R: Attributes, Factors, and Matrices

Signal Data Science

In this lesson, we'll be discussing **attributes**.

First, we'll begin with a high-level overview of what attributes are. Afterward, we'll look at two very common use cases of attributes: *factors* and *matrices*.

Attributes

Every object in R can be associated with its **attributes**, where each attribute of an object has a unique *name* and takes on some *value*.

- Use `attributes()` to view the attributes of the built-in variable `mtcars`. Try modifying some of the attributes through direct assignment – does this work? When does it fail?
- Using `attr()`, write a function that "doubles" all the names of a named list (*e.g.*, `"colname"` $\rightarrow$ `"colnamecolname"`).

In general, attributes are not preserved upon object modification. However, *names*, *dimensions*, and *class* usually persist. These three attributes, being very important, have dedicated functions to access them (`names()`, `dim()`, and `class()`), which should always be used instead of `attr()`.

Factors

Factors are used to represent data that can fall into one of a finite number of categories. Examples of data most naturally encoded as factors include: gender, marital status, race, nationality, and profession.

The *factor* class is built on top of integer vectors. This is the crucial insight about factors, similar to the insight about data frames being built on top of lists, and if you remember a single thing from this section, let it be *that*.

Factors differ from simple integer vectors in one important way: each factor is associated with a `"levels"` attribute, accessible through `levels()`. Each entry in a factor *must* be equal to one of its levels.

- Create a factor with `factor()` applied to an arbitrary vector, examine its levels, and try to assign a value to the factor that isn't one of its levels. Add that value to its levels (using `levels()`) and retry the assignment.
- Can you combine two factors with `c()`? If not, what's a reason why?

Hadley Wickham has the following to say about where factors often show up:

> Sometimes when a data frame is read directly from a file, a column you'd thought would produce a numeric vector instead produces a factor. This is caused by a non-numeric value in the column, often a missing value encoded in a special way like `.` or `-`. **To remedy the situation, coerce the vector from a factor to a character vector, and then from a character to a double vector.** (Be sure to check for missing values after this process.) Of course, a much better plan is to discover what caused the problem in the first place and fix that; using the `na.strings` argument to `read.csv()` is often a good place to start.
>
> Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels or their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data.[1]
>
> While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others (like `c()`) will use the underlying integer values. For this reason, **it's usually best to explicitly convert factors to character vectors if you need string-like behaviour**.

Be sure to pay attention to these details when you load data from external sources.

- Define `f1 = factor(letters)`, `f2 = rev(factor(letters))`, and `f3 = factor(letters, levels = rev(letters))`. How do these three factors differ?
- Suppose you're reading in data from a file that's read into a character file into a variable `fruits`, which ends up being equal to `c("apple", "grapefruit", "NA", "apple", "apple, "-", "grapefruit", "durian")`. Read the `factor()` documentation and figure out how to

[1] Wickham also writes: "A global option, `options(stringsAsFactors = FALSE)`, is available to control this behaviour, but I don't recommend using it. Changing a global option may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing), and global options make code harder to understand because they increase the number of lines you need to read to understand how a single line of code will behave."

convert `fruits` into a factor such that the character-encoded missing values, `"NA"` and `"-"`, end up as actual `NA`s. Use `table()` to view the `fruits` factor and verify that the `"NA"` and `"-"` strings were correctly converted into `NA`s.

- Write a function that takes in an arbitrary *character vector* and returns that vector as a factor, with `NA` included in the levels of the factor if it exists in the original factor.
- Write a function takes in a data frame `df` and converts the first `floor(ncol(df)/2)` columns into factors.
- Write a function that takes in a data frame and converts every column with at most 5 unique values into a factor. (You may find `unique()` useful.)
- Write a function that takes in a data frame and, for each factor column, replaces every `NA` with the most common non-`NA` value in the column. Generate a toy dataframe to use to demonstrate that your function works. Write a different function that replaces every `NA` value with a random level of the factor, distributed identically to their relative frequencies in the column's non-`NA` values. How can you make this imputation method reproducible? (*Hint:* Try `set.seed()`.)

The motivation behind the functionality in the next exercise is that linear regressions can't be directly run with factors, which are *categorical* variables, in the predictors; the integer-valued `levels()` of a factor don't encode any meaning. Instead, we separate all but one level out into *binary indicator variables*, which we can regress against. The simplest and most common example of this is encoding a gender variable as 0 or 1.

- Write a function that takes in a data frame, with some but not all columns being factors, and expands each factor into a set of *indicator variables* within the data frame. Precisely, for each factor, *replace* that factor column with a number of *binary indicator variables*, having these properties:
  - Every level of the factor, aside from the first level, corresponds to a new binary indicator variable.[2]
  - For a particular row, each binary indicator variable takes on the value 1 if the original factor was equal to its corresponding level and 0 otherwise.
  - For each binary indicator variable, its name is equal to the following strings concatenated together in order: (1) the name of the original

[2]The reason for not making a binary indicator variable for the first level of the factor is that the state of "factor is equal to its first level" can already be represented by the `length(levels()) - 1` binary indicator variables all being set to 0. As such, if we add in a binary indicator variable for the first level, it will cause problems with collinearity that breaks linear techniques (including regression).

factor, (2) an underscore ("_"), and (3) the name of the factor level itself.

You can assume that there are no NAs in the input dataframe. Test your code on the data frame `df = mtcars[1:10,]; for (n in c("cyl", "am", "carb")) df[[n]] = factor(df[[n]])`; you should obtain a result with 13 columns.

To illustrate the desired functionality, consider the following diagram:

```
col      col_2  col_3
---      -----  -----
 1         0      0
 2         1      0
 1    =>   0      0
 2         1      0
 2         1      0
 3         0      1
```

Finally, you'll look at some real (and messy!) data from a multi-decade study of adolescent health.

- Use `load()` to load `time.dat`, a two-column subset of the data from the National Longitudinal Study of Adolescent Health. (The function will load it into the variable `df`.) Look at the documentation for Wave II: In-Home Questionnaire, Public Use Sample and read about the two questions in the data (check the column names). Write some code to convert each column to numeric values representing "number of hours past 8:00 PM"[3] and plot two histograms, one for each column, overlaid on top of each other using multiple `geom_histogram()` calls with the `fill` and `alpha` parameters. (*Hint:* Move the `aes()` call into the `geom_histogram()` calls.)[4]

Matrices

Arrays are simply atomic vectors with a *dimension attribute*, which must be a vector of integers. (The size of the vector must be compatible with its dimensions.) **A matrix is an array with two dimensions.** Matrices can be created as such:

```
> matrix(1:100, nrow=10)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
```

[3]Deciding what the cutoff point is where you stop counting something as "X hours after 8 PM" and start counting it as "24-X hours before 8 PM" is an arbitrary choice in principle, but some choices are more sensible than others – use your human judgment to optimize for interpretability!

[4]You can use string manipulation or R's inbuilt time classes. Try doing it both ways. For plotting the overlaid histograms, try `ggplot(df) + geom_histogram(aes(x=H2GH42), fill="red", alpha=0.2) + geom_histogram(aes(x=H2GH43), fill="blue", alpha=0.2)`.

```
 [1,]    1   11   21   31   41   51   61   71   81    91
 [2,]    2   12   22   32   42   52   62   72   82    92
 [3,]    3   13   23   33   43   53   63   73   83    93
 [4,]    4   14   24   34   44   54   64   74   84    94
 [5,]    5   15   25   35   45   55   65   75   85    95
 [6,]    6   16   26   36   46   56   66   76   86    96
 [7,]    7   17   27   37   47   57   67   77   87    97
 [8,]    8   18   28   38   48   58   68   78   88    98
 [9,]    9   19   29   39   49   59   69   79   89    99
[10,]   10   20   30   40   50   60   70   80   90   100
```

Now, some exercises:

- Write a function that takes an n-by-m numeric matrix and turns it into a m-by-n numeric matrix where both matrices give the same result when `as.numeric()` is applied.[5]

Matrices work similarly to data frames: you have access to `colnames()`, `rownames()`, `ncol()`, and `nrow()`, and subsetting (broadly speaking) works the same way.[6]

- Run the following code: `df = data.frame(matrix(1:100, nrow=10)); df[5, 5] = NA; df[6, 6] = NA`. Figure out how `df[is.na(df)]` works. Write and test a function that takes as input a data frame `df` of purely numeric data and a number `k`, returning a vector of every number in `df` divisible by `k`.

You won't work directly with matrices directly very much for most data science applications – data frames are much more common and important. In practice, they mostly show up as *intermediate forms* in the conversion and manipulation of data, and it's helpful to be aware of matrices so you can debug problems when they show up. (Knowing how matrices work in R might be more important if you're doing numerical simulations of some sort, perhaps in the computational science.)

- Write a function `min_matrix(n, m)` with `n` rows and `m` columns where the value in row i, column j is equal to $\min(i, j)$.
- Write a function to determine if an input matrix is symmetric across its main diagonal (so the element in row i, column j is equal to the element in row j, column i). Only square matrices can be symmetric; you may find `t()` helpful.
- Write a function `trace(mat)` to calculate the trace of an input matrix, *i.e.*, the sum of its diagonal elements. (You may find `diag()` helpful.) Check whether or not the `trace()` function is *linear*, that is, whether

[5]Just modify the attributes directly by assigning to `dim(mat)`, `attributes(mat)$dim`, or `attr(mat, "dim")`!

[6]There are some nuances – if you want to learn about them, read Hadley Wickham's *Advanced R*.

it's true that `trace(A + B) = trace(A) + trace(B)` and `trace(c*A) = c*trace(A)` for some constant `c`.

Matrices can be multiplied together in a special way which produces another matrix. Such operations are *very* common, arising all over the sciences and in many numerical algorithms.[7]

- If you aren't familiar with the operation of matrix multiplication, read about it briefly on Wikipedia. Does matrix multiplication work normally with the `*` operator? Why or why not? (Try testing multiplication with the identity matrix.)[8]
- Write a function `matrix_mult(A, B)` that implements matrix multiplication, computing `A %*% B` (without using the `%*%` operator, of course). Using the `timeit` package, compare the performance of `matrix_mult()` with the built-in `%*%` operator for matrices of different sizes. Can you speed up your code?

In the following exercises, you will analyze the output of the mystery function `mystery = function(x) matrix(c(cos(x), -sin(x), sin(x), cos(x)), nrow=2)`.

- Is the output of `mystery()` *periodic* in some sense with respect to its inputs? Check if this holds in practice; if there's a discrepancy, explain it. (*Hint:* If you aren't familiar with the behavior of the `sin()` and `cos()` functions, graph a scatterplot of their values against `seq(0, 10, 0.01)`.)[9]
- Write a function to turn lists of length 4 into 2-by-2 matrices, forming a list-matrix capable of holding different data types. (*Hint:* Use `dim()`.) Speculate on some use cases of list-matrices.
- On the 2D plane, we can identify the *point* (x, y) with the *column vector* `matrix(c(x,y), nrow=2, ncol=1)`. First, write a function which takes in a list of column vectors, internally adds each one as a row of a two-column dataframe, and then uses `ggplot()` to graph a *scatterplot* of all the points in the input list. Second, modify this function to accept an argument `x` and so that for every column vector `c` in its list of points, instead of putting the data in `c` directly into the data frame, it does so for the product `mystery(x) %*% c` (for the `mystery()` function defined in a previous exercise). Experiment with different values of `x` and come up with a geometric understanding of how the output graph changes as you modify `x`.[10]

[7] Indeed, quantum mechanics was originally formulated as the theory of matrix mechanics owing to the useful properties of matrix multiplication.

[8] Matrices are vectors, so the multiplication is done element-by-element. Proper matrix multiplication is done with the the `%*%` operator.

[9] We have `mystery(0) != mystery(2*pi)` because of floating-point imprecision.

[10] Calling `mystery(x)` returns a 2-dimensional rotation matrix corresponding to rotation through the angle `x` (given in radians).

Principal component analysis

The technique of principal component analysis (PCA) can be conceptualized in the following manner: First, we find the direction along which the data varies the most (analogous to the semi-major axis of an ellipse). That is the first principal component. We then 'subtract off' that dimension of variation from the data and, in the reduced data, find the direction along which the data varies the most. This is the second principal component. Repeating this process, we end up with p principal components for a dataset of dimension p (containing p variables).

PCA is actually ordinarily calculated through the computation of the singular value decomposition of the covariance matrix of the data, but the above procedure better illustrates the intuition behind PCA. The first principal component (PC1) is the dimension which accounts for as much variation as possible, PC2 accounts for as much variation as possible after PC1, PC3 accounts for as much variation as possible after PC1 and PC2, and so on and so forth. PCA can also be thought of as a rotation of the coordinate axes.

- Read this StackExchange answer explaining the intuition behind PCA.

In the following, we will implement our own version of PCA using the method of successive principal component extraction described above and compare our answer to the PCA method implemented in R's `prcomp()`.

For illustration, let us consider a matrix of example data `X = matrix(1:100, nrow=10)`. First, `scale()` the matrix. (If we want to analyze the dimensions of greatest variation, it's important to first center our data and to scale the variables so that choice of units does not affect our end result.) We can get the results of PCA by simply running `prcomp(X)`, with the principal components being stored in the `$rotation` variable of the `prcomp()` output. Each principal component is represented by a vector pointing along the dimension corresponding to that PC.

We want to implement our own method for extracting the principal components, which will also be represented by such vectors. In order to check our results, we need a way to see if the `prcomp()` PCs point in the same directions as our PCs. To determine if two vectors point in the same direction, we can't just compare the elements of the vectors; for example, $(1,1)$ and $(10,10)$ point in the same direction but are clearly quite different on an element-by-element basis.

The key is to look at the dot product of the two vectors, given by $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$. It is also given by the identity $\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\| \, \|\mathbf{w}\| \cos\theta$, where $\|\mathbf{v}\|$ is the *magnitude* of the vector $\mathbf{v}$ (the square root of the sum of the square of each element of $\mathbf{v}$) and θ is the *angle* between the two vectors $\mathbf{v}$ and $\mathbf{w}$. If the angle between two vectors is close to either 0 or 180, then they point along the same direction (an angle of 180 means that one is flipped relative to the other, like $(1,1)$ and $(-1,-1)$).

- Write a function `norm(v)` which calculates the norm of `v`, `dot_prod(v,` `w)` which calculates the dot product of `v` and `w`, and `angle(v, w)` which calculates the angle between `v` and `w`. You may find the inverse cosine function `acos()` useful.

Next, given a matrix of data $\mathbf{X}$, the vector representing the dimension of greatest variation in the data is given by the vector $\mathbf{w}$ which *maximizes* the expression $(\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w})\ /\ (\mathbf{w}^\mathsf{T}\mathbf{w})$, where $\mathbf{X}^\mathsf{T}$ represents the transpose of the matrix $\mathbf{X}$.

- Write a function `objective(X, w)` which calculates the value of the above expression for a given matrix `X` and vector `w`. Since it is an expression which we wish to maximize, it is called an objective function.

Add the following code into your script:

```
objective_X = function(X) {
  function (w) {
    objective(X, w)
  }
}
```

The above function, `objective_X()`, will return a *function of `w` only* which evaluates the objective function at that value of `w`. The purpose of `objective_X()` is as follows: Often, we will want to evaluate the objective function for a given data matrix `X` many different times for different values of `w`. As such, we can set `obj = objective_X(X)`, and whenever we want to evaluate the objective function for different vectors `w` we just call `obj(w)`.

- Write a function `extract_pc(X)` which uses `optim()` to maximize the objective function in order to find a vector corresponding to the first principal component of the data matrix `X`. Keep the following in mind:
  - By default, `optim()` tries to *minimize* the objective function; read the documentation to figure out how to change this to maximization.
  - Change the default behavior of `optim()` so that the maximum number of iterations is 10000.
- Write a function `prcomp_pc(X, k)` which scales `X` and then uses `prcomp()` to extract and return the `k`th principal component of `X`.