

Built-in Tests/Exploring Data

Like most statistical packages R can do lots of tests on data with a minimum of fuss. Just as an example, load the coalition data and then try the Shapiro-Wilk normality test:

```
shapiro.test(coal.data$durat)
```

Clearly, W is significant so we reject the hypothesis that the durations are normal. Standard tests like Kolmogorov-Smirnov `ks.test()`, t-tests `t.test()` and Kruskal-Wallis `kruskal.test()` are available too. Take a look at

```
apropos("test")
```

for starters.

Notice that you may want to plot the (empirical) CDF of some data. Again, this is pretty intuitive:

```
dur.ecdf<-ecdf(coal.data$durat)
plot(dur.ecdf, col.h="blue")
```

Notice that we first assign the ecdf, then plot it. When we plot it, we use `col.h` as the coloring argument. This is unusual, but comes from the fact that ecdf has lots of individual pieces that can be colored different ways.

We can compare the empirical cdf with the cdf we are actually modeling it as. In this case, King et al. treat the durations as exponentially distributed. Is this a good fit? One crude way to eyeball it is to create some random exponential numbers with the same arrival rate and compare the ecdf.

```
ran.nos<-rexp(n=nrow(coal.data), rate=1/mean(coal.data$durat))
```

Notice:

- `rexp()` tells R we want random numbers drawn from an exponential distribution
- `n=nrow(coal.data)` tells R we want as many of them as there are observations in the data set (which is 314)
- `rate=1/mean(coal.data$durat)` tells R that the characteristic parameter of those random numbers—i.e. the rate (sometimes called λ)—should be one over the mean of the actual durations in the data. Always make sure you check the way that R is parameterizing its distributions: it may be the same or the reciprocal of your textbook definitions. See esp the **details** in the help of `?rbeta` and `?rgamma`

The fit looks ok to me.

Finally, it never hurts to do a few box plots. Here I'm looking at the temperature of some beavers, according to their activity level. First, you'll need to load the data

```
data(beavers)
boxplot(temp~ativ, data=beaver1)
```

Notice:

- the data set we want is `beaver1`, but that is kept within the `beavers` data set
- the tilde `~` tells R we want ‘the thing on the left explained by the thing on the right’. Here, that means `temp` explained by `ativ`. We will need this structure again.

Plotting plus: Identification, Legends and Greek Letters

Suppose we created the following plot (from some UN data) of GDP versus infant mortality:

```
data(UN)
plot(UN$gdp, UN$infant.mortality)
```

We would like to label (some of) the points. In this case, the data set is arranged such that the `row.names` are the actual countries (take a look at the data to see this). We can this fact along with the `identify()` function:

```
identify(UN$gdp, UN$infant.mortality, labels=row.names(UN))
```

Now click on the points that you care about. Notice

- the first argument of `identify()` is the x -axis which we need to ‘look up’ for the definition of the points
- the second argument of `identify()` is the y -axis which we need to ‘look up’ for the definition of the points
- `labels=` tells R what it should call the points. Here, that is the `row.names()` of the data set.

A different problem occurs when we want to label curves or points in general. Consider the following data set in which cats are categorized as male or female and then their body and heart weights are recorded. Suppose we want to plot points, with each sex color-coded. First, we will load and subset the data:

```
data(cats)
toms<-subset(cats, cats$Sex=="M")
queens<-subset(cats, cats$Sex=="F")
```

Then we’ll plot their data in different colors

```
plot(toms$Hwt, toms$Bwt, pch=19, col="blue")
lines(queens$Hwt, queens$Bwt, pch=21, col="red", type="p")
```

Note the use of `lines()` instead of `plot()` in the second call. Also, the use of `type="p"` to tell R we need points, not lines. The `pch=` argument is telling R to change the plotting character it used.

To add a legend, we need the `legend()` command:

```
legend(locator(), legend=c("Toms","Queens"), col=c("blue", "red"), pch=c(19,21))
```

Lots going on here:

- the first argument tells R we want to position the legend with the mouse: you will need to actually click somewhere on the plot for the legend to appear. There are other options here: we could specify an x,y point location or use a `position` argument (see `?legend`).
- the `legend=` itself is a character vector to be stacked—it is the names of our groups
- `col=` are the colors we used in order of the legend (i.e. Toms, then Queens)
- `pch=` are the plotting characters we used in order of the legend (i.e. Toms, then Queens)

Re-ordering a Data Set

Sometimes we need to reorder our data according to a particular variable's value. We need `order()` to do this, and we need it in a slightly convoluted way. The `order()` function rearranges a vector into ascending or descending order, but it *returns* a vector describing the order. To see this, consider

```
J<-c(1,pi,8,2,6,100,89) order(J)
```

The output tells you that the vector ascends in order 1st element, 4th element, 2nd element, 5th, 3rd, 7th, 6th—which, when you look at the vector, makes sense.

Suppose we wanted to re-order the coalition data by the `durat` variable. The magic happens with

```
o.coal<-coal.data[order(coal.data$durat),]
```

which is telling R to rearrange the *rows* (i.e. observations) of `coal.data` according to their values of `durat` (note the comma after the order call).

Missingness and Logical Vectors

With R, as in life, things are TRUE, FALSE or not known. Consider the following:

```
y<-c(1:5, NA)
```

Here NA is simply a piece of missing data. In general, that will be problematic for us, and we might need to tell R to ignore it. Try `mean(y)` for example and then try `mean(y, na.rm=T)` which tells R to take the mean of the vector `y`, but drop any NA values it finds in so doing.

We can ask R about the characteristics of the vector from this logical perspective. We can ask, for example, which elements of `y` are greater than 2:

```
y>2
```

We can convert this vector to a numerical (0,1,NA) vector using a straightforward trick:

```
z <- y>2
as.numeric(z)
```

This may not seem particularly important now, but it can be very helpful. Note: the logical operators are `==` ($=$), `>=` (\geq), `<=` (\leq) and `!=` (\neq).

An extra thing to notice here, that may be useful elsewhere, is the function `is.na()` used like this:

```
is.na(y)
```

which asks which values of `y` are missing. Of course, we can use `as.numeric(is.na(y))` here too.

A really helpful function that uses logical operators is `subset()` that enables us to pick up members of a vector (or matrix) that possess a certain property. Consider:

```
h<-matrix(1:100)
subset(h, h>64)
```

This will produce a subset of the members of `h` *such that* `h>64` is satisfied. Going back to our previous data work, consider

```
load("z:/pathway/coaldata.rdata")
short.lived<-subset(coal.data, coal.data$durat<9)
```

which will create a new data frame called `short.lived` which contains only those cabinets which collapsed before 9 months were up. To check we have it right, try

```
summary(short.lived$durat)
```

A nice extra command you might see is `a%%b` which checks whether `a` can be divided by `b` until there is no remainder. For example:

```
subset(h, h%%7==0)
```