STATS 380

Factors and Data Frames

Categorical Data

- A good deal of statistical data is of a form which indicates which one of several possible categories that an observation falls into.
- Examples are:
  - Eye colour: *brown*, *hazel*, *green*, *blue*.
  - Location: *North Island*, *South Island*, *other*.
  - Gender: *female*, *male*.
  - Pain level: *low*, *medium*, *high*.
- R provides a facility for creating this kind of data through the functions `factor` and `ordered`.

Factors

- The function `factor` creates data objects which represent variables containing *unordered* categorical data.
- It takes a character or numeric vector as an argument and returns a factor.

```
> (eyes = c("hazel", "blue", "brown",
            "green", "blue", "brown"))
[1] "hazel" "blue"  "brown" "green" "blue"
[6] "brown"
> (eyecol = factor(eyes))
[1] hazel blue  brown green blue  brown
Levels: blue brown green hazel
```

Factor Levels

- Elements of the set of possible categories that a categorical variable can take on are known as the *levels* of that factor.
- By default, R takes the levels to be the set of values occuring in the input data vector, *sorted into ascending order* (either numerically or alphabetically).
- When a factor is printed, the levels of the factor are displayed after the variable.

```
> eyecol
[1] hazel blue  brown green blue  brown
Levels: blue brown green hazel
```

Specifying Factor Levels

- The default set of factor levels, and the order they appear in can be specified with a second argument to `factor`.

```
> (eyecol = factor(eyes,
          levels = c("blue", "green",
                     "hazel", "brown")))
[1] hazel blue  brown green blue  brown
Levels: blue green hazel brown
```

- The levels of a factor can be obtained with the `levels` function.

```
> levels(eyecol)
[1] "blue"  "green" "hazel" "brown"
```

Ordered Factors

- Sometimes there is a natural order to a factor's levels. In this case factors are described as *ordered factors*.
- Ordered factors are created with the R function `ordered`.
- It is important to specify the levels when creating an ordered factor to ensure that they are in the correct relationship to each other.

```
> pain = ordered(c("low", "medium", "medium",
                   "high", "medium", "low"),
                 levels = c("low", "medium",
                            "high"))
```

Distinguishing Factors and Ordered Factors

- Factors and ordered factors can be told distinguished by the way their level sets are printed.
- The levels of ordered factors are separated by "<", when they are printed.

```
> eyecol
[1] hazel blue  brown green blue  brown
Levels: blue green hazel brown

> pain
[1] low    medium medium high   medium low
Levels: low < medium < high
```

Factor Predicates

- To tell whether a value is a factor, use the functions, `is.factor` and `is.ordered`.

```
> is.factor(eyecol)
[1] TRUE
> is.factor(pain)
[1] TRUE
> is.ordered(eyecol)
[1] FALSE
> is.ordered(pain)
[1] TRUE
```

Operations on Factors

- One of the only operations that makes sense on an unordered factor is to compare its values with a particular value using `==` or `!=`.
- For ordered factors, comparisons using `<`, `<=`, `>` and `>=` also make sense.

```
> eyecol == "blue"
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE

> eyecol < "blue"
[1] NA NA NA NA NA NA
Warning message:

In Ops.factor(eyecol, "blue") :
    '<' not meaningful for factors
```

Subsetting and Factors

- Because factors behave as though they were vectors (and internally they are vectors), the same kinds of subsetting operations apply to them.
- For example, if we have a vector `hgt` which contains the heights of class members and a factor called `sex` which contains the gender of class members then the following expressions make sense

```
hgt[1:10]             # first 10 heights
sex[1:10]             # first 10 genders
hgt[sex == "male"]    # heights of males
hgt[sex == "female"]  # heights of females
sex[hgt > 180]        # genders of tall people
```

Tabulation

- One of the few things that can be done with factors is to count the number of times each level occurs.
- This can be done with the function `table`.

```
> table(eyecol)
eyecol
 blue green hazel brown
    2     1     1     2

> table(pain)
pain
   low medium   high
     2      3      1
```

Cross-Tabulation

- It is also possible to use `table` to count the number of times each combination of the levels of two (or more) factors occurs.

```
> table(eyecol, pain)
       pain
eyecol  low medium high
  blue    0      2    0
  green   0      0    1
  hazel   1      0    0
  brown   1      1    0
```

- The resulting matrix (or more generally, array) is called a *contingency table*.

Obtaining Summaries over Factor Levels

- Factors provide a way of defining subgroups in a data set. It is useful to be able to obtain summaries for these subgroups. The function `tapply` can be used to do this. The function call

  `tapply(`*variable*`,` *factor*`,` *summary*`)`

  returns a vector containing the specified summaries for the given vector, broken down into the subgroups defined by specified factor.

- For the class height example, the expression

  ```
  tapply(hgt, sex, mean)
  ```

  will return a vector containing two elements; the average heights for males and females in the class. The values are named by the factor levels.

More Complex Summaries

- The `tapply` function can also obtain summaries broken down by several factors.
- An expression of the form

  `tapply(`*variable*`, list(`*fac1*`,` *fac2*`, ...),`
  summary`)`

  will produce an array, giving the summary broken down in the subgroups specified by the combinations of the given factor levels.

Statistical Models and Factors

- Factors are used in many statistical techniques.
- Examples are:
  - Analysis of variance
  - Analysis of covariance
  - Generalised linear models
  - Categorical response models

Binning Numeric Data

- When a numeric variable has a large number of observations, it can be useful to summarise the variable by defining a set of intervals, or bins, and counting the number of observations in each bin.
- R has a function, called `cut`, that can be used to assist this process.
- The `cut` function takes a numeric vector and a set of *cut points* and produces a factor indicating which interval each point lies in.

Example: Binning Numeric Data

```
> x = rnorm(1000)
> cuts = c(-Inf, -3:3, Inf)

> ints = cut(x, cuts)
> ints[1:5]
[1] (-1,0] (-1,0] (1,2]  (0,1]  (0,1]
8 Levels: (-Inf,-3] (-3,-2] ... (3, Inf]

> table(ints)
ints
(-Inf,-3]   (-3,-2]   (-2,-1]    (-1,0]
        0        19       146       330
    (0,1]     (1,2]     (2,3]  (3, Inf]
      348       129        27         1
```

Data Frames

- Data frames provide a way of grouping a number of related variables into a single data object.
- The function `data.frame` takes a number of vectors and/or factors and returns a single object containing all the variables.

  `df = data.frame(`*var1*`,` *var2*`, ...)`

- Each of the *var1*, *var2*, . . . is either an expression specifying a vector or factor, or a named expression of the form

  name `=` *var*

  where *name* provides a name for the given variable in the data frame.

An Example

A simple gender/height data set.

```
> sex = factor(rep(c("female", "male"), c(4, 4)))
> hgt = c(165, 176, 171, 177, 176, 193, 180, 193)
> (classinfo = data.frame(sex, hgt))
     sex hgt
1 female 165
2 female 176
3 female 171
4 female 177
5   male 176
6   male 193
7   male 180
8   male 193

> is.data.frame(classinfo)
[1] TRUE
```

Data Frame Structure

- Because data frames have a simple rectangular row/column layout, it is tempting to treat them as matrices.
- This is possible, but it is conceptually wrong and can lead to very inefficient computations.
- It is better to treat a data frame as a list of variables because this is how they are actually stored.

Subsetting

- Subsets can be extracted from data frames in the same way as from matrices.

```
> classinfo[c(1,3,5,7), ]
     sex hgt
1 female 165
3 female 171
5   male 176
7   male 180

> classinfo[,1]
[1] female female female female male   male
[7] male   male
Levels: female male
```

Extracting Variables from Data Frames

- The underlying representation of data frames is as a named list of vectors and factors. This representation can be used to extract elements by name.

```
> classinfo$hgt
[1] 165 176 171 177 176 193 180 193

> classinfo$sex
[1] female female female female male   male
[7] male   male
Levels: female male

> tapply(classinfo$hgt, classinfo$sex, mean)
female   male
172.25 185.50
```

Expressions Involving Data Frame Variables

- Names like `classinfo$sex` and `classinfo$hgt` can be tiresome to type, and there is a special way of specifying expressions involving variables from data frames.

```
> with(classinfo, tapply(hgt, sex, mean))
female   male
172.25 185.50
```

- The first argument to `with` is a data frame. The second is an expression involving the variables from the data frame.
- The second argument can be a compound expression grouped using `{` and `}`. (But remember that only the last expression in the compound will be returned as the value of the `with`.)

Adding Derived Variables to Data Frames

- The function `transform` can be used to produce new variables from those already present in a data frame and to combine all the variables into a new data frame.

```
> (nclass = transform(classinfo, hgt2 = hgt^2))
     sex hgt  hgt2
1 female 165 27225
2 female 176 30976
3 female 171 29241
4 female 177 31329
5   male 176 30976
6   male 193 37249
7   male 180 32400
8   male 193 37249
```

Alternative Subsetting Facilities

- Treating data frames as matrices is unnatural.
- The R function `subset` provides a better way of extracting subsets.
- Here is how to extract the subset of "males who are taller than 190cm" from the `classinfo` data frame.

```
> (mt = subset(classinfo,
               sex == "male" & hgt > 190))
   sex hgt
6 male 193
8 male 193
```

Selecting by Index

- The second argument to `subset` is required to be a logical vector.
- Sometimes, though, we need to extract rows that correspond to particular row indices.
- One example of this is when we decide to take a random sample of rows.

```
> (is = sample(nrow(classinfo), 4))
[1] 2 8 1 3
```

Selecting Cases by Index — Matrix Approach

- Treating a data frame as a matrix makes it possible to to extract the rows with the specified indices.

```
> classinfo[is,]
     sex hgt
2 female 176
8   male 193
1 female 165
3 female 171
```

- The values have been extracted in the order that the indices appear in the variable `is`.
- To get the values in "data order," simply sort the values in `is`.

```
> classinfo[sort(is),]
```

Selecting Cases by Index — Subset Approach

- It is easy to turn row indices into a true/false selection using matching.

```
> subset(classinfo, 1:nrow(classinfo) %in% is)
     sex hgt
1 female 165
2 female 176
3 female 171
8   male 193
```

- The expression `x %in% y` returns true/false values depending on whether each element of `x` does, or does not, occur in `y`.
- Notice that this produces the values in "data order."

Selecting Variables

- There is also a `select` argument to `subset` which can be used to select variables from a data frame.
- Here is an example selecting cases where `hgt > 175` for the variables `hgt2` and `sex` in the data frame `nclass`.

```
> subset(nclass, hgt > 175,
                 select = c(hgt2, sex))
   hgt2    sex
2 30976 female
4 31329 female
5 30976   male
6 37249   male
7 32400   male
8 37249   male
```

The Select Argument

- The select argument works as follows. The variable names are first replaced by their column indices and then the expression is evaluated. This means that selections like:

```
c(sex, age:weight, 20:30)
```

  will work. The ability to work with variable names rather than column indices can be helpful.
- Always be careful, however, to check that you are getting what you think you are getting.
- The function `names` will get the (vector of) names of the variables in a data frame. This can be helpful.

Sorting and Ordering

- The function `sort` can be used to sort a vector into ascending (or descending) order.

```
> (x = sample(1:10))
 [1]  5  6  4  1  7  2  8  9 10  3

> sort(x)
 [1]  1  2  3  4  5  6  7  8  9 10

> sort(x, decreasing = TRUE)
 [1] 10  9  8  7  6  5  4  3  2  1
```

Ordering Permutation

- The function `order` can be use to obtain an ordering permutation for a vector.
- An ordering permutation for `x` contains the rearrangement of `1:length(x)` which can be used to rearrange `x` into ascending (or descending) order.

```
> o = order(x)
> x[o]
 [1]  1  2  3  4  5  6  7  8  9 10

> o = order(x, decreasing = TRUE)
> x[o]
 [1] 10  9  8  7  6  5  4  3  2  1
```

Sorting Data Frames

- Ordering permutations can be used to reorder data frames so that one particular variable is in ascending (or descending) order.

```
> o = order(classinfo$hgt)
> classinfo[o,]
     sex hgt
1 female 165
3 female 171
2 female 176
5   male 176
4 female 177
7   male 180
6   male 193
8   male 193
```

Reading Data

- The standard way of storing statistical data is to store them in a rectangular form with rows corresponding to observations and columns corresponding to variables.
- Speadsheets are often used to store and manipulate data in this way.
- The function `read.table` can be used to read data which has been stored in this way.
- The first argument to `read.table` identifies the file to be read.

Example Data

- The file "mydatafile.txt" contains the (quoted) names of the islands of New Zealand bigger than 1000 square kilometers, together with the corresponding areas.

```
"South Island"    151215
"North Island"    113729
"Stewart Island"    1746
```

Example

- The data in the file can be read into a data frame as follows:

```
> (nz = read.table("mydatafile.txt"))
              V1     V2
1   South Island 151215
2   North Island 113729
3 Stewart Island   1746
```

- Notice that the function has created the names `V1` and `V2` for the variables.

Example (Continued)

- Using the optional argument `col.names`, it is possible to provide appropriate names for the variables.

```
> (nz = read.table("mydatafile.txt",
            col.names = c("Island", "Area")))
          Island   Area
1   South Island 151215
2   North Island 113729
3 Stewart Island   1746
```

Example (Continued)

- Using the optional argument `row.names`, it is possible to specify a variable to act as row names for the data frame.

```
> (nz = read.table("mydatafile.txt",
            col.names = c("Island", "Area"),
            row.names = "Island"))
                 Area
South Island   151215
North Island   113729
Stewart Island   1746
```

Example (Continued)

- There are many other arguments to `read.table` that can be used to customise how it works.

```
> (nz = read.table("mydatafile.txt",
            col.names = c("Island", "Area"),
            row.names = "Island",
            colClasses = c("character",
                           "character")))
                 Area
South Island   151215
North Island   113729
Stewart Island   1746
> nz$Area
[1] "151215" "113729" "1746"
```

Customised `read.table` Variants

There are a number of variants of `read.table` which have slightly different behaviour.

- `read.csv` assumes that (by default) columns are separated by commas.
- `read.csv2` assumes that (by default) columns are separated by semicolons and that the decimal indicator is a comma.
- `read.delim` assumes that (by default) columns are separated by tabs.
- `read.delim2` assumes that (by default) columns are separated by tabs and that the decimal indicator is a comma.