, .56, .56, 1.1, 1.1)
velocity <- c(76L, 47L, 97L, 107L, 123L, 139L, 159L, 152L, 191L, 201L, 207L, 200L)
```

Since `conc` contains decimals, it is numeric (double). The `L` following the integers forces `velocity` to be integer.

If you execute each chunk individually (by clicking on the right error above), you will see that `conc` and `velocity` are in the Global Environment (Click the `Environment` tab.). When R searches for a symbol name, e.g., `conc`, it starts in the Global Environment.

From a theoretical perspective `velocity` can be modeled by the Michaelis-Menten function:

$$y = \frac{\beta_0 x}{\beta_1 + x} + \epsilon$$

.
where x is the concentration (`conc`) and y is the velocity of the enzymatic reaction (`velocity`). This can be fit by a nonlinear regression using the `nls` function. Notice that the R expression, `velocity ~ (beta0 * conc) / (beta1 + conc)`, mimics the mathematical expression above, but without the addive error term given by ϵ, which is assumed by default.

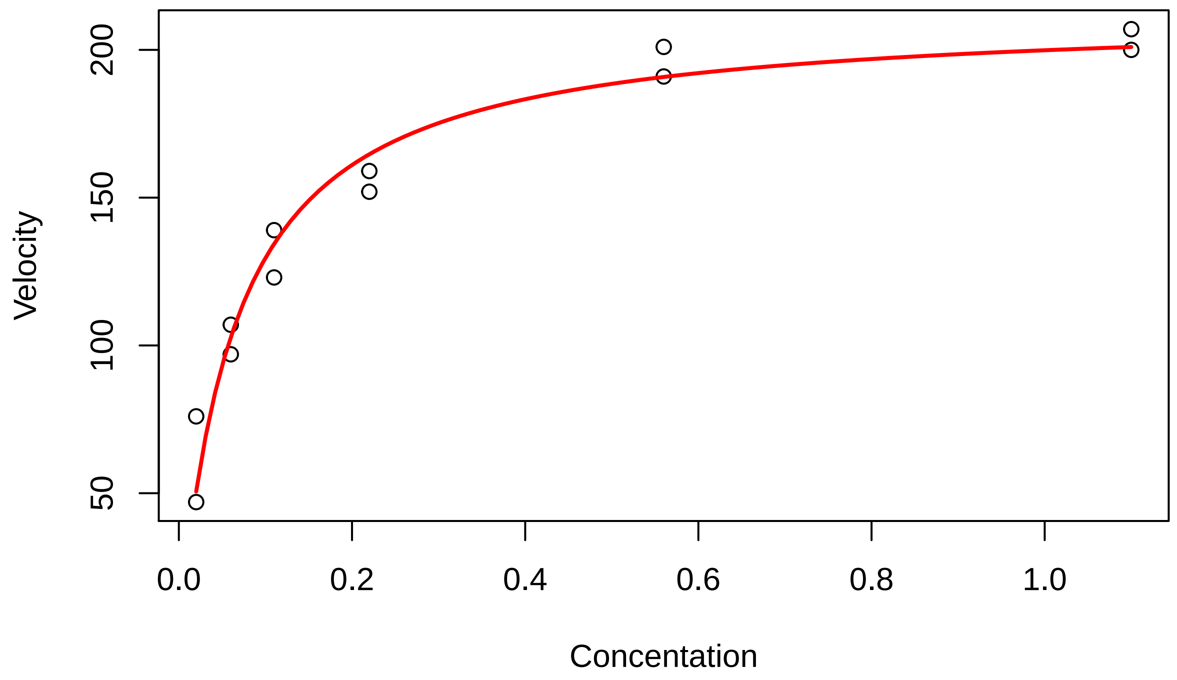
```
puromycin.nls <- nls(velocity ~ (beta0 * conc) / (beta1 + conc),
                     start=c(beta0=200, beta1=0.1))
summary(puromycin.nls)
```

```
##
## Formula: velocity ~ (beta0 * conc)/(beta1 + conc)
##
## Parameters:
##        Estimate Std. Error t value Pr(>|t|)
## beta0 2.127e+02  6.947e+00  30.615 3.24e-11 ***
## beta1 6.412e-02  8.281e-03   7.743 1.57e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.93 on 10 degrees of freedom
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 6.103e-06
```

Initial values must be set. β_0 is the asymptotic value of `velocity`, about 200. β_1 is the value of `conc` at which the function reaches half its asymptotic value, about 0.1. We fit the model using atomic vectors for x and y, in our case `conc` and `velocity`, respectively. The `start` argument uses a named numeric vector (done here) or a named list (see below).

Next, let's visually explore the fit.

```
plot(conc, velocity, xlab="Concentation", ylab="Velocity")
conc.seq <- seq(min(conc), max(conc), length=100)
lines(conc.seq, predict(puromycin.nls, list(conc=conc.seq)), col = "red", lwd = 2)
```



In order to plot the curve, we use the `lines` function. We could have used `fitted` rather than `predict` to get the y-values, but the curves would have been segmented. Instead, we use a sequence of x values from `min` to `max` concentration of length 100 to get a smooth curve. The data appear to follow the Michaelis-Menten function, but we have not yet done a formal test.

Numeric vectors can be used in arithmetic expressions where the operations are performed element by element. Rules exist for cases in which vectors in an expression are not of equal length. This is called *vectorized arithmetic*. See CRAN's An Introduction to R. Likewise, strings which form the elements of a character vector can be manipulated.

Matrices and Arrays If a `dim()` attribute is added to an atomic vector, it behaves like a multi-dimensional array. A matrix is a special case of array, which has two dimensions. Matrices are commonly used in computational statistics.

Matrices are easy to construct in R

```
x <- c(1:12)
dim(x) <- c(6, 2)
x
```

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

```
length(x)
```

```
## [1] 12
```

```
nrow(x)
```

```
## [1] 6
```

```r
ncol(x)
```

```
## [1] 2
```

Names for the rows and columns of the matrix can easily be added as follows:

```r
rownames(x) <- paste("Obs", 1:6, sep="")
colnames(x) <- paste("Var", 1:2, sep="")
x
```

```
##      Var1 Var2
## Obs1    1    7
## Obs2    2    8
## Obs3    3    9
## Obs4    4   10
## Obs5    5   11
## Obs6    6   12
```

R has many operators and functions for manipulating matrices. For example, `%*%` is used for matrix multiplication and `t` is a function for transposing a matrix. See An Introduction to R for details.

A three dimension array is also easily constructed.

```r
dim(x) <- c(3, 2, 2)
x
```

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

We can think of this as two 3×2 matrices forming a three-dimensional structure.

We can also use the `matrix` and `array` functions to create matrices and arrays.

```r
matrix(1:12, nrow=6, ncol=2)
```

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

```r
array(1:12, c(3, 2, 2))
```

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

1.4.2 Lists

Lists differ from atomic vectors because their elements can be of any type, including lists, i.e., a lists is a recursive data structure. You construct lists by using `list()`.

```
(l <- list(c(1, 2, 3), c("a", "b"), c(TRUE, FALSE)))
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a" "b"
##
## [[3]]
## [1]  TRUE FALSE
```

Single brackets, e.g., `[1]` extract the first element of the list, i.e., a list, whereas double brackets `[[1]]` extract the actual elements of the list.

```
l[1]
```

```
## [[1]]
## [1] 1 2 3
```

```
l[[1]]
```

```
## [1] 1 2 3
```

Notice that the first element of the list is a double vector, the second is a character vector, and the third element is a logical vector.

The following list has an element which is a list, i.e., the data structure is recursive:

```
(l.recursive <- list(list(c(1, 2,3), c("a", "b")), c(TRUE, FALSE)))
```

```
## [[1]]
## [[1]][[1]]
## [1] 1 2 3
##
## [[1]][[2]]
## [1] "a" "b"
##
##
## [[2]]
## [1]  TRUE FALSE
```

```
is.recursive(l.recursive)
```

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

The first element of the list is a list with two elements given by `[[1]][[1]]` and `[[1]][[2]]`

```
l.recursive[1]
```

```
## [[1]]
## [[1]][[1]]
## [1] 1 2 3
##
## [[1]][[2]]
## [1] "a" "b"
```

```
l.recursive[[1]][[1]]
```

```
## [1] 1 2 3
```

The elements of the list can be named, which allows one to extract each element more easily with a '$' as follows:

```
# A named list
(l.named <- list(A=c(1, 2, 3), B=c("a", "b"), C=c(TRUE, FALSE)))
```

```
## $A
## [1] 1 2 3
##
## $B
## [1] "a" "b"
##
## $C
## [1]  TRUE FALSE
```

```
l.named$B
```

```
## [1] "a" "b"
```

```
l.named[[2]]
```

```
## [1] "a" "b"
```

More information on lists is available in An Introcution to R.

1.4.3 Data Frames

A data frame is a list of equal-length vectors in which the elements of each column must be of the same type, but the columns can be of different types.

```
df <- data.frame(Var1=1:6, Var2=7:12, Var3=rep(c("a", "b"), each=3))
df
```

```
##   Var1 Var2 Var3
## 1    1    7    a
## 2    2    8    a
## 3    3    9    a
## 4    4   10    b
## 5    5   11    b
## 6    6   12    b
```

```
str(df)
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ Var1: int  1 2 3 4 5 6
##  $ Var2: int  7 8 9 10 11 12
##  $ Var3: Factor w/ 2 levels "a","b": 1 1 1 2 2 2
```

The function `str` shows the structure of an R object. Notice that `Var3` is coerced (by default) into a factor with 2 levels.

```r
attributes(df$Var3)
```

```
## $levels
## [1] "a" "b"
##
## $class
## [1] "factor"
```

```r
levels(df$Var3)
```

```
## [1] "a" "b"
```

A factor has more structure than a character vector, e.g., a `levels` attribute. A factor can be created by the `factor` function.

We have created various R objects which are now in the Global Environment, which can be seen by:

```r
ls()
```

```
## [1] "conc"          "conc.seq"      "df"            "l"
## [5] "l.named"       "l.recursive"   "puromycin.nls" "velocity"
## [9] "x"
```

Let's remove the variables we no longer need:

```r
rm(conc.seq, df, l, l.named, l.recursive, puromycin.nls, x)
```

Rather than analyze the puromycin data using two vectors, we can combine `conc` and `velocity` into a data frame.

```r
puromycin.df <- data.frame(conc=conc, velocity=velocity)
puromycin.df
```

```
##    conc velocity
## 1  0.02       76
## 2  0.02       47
## 3  0.06       97
## 4  0.06      107
## 5  0.11      123
## 6  0.11      139
## 7  0.22      159
## 8  0.22      152
## 9  0.56      191
## 10 0.56      201
## 11 1.10      207
## 12 1.10      200
```

```r
str(puromycin.df)
```

```
## 'data.frame':    12 obs. of  2 variables:
##  $ conc    : num  0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 ...
##  $ velocity: int  76 47 97 107 123 139 159 152 191 201 ...
```

Once we have formed the data frame `puromycin.df`, we can remove `conc` and `velocity` from the Global environment using `rm`:

```r
ls()
```

```
## [1] "conc"         "puromycin.df" "velocity"
```

```r
rm(conc, velocity)
ls()
```

```
## [1] "puromycin.df"
```

If you would try to run the `nls` model above, it would not run since R would not be able to find `conc` and `velocity`. You would need to substitute `puromycin.df$conc` and `puromycin.df$velocity` for these variables. Since this is awkward, we look at two solutions.

The first way involves specifying the data frame in the `data` argument.

```r
puromycin.nls <- nls(velocity ~ (beta0 * conc) / (beta1 + conc),
                     data=puromycin.df, start=c(beta0=200, beta1=0.1))
puromycin.nls$data
```

```
## puromycin.df
```

In the first model above, `conc` and `velocity` are found by `nls` in the global environment, whereas here these variables are found in the environment created by the data frame `puromycin.df`.

The problem with this approach is that `conc` and `velocity` are only available in this single statement. R will not automatically look inside a data frame to find variable names. A better approach is to put `puromycin.df` in R's search path.

```r
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

```r
ls() # or ls(pos=1), i.e., position 1 (Global Env) is the default for ls()
```

```
## [1] "puromycin.df"  "puromycin.nls"
```

```r
attach(puromycin.df)
search()
```

```
##  [1] ".GlobalEnv"        "puromycin.df"      "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

```r
ls(pos=2)
```

```
## [1] "conc"     "velocity"
```

Notice that `puromycin.df` was placed in the second position in the search path. Now we can run the model without using the `$` operator or specifying the `data` argument.

```r
puromycin.nls <- nls(velocity ~ (beta0 * conc) / (beta1 + conc),
                     start=c(beta0=200, beta1=0.1))
puromycin.nls$data
```

```
## parent.frame()
```

```r
puromycin.nls
```

```
## Nonlinear regression model
##   model: velocity ~ (beta0 * conc)/(beta1 + conc)
##    data: parent.frame()
##     beta0     beta1
## 212.68363   0.06412
##  residual sum-of-squares: 1195
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 6.103e-06
```

R looks for `conc` and `velocity` in the Global Environment and does not find them. It then looks for them in position 2 and finds them.

If you look at the Environment tab and execute R chunk by chunk, you can see how the contents of the Global Environment change. You can also click on the dropdown arrow and select `puromycin.df` to see its contents.

Once we are finished accessing the variables in `puromycin.df`, it should be detached. Otherwise we will keep attaching another copy every time the code is run.

```r
detach(puromycin.df)
```

We can summarize the output and test the parameters for significance using `summary`.

```r
summary(puromycin.nls)
```

```
##
## Formula: velocity ~ (beta0 * conc)/(beta1 + conc)
##
## Parameters:
##        Estimate Std. Error t value Pr(>|t|)
## beta0 2.127e+02  6.947e+00  30.615 3.24e-11 ***
## beta1 6.412e-02  8.281e-03   7.743 1.57e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.93 on 10 degrees of freedom
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 6.103e-06
```

Notice that both regression coefficients are highly significant.

The concept of environments is critical to understanding R's functional programming paradigm.

More information on data frames is available in An Introcution to R.