Dataframes

Lecture 8

Nicholas Christian

BIOST 2094 Spring 2011

Outline

1. Importing and exporting data
2. Tools for preparing and cleaning datasets
   - ☐ Sorting
   - ☐ Duplicates
   - ☐ First entry
   - ☐ Merging
   - ☐ Reshaping
   - ☐ Missing values

Dataframes

- R refers to datasets as dataframes
- A dataframe is a matrix-like structure, where the columns can be of different types. You can also think of a dataframe as a list. Each column is an element of the list and each element has the same length.
- To get/replace elements of a dataframe use either [ ] or $. The [ ] are used to access rows and columns and the $ is used to get entire columns

```
# state.x77, is a built-in R dataset of state facts stored as a matrix
# Type data(), to see a list of built-in datasets
data <- data.frame(state.x77) # First, convert to a dataframe

head(data)    # Print the first few rows of a dataset/matrix
tail(data)    # Print the last few rows of a dataset/matrix

names(data)                        # Column names
colnames(data); rownames(data)     # Column and row names
dim(data)                          # Dimension of the dataframe

data[,c("Population", "Income")]    # "Population" and "Income" columns
data$Area                           # Get the column "Area"
data[1:5,]                          # Get the first five rows
```

Importing Data from the Keyboard

- The `scan()` function is used to type or paste a few numbers into a vector via the keyboard.
- Press enter after typing each number and press enter twice to stop entering data
- Can also paste multiple numbers at each prompt.
- `scan()` can also be used to import datasets. It is a very flexible function, but is also harder to use. The function `read.table()` provides an easier interface.

```
 > x <- scan()
1: 4
2: 6
3: 10
4: 11
5:
Read 4 items
> x
[1]  4  6 10 11
```

Importing Data from Files

- The function `read.table()` is the easiest way to import data into `R`. The preferred raw data format is either a tab delimited or a comma-separate file (CSV).
- The simplest and recommended way to import Excel files is to do a Save As in Excel and save the file as a tab delimited or CSV file and then import this file in to `R`.
- Similarly, for SAS files export the file as a tab delimited or CSV file using `proc export`.
- Functions for importing data,

`read.table()`	Reads a file in table format and creates a dataframe
`read.csv()`	Same as `read.table()` where `sep=","`
`read.fwf()`	Read a table of fixed width formatted data. Data that is not delimited, need to specify the width of the fields.

read.table()

```
read.table(file, header = FALSE, sep = "", skip, as.is,
           stringsAsFactors=TRUE)
```

`file`	The name of the file to import
`header`	Logical, does the first row contain column labels
`sep`	Field separator character `sep=" "` space (default) `sep="\t"` tab-delimited `sep=","` comma-separated
`skip`	Number of lines to skip before reading data
`as.is`	Vector of numeric or character indices which specify which columns should not be converted to factors
`stringsAsFactors`	Logical should character vectors be converted to factors

- By default `R` converts character string variables to factor variables, use `stringsAsFactors` or `as.is` to change the default
- There are many more arguments for `read.table` that allow you to adjust for the format of your data

`read.table()`

- The default field separator is a space, if you use this separator keep in mind that,
  - Column names cannot have spaces
  - Character data cannot have spaces
  - There can be trouble interpreting extra blank spaces as missing data. Need to include missing data explicitly in the dataset by using `NA`.
- Recommended approach is to use commas to separate the fields. By importing a CSV file we don't have any of the problems that can occur when you use a space.
- To avoid typing the file path either set the working directory or use `file.choose()`

Example - `read.table()`

```
data <- read.table("example.csv", header=TRUE, sep=",", skip=7)
str(data) # Gives the structure of data

# Have all character strings not be treated as a factor variable
data <- read.table("example.csv", header=TRUE, sep=",", skip=7,
stringsAsFactors=FALSE)
str(data)

# Have character strings in selected columns not be treated as a factor
data <- read.table("example.csv",header=TRUE,sep=",", skip=7, as.is="dr")
str(data)

# Have the start date be a Date object
data <- read.table("example.csv",header=TRUE,sep=",", skip=7, as.is="dr")
Start.Date <- as.Date(data$d.start, "%m/%d/%Y")
cbind(data[,-7], Start.Date)

# Load data using file.choose()
data <- read.table(file.choose(), header=TRUE, sep=",", skip=7)
```

Attaching a Dataframe

- The function `search()` displays the search path for `R` objects. When `R` looks for an object it first looks in the global environment then proceeds through the search path looking for the object. The search path lists attached dataframes and loaded libraries.
- The function `attach()` (`detach()`) attaches (detaches) a dataframe to the search path. This means that the column names of the dataframe are searched by `R` when evaluating a variable, so variables in the dataframe can be accessed by simply giving their names.

```
head(CO2)         # CO2 dataset, carbon dioxide uptake in grass plants
mean(CO2$uptake)  # Average uptake

search()          # Search path

attach(CO2)       # Attach dataset
search()          # If R can find the variable in the global environment
                  # it first looks in CO2

mean(uptake)      # Average uptake

detach(CO2)       # Detach dataset
search()
```

Caution with Attaching a Dataframe

- `attach()` is okay if you are just working on one dataset and your purpose is mostly on analysis, but if you are going to have several datasets and lots of variables avoid using `attach()`.
- If you attach a dataframe and use simple names like `x` and `y`, it is very possible to have very different objects with the same name which can cause problems
- `R` prints a warning message if attaching a dataframe causes a duplication of one or more names.
- Several modeling functions like `lm()` and `glm()` have a `data` argument so there is no need to attach a dataframe
- For functions without a `data` argument use `with()`. This function evaluates an `R` expression in an environment constructed from the dataframe.
- If you do use `attach()` call `detach()` when you are finished working with the dataframe to avoid errors.

Example - Caution with `attach()`

```
# Type defined in the global environment
Type <- gl(2, 5, labels=c("Type A", "Type B"))

attach(CO2) # Attach CO2
Type        # Get Type from global environment (first in search())
            # instead of from CO2
detach(CO2)

# Use with instead of attach, can also be simpler than using the $
with(CO2, table(Type, Plant))
with(CO2, boxplot(uptake~Treatment))

# Some modeling functions have a data argument
lm(uptake ~ Treatment, data=CO2)
```

Importing datasets using `scan()`

- The `scan()` function is very flexible, but as a result is also harder to use then `read.table()`. However, `scan()` can be useful when you need to import odd datasets.
- For example, suppose we want to input a dataset that has a different number of values on each line.

```
# Determine the number of lines, by using the new-line character
# as a field separator
lines <- length(scan("oddData.txt", sep="\n"))

# Read one line at a time, skipping to the current line
sapply(0:(lines-1), function(x) scan("oddData.txt", skip=x, nlines=1))
```

Exporting Datasets

```
write.table(x, file="", sep=" ", row.names=TRUE,
            col.names=TRUE)
```

`x`	The object to be saved, either a matrix or dataframe
`file`	File name
`sep`	Field separator
`row.names`	Logical, include row.names
`col.names`	Logical, include col.names

- There is also a wrapper function, `write.csv()` for creating a CSV file by calling `write.table()` with `sep=","`.

```
# Export R dataframe as a CSV file
write.table(data, "export.example.csv", sep=",", row.names=FALSE)
write.table(data, file.choose(new=TRUE), sep=",", row.names=FALSE)
```

Exporting Console Output

- The function `capture.output()` evaluates its argument and either returns a character string or sends the output to a file
- Use `sink()` to divert all output to a file

```
x <- rnorm(100)
y <- rnorm(100)
fit <- lm(y~x)

capture.output(summary(fit))
capture.output(summary(fit), file="lm.out.txt")

# Send all output to a file (not expressions)
sink("lm.sink.txt")   # Start sink
mean(x)
summary(fit)
sink()                # Stop sink
```

Sorting

- Sort a dataframe by row using order() and [ ]

```
attach(CO2)   # Since we will only be dealing with variables in CO2

# order() returns a vector of positions that correspond to the
# argument vector, rearranged so that the elements are sorted
order(uptake)
uptake[order(uptake)]   # Same as sort(uptake)

# Sort CO2 by uptake, descending
CO2[order(uptake),]

# Sort CO2 by uptake, ascending
CO2[rev(order(uptake)),]
CO2[order(uptake, decreasing=TRUE),]

# Sort CO2 by conc then uptake
CO2[order(conc, uptake),]

# Sort CO2 by conc (ascending) then uptake (descending), only for numeric data
CO2[order(conc, -uptake),]

# Sort CO2 by Plant, an ordered factor variable
CO2[order(Plant),]

# Sort CO2 by Plant (descending) then uptake (ascending)
# rank() converts Plant to numeric
CO2[order(-rank(Plant), uptake),]
```

Duplicates

- The function `unique()` will return a dataframe with the duplicate rows or columns removed. Use the `incomparables` argument to specify what values not to compare
- `duplicated()` returns a logical vector indicating which rows are duplicates
- **NOTE!** `unique()` and `duplicated()` only work for imported dataframes and does not work for dataframes created during an `R` session

```
# Load Data
data <- read.table("example.csv",header=TRUE, sep=",", skip=8, as.is="dr")
Start.Date <- as.Date(data$d.start, "%m/%d/%Y")
data <- cbind(data[,-7], Start.Date)

unique(data)                 # Dataset with duplicate rows removed
data[duplicated(data),]      # Duplicate rows
```

Matching

- The `match()` function can be used to select the first entry of a variable.

```
# Load Data
data <- read.table("example.csv",header=TRUE,sep=",", skip=7, as.is="dr")
visit.date <- as.Date(data$visit, "%m/%d/%Y")
data <- cbind(data[,-2], visit.date)

# Remove duplicates
data.nodup <- unique(data)

# Sort by ID then visit date so that the first entry for each patient
# is the first visit
data.nodup.sort <- with(data.nodup, data.nodup[order(ID, visit.date),])

# Get the first visit of each patient
first <- with(data.nodup.sort, match(unique(ID), ID))
data.nodup.sort[first,]
```

Merging

- Use `merge()` to merge two and only two dataframes

```
# Merge two datasets by an ID variable, where ID is the same for both datasets
data1 <- data.frame(ID=1:5, x=letters[1:5])
data2 <- data.frame(ID=1:5, y=letters[6:10])
merge(data1, data2)
```

```
# Merge two datasets by an ID variable, where ID is not the same for both datasets
data1 <- data.frame(ID=1:5, x=letters[1:5])
data2 <- data.frame(ID=4:8, y=letters[6:10])
merge(data1, data2)
merge(data1, data2, all=TRUE)
merge(data1, data2, all.x=TRUE) # Only keep the rows from the 1st argument data1
merge(data1, data2, all.y=TRUE) # Only keep the rows from the 2nd argument data2
```

```
# Merge two datasets by an ID variable, where both dataset have the same names
data1 <- data.frame(ID=1:5, x=letters[1:5])
data2 <- data.frame(ID=1:5, x=letters[6:10])
merge(data1, data2, all=TRUE)      # Add rows
merge(data1, data2, by="ID")       # Add columns
merge(data1, data2, by="ID", suffixes=c(1, 2))
```

```
# Merge two datasets by an ID variable, where the ID variable has a different name
data1 <- data.frame(ID1=1:5, x=letters[1:5])
data2 <- data.frame(ID2=1:5, x=letters[6:10])
merge(data1, data2, by.x="ID1", by.y="ID2")
```

Reshaping

- Use reshape() to transform data from wide format to long format and vice versa.

```
# Reshape a wide dataset into long format
# The variable names t.1 and t.2 are intentional, use ".1" and ".2" suffixes
# to make the reshaping much easier
wide <- data.frame(ID=1:5,x=c(10,20,30,40,50),t.1=letters[1:5],t.2=letters[22:26])
reshape(wide, varying=c("t.1", "t.2"), direction="long")
reshape.long <- reshape(wide, varying=c("t.1", "t.2"), direction="long",
                        timevar="visit", idvar="index", drop="ID")

# A reshaped dataset contains attributes that make it easy to move
# between long and wide format, after the first reshape
reshape(reshape.long, direction="wide")

# Reshape a long dataset into wide format
long <- data.frame(ID=rep(1:5, 2), x=rep(c(10,20,30,40,50), 2),
                      t=c(letters[1:5], letters[22:26]), time=rep(1:2, each=5))
reshape(long, idvar="ID", v.names="t", timevar="time", direction="wide")
reshape.wide <- reshape(long, idvar="ID", v.names=c("t", "x"), timevar="time",
                        direction="wide")

# Easily go back to wide format
reshape(reshape.wide, direction="long")
```

Stacking/Unstacking

- The `stack()` function concatenates multiple vectors from separate columns of a dataframe into a single vector along with a factor indicating where each observation originated; `unstack()` reverses this operation.

```
data <- data.frame(label=rep(c("A", "B", "C"), 3), value=sample(9))

# First argument is an object to be stacked or unstacked
# Second argument is a formula, "values to unstack" ~ "groups to create"
uns <- unstack(data, value~label)

# Re-stack
stack(uns)

# Select which columns to stack
data <- data.frame(A=sample(1:9, 3), B=sample(1:9, 3), C=sample(1:9, 3))
stack(data, select=c("A", "B"))
```

Missing Data

- Use `na.omit()` to remove missing data from a dataset and use `na.fail()` to signal an error if a dataset contains `NA`
- `complete.cases()` returns a logical vector indicating which rows have no missing data

```
data <- data.frame(x=c(1,2,3), y=c(5, NA, 8))
na.omit(data) # Remove all rows with missing data

# Use na.fail to test if a dataset is complete
NeedCompleteData <- function(data) {
      na.fail(data)  # Return an error message if missing data

lm(y, x, data=data)
}
NeedCompleteData(data)

sum(complete.cases(data))  # Get the number of complete cases
sum(!complete.cases(data)) # Get the number of incomplete cases
```