Data analysis in R

IML Machine Learning Workshop, CERN

Andrew John Lowe
Wigner Research Centre for Physics,
Hungarian Academy of Sciences

Downloading and installing R

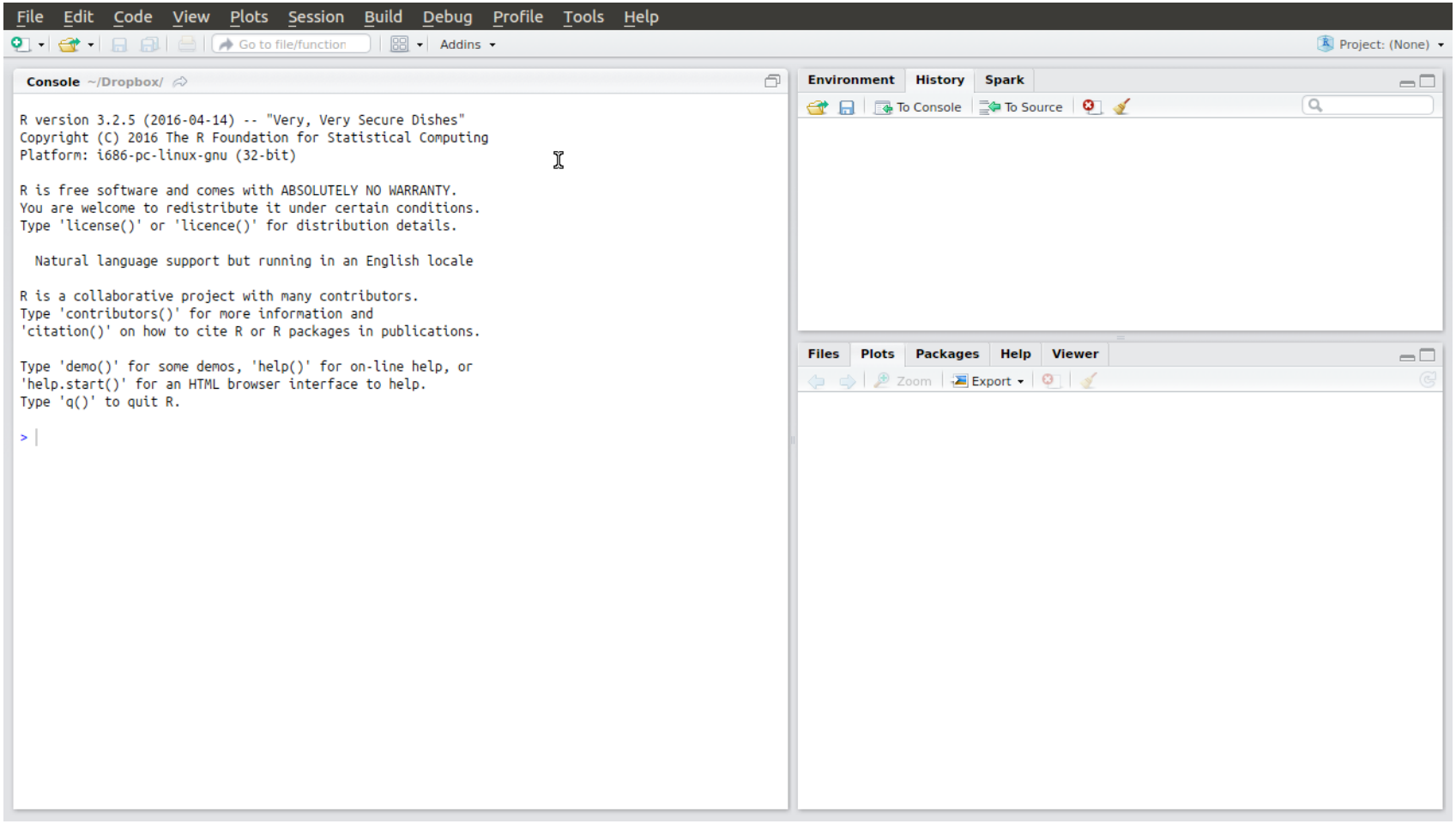
You can follow this tutorial while I talk

1. Download R for Linux, Mac or Windows from the Comprehensive R Archive Network (CRAN): `https://cran.r-project.org/`
   - Latest release version 3.3.3 (2017-03-06, "Another Canoe")

RStudio is a free and open-source integrated development environment (IDE) for R

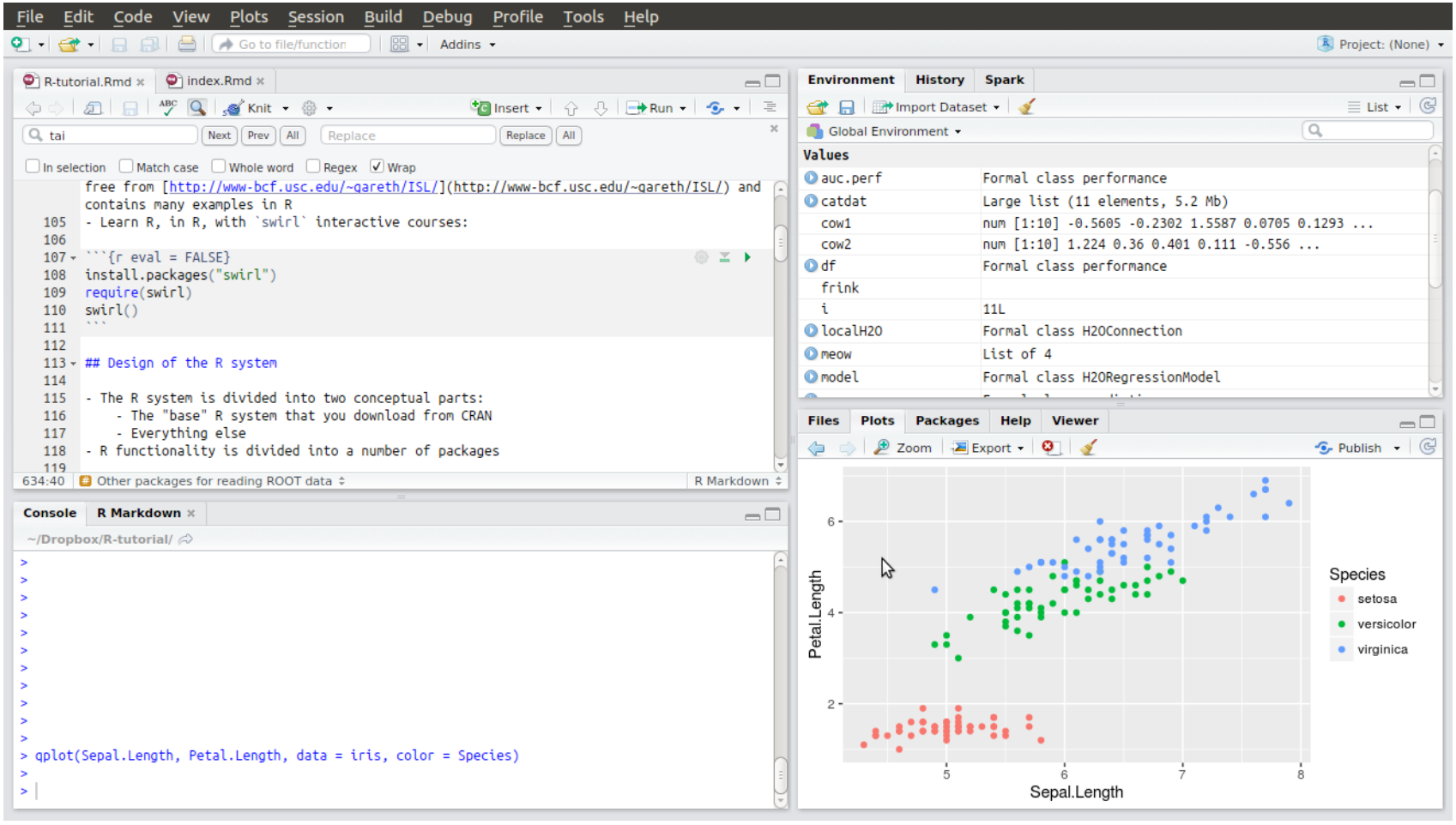
2. Download RStudio from `https://www.rstudio.com/`
   - Make sure you have installed R first!
   - Get RStudio Desktop (free)
   - Latest release version 1.0.136

Rstudio



RStudio looks something like this.

RStudio



RStudio with source editor, console, environment and plot pane.

What is R?

- An open source programming language for statistical computing
- R is a dialect of the S language
- S is a language that was developed by John Chambers and others at Bell Labs
- S was initiated in 1976 as an internal statistical analysis environment — originally implemented as FORTRAN libraries
- Rewritten in C in 1988
- S continues to this day as part of the GNU free software project
- R created by Ross Ihaka and Robert Gentleman in 1991
- First announcement of R to the public in 1993
- GNU General Public License makes R free software in 1995
- Version 1.0.0 release in 2000

Why use R?

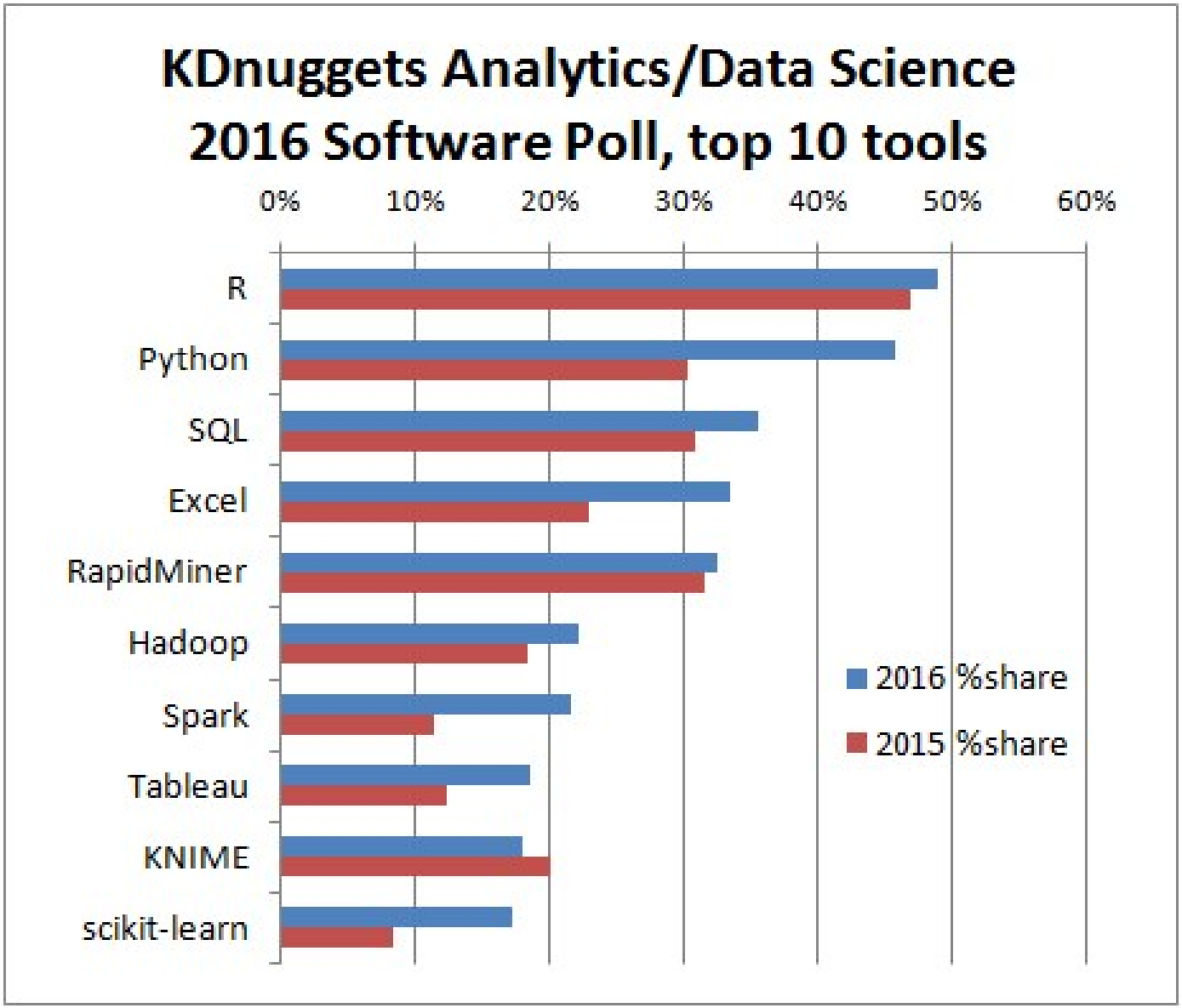
- Free
- Runs on almost any standard computing platform/OS
- Frequent releases; active development
- Very active and vibrant user community
  - Estimated ~2 million users worldwide
  - R-help and R-devel mailing lists, Stack Overflow
  - Frequent conferences; useR!, EARL, etc.
  - 388 Meetup groups worldwide
  - R-Ladies chapters in many major cities
- Big-name backing: Microsoft, Google, IBM, Oracle, ...
- Functionality is divided into modular packages
  - Download and install just what you need
  - There are now $>$ 10000 packages on CRAN
- Graphics capabilities very sophisticated
- Useful for interactive work, but contains a powerful programming language for developing new tools

Any other reasons for using R?

- R enables fast prototyping and high-level abstractions that let you concentrate on what you want to achieve, rather than on the mechanics of how you might do it
  - Enables you to stay "in the flow" of data analysis
- Latest machine learning algorithms are available
- Your technical questions have probably already been answered on Stack Overflow
- R offers a pleasant user experience

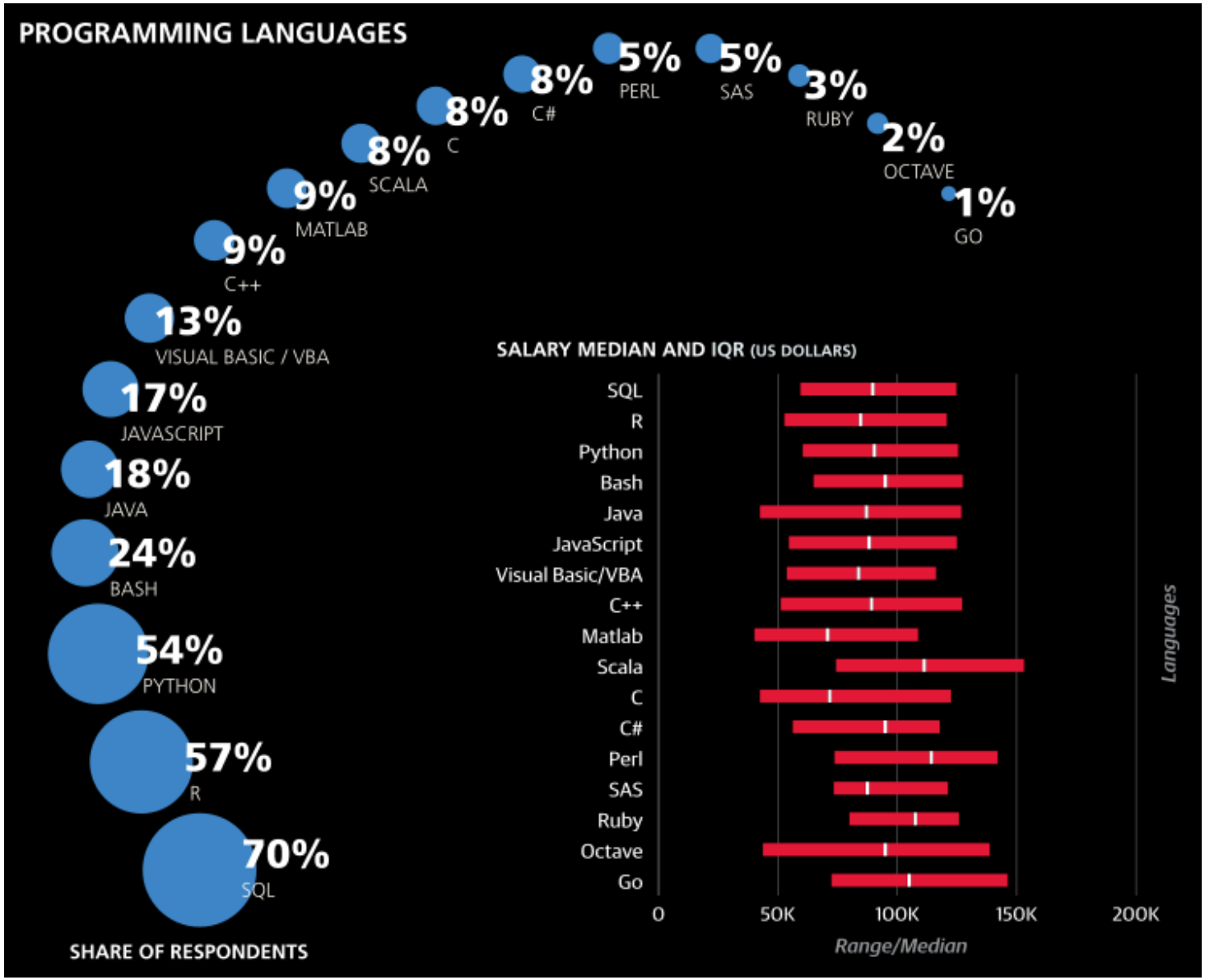
R allows you to concentrate on your data, not on your tools

KDNuggets 2016 Advanced Analytics Survey



KDnuggets: R remains leading tool, but Python usage growing very fast.

O'Reilly 2016 Data Science Salary Survey



O'Reilly: SQL is king, but R and Python very popular.

Drawbacks of R

- Essentially based on 40-year-old technology
- Functionality is based on consumer demand and user contributions
- Objects must generally be stored in physical memory
  - Your data must fit within the contiguous RAM of your hardware
  - True of other software such as scikit-learn, WEKA, and TMVA
- There have been advancements to deal with this, including:
  - Interfaces to Spark and Hadoop
  - Packages `ff` and `bigmemory`
  - File-backed data objects
- Big RAM is eating Big Data!
  - You can now get 2 TB X1 instances on Amazon EC2
  - Google and Microsoft also offer instances with large RAM
  - Trend driven by companies who increasingly are replacing local hardware with cloud computing and need massive compute resources

Resources for learning R

- Online courses on Coursera, edX, DataCamp, and elsewhere
- For machine learning specifically, the book ***Introduction to Statistical Learning*** (Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani) can be downloaded for free from `http://www-bcf.usc.edu/~gareth/ISL/` and contains many examples in R
- Learn R, in R, with `swirl` interactive courses:

```
install.packages("swirl")
require(swirl)
swirl()
```

Design of the R system

- The R system is divided into two conceptual parts:
  - The "base" R system that you download from CRAN
  - Everything else
- R functionality is divided into a number of packages

Where to get packages

- CRAN `https://cran.r-project.org`
- GitHub
- Bioconductor `https://www.bioconductor.org/`
  - Mostly bioinformatics and genomics stuff
- Neuroconductor `https://www.neuroconductor.org/`
  - The new kid on the block: computational imaging software for brain imaging
- RForge `http://rforge.net/`
  - Not so well known
- Some .tar.gz file you downloaded from a website
  - Use caution!

Installing from CRAN

```
install.packages("devtools") # Install it
require(devtools) # Load it
library(devtools) # Alternatively, load like this
```

You might need to specify a repository:

```
install.packages("devtools",
                 repos = "http://stat.ethz.ch/CRAN/")
```

List of mirrors here:
https://cran.r-project.org/mirrors.html

Installing from Bioconductor and Neuroconductor

```
source("https://bioconductor.org/biocLite.R")
biocLite()
biocLite("GenomicFeatures") # For example
```

```
source("https://neuroconductor.org/neurocLite.R")
neuro_install(c("fslr", "hcp")) # Install these two
```

Installing from GitHub

```
install.packages("devtools") # Install devtools first!
require(devtools) # Load devtools
install_github("mlr-org/mlr") # Install mlr
require(mlr) # Load mlr
```

Tip: What if the package you want has been removed from, say, CRAN or Bioconductor? Both have read-only mirrors of their R repositories. These mirrors can be a useful alternative source for packages that have disappeared from their R repositories.

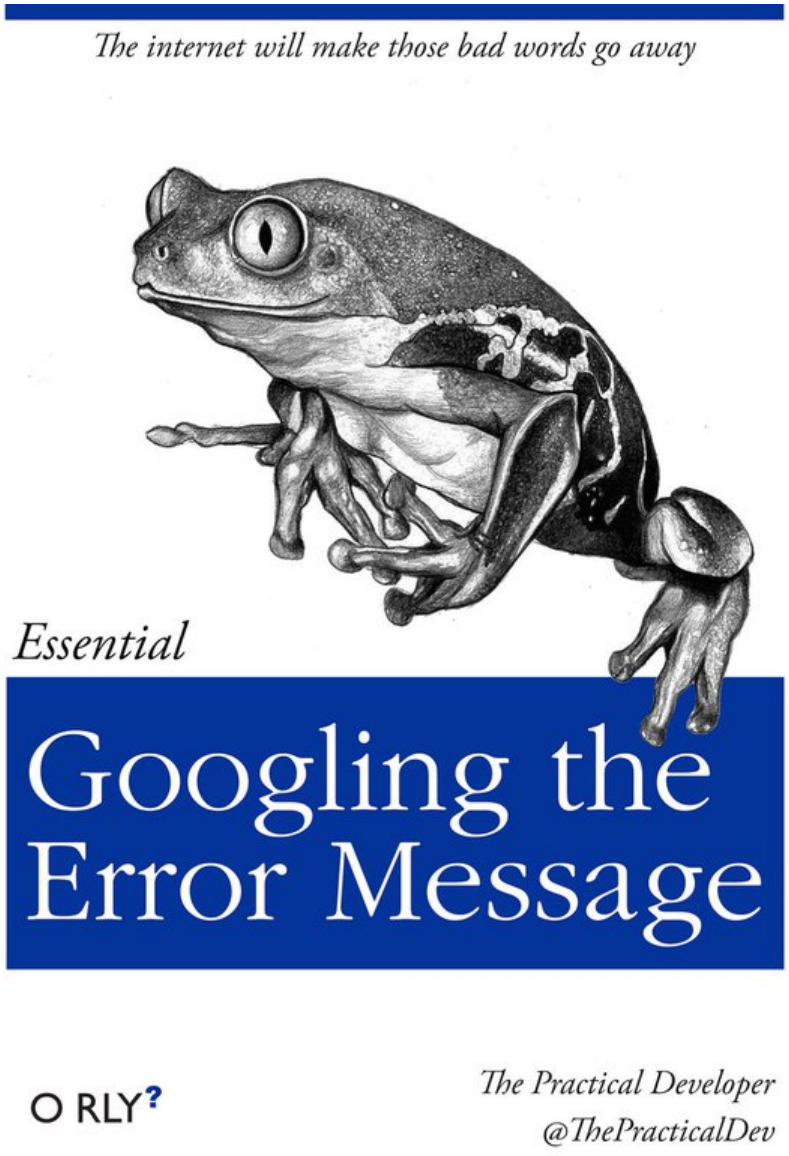
How to get help

- Typing `?command` will display the help pages for `command`, *e.g.*:

```
# This will display help on plotting of R objects:
?plot
```

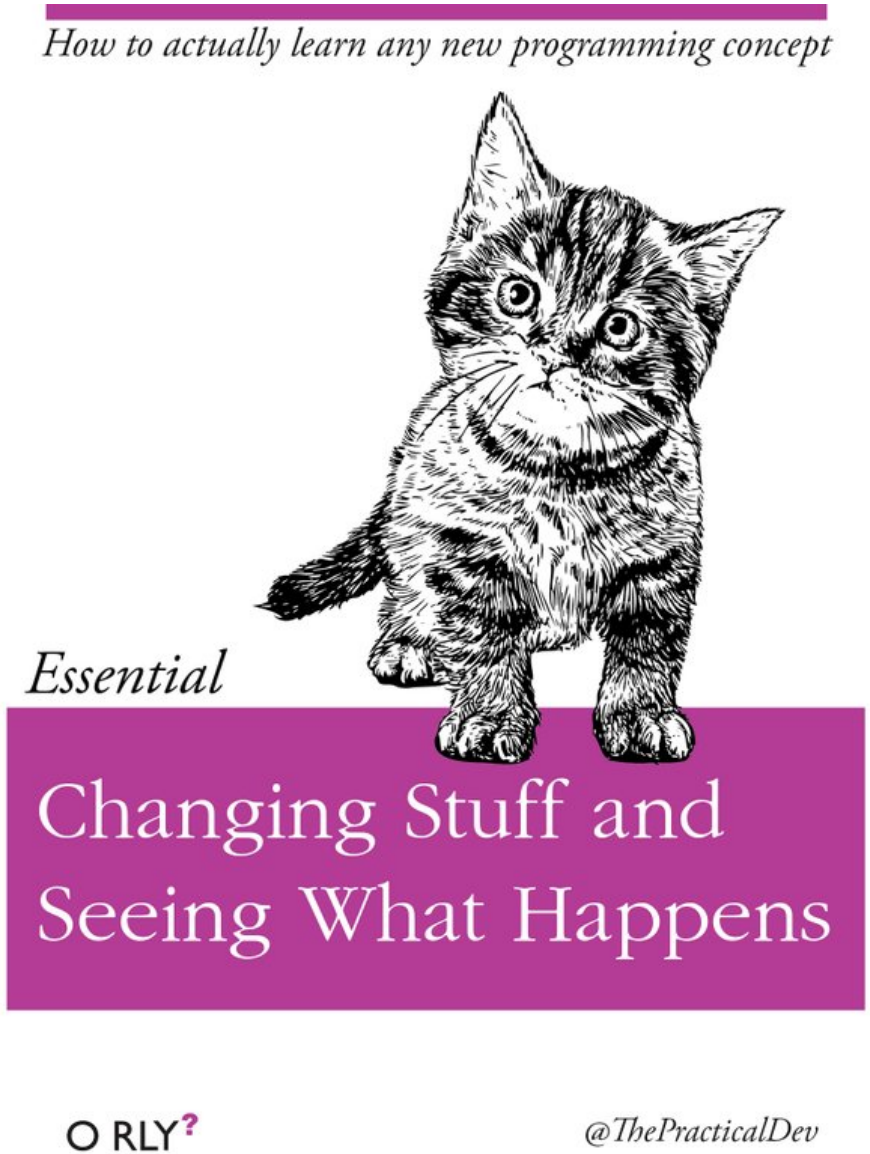
- RTFM! Google for package reference manuals on CRAN
- CRAN packages usually come with example code; usually found at the bottom of the help pages or on the CRAN webpage for the package (search for "Vignettes")
- CRAN Task Views: view packages for particular tasks `https://cran.r-project.org/web/views/`
- Click *Help* on the RStudio toolbar for cheatsheets and quick reference guides

Other ways for getting help



I recommend this.

A good way to learn R



Another gripping page-turner.

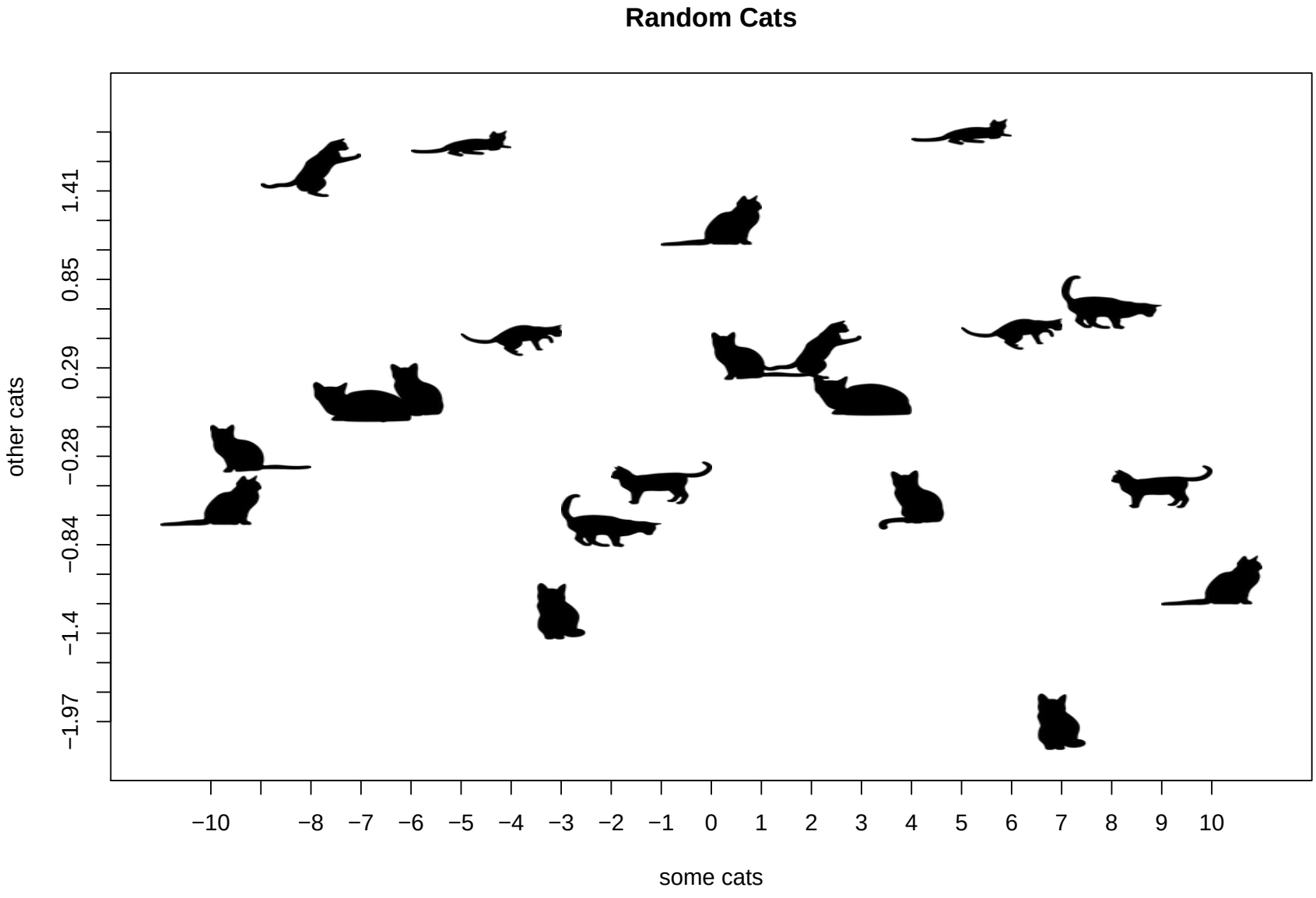
Packages for fun/strange things you can do with R

- `twitteR`: send tweets (or do sentiment analysis)
- `ubeR`: call an Uber
- `rpushbullet`: send a notification to your phone
  - ("Your code finished running!")
- `predatory`: keep track of predatory publishers
- `xkcd`: draw your plots in *xkcd* style
- `remoji`: plot using emoji
- `magick`: advanced image processing
- `catterplots`: plot using cats

Image processing with magick

Obviously, you can replace Prof. Frink with (for example) the logo for your institute or experiment.

Meow


Random Cats
other cats
1.41
0.85
0.29
−0.28
−0.84
−1.4
−1.97
−10
−8
−7
−6
−5
−4
−3
−2
−1
0
1
2
3
4
5
6
7
8
9
10
some cats

Fundamentals

Data types

- R has five basic or "atomic" classes of objects:
  - character
  - numeric (real numbers)
  - integer
  - complex
  - logical (Boolean `TRUE`/`FALSE` or `T`/`F` in abbreviated form)
- The most basic object is a vector
- Vectors are homogeneous containers (elements all same type)

Numbers

- By default, numbers are objects of the class numeric
- If you explicitly want an integer, you need to specify the `L` suffix: `42L`
- Special numbers:
  - `Inf`: infinity
  - `NaN`: undefined value; "not a number", *e.g.* 0/0. Another example: p_{T} of subleading jet in an event with only one jet can be represented with a NaN. Arithmetic with a NaN value results in a NaN. NaNs can also be thought of as representing a missing value.
- Get numerical characteristics of your machine:

```
.Machine
# For information on interpreting the output:
?.Machine
```

Attributes

R objects can have attributes:

- Names
- Dimensions (*e.g.*, *matrices*)
- Length (*e.g.*, *vectors*)
- Class

```
# Create an empty matrix with 2 columns and 3 rows:
m <- matrix(ncol = 2, nrow = 3)
# Dimensions:
dim(m)
```

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

```
class(m)
```

```
## [1] "matrix"
```

Evaluation

- When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.
- <- is the assignment operator (you might also see = in code)
- Tip: keyboard shortcut in RStudio is `Alt+-` (Windows/Linux) or `Option+-` (Mac)

```
# This is a comment!
x <- 42 # Nothing printed
x # Auto-printed
```

```
## [1] 42
```

```
print(x) # Explicit printing
```

```
## [1] 42
```

Sequences

```
x <- 1:5 # Create integer sequence
x
```

```
## [1] 1 2 3 4 5
```

```
x <- seq(from = 0, to = 1, by = 0.2)
x
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Vectors

The `c()` ("concatenate") function can be used to create vectors of objects:

```
x <- c(0.3, 0.5) # Numeric
x <- c(TRUE, FALSE) # Logical
x <- c(T, F) # Logical
x <- c("A", "B", "C") # Character
x <- c(1+5i, 3-2i) # Complex
```

Using the `vector()` function:

```
x <- vector("numeric", length = 10)
x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Mixing objects

```
y <- c(1.7, "a") # y is character
y <- c(TRUE, 2) # y is numeric
y <- c("a", TRUE) # y is character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

Explicit coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available:

```
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Explicit coercion

Nonsensical coercion results in `NA`s (missing values):

```
x <- c("a", "b", "c")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

Matrices

Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol):

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(m)
```

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

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns.

Reshaping matrices

Matrices can also be created directly from vectors by adding a dimension attribute:

```
v <- 1:10
v
```

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

```
dim(v) <- c(2, 5)
v
```

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

We can also reshape matrices (unroll them) using the same method.

cbind-ing and rbind-ing

Matrices can be created by column-binding or row-binding with `cbind()` and `rbind()`:

```
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x, y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Lists

Lists are a special type of vector that can contain elements of different classes:

```
x <- list(1, "a", TRUE, 1 + 4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*:

- `signal/background`
- `bjet/light`
- `male/female`
- `benign/malignant`
- `low/middle/high`
- *etc.*

Similar to `enums` in C++

Factors

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level:

```
x <- factor(c("signal", "signal",
              "background", "signal"))
x
```

```
## [1] signal     signal     background signal
## Levels: background signal
```

```
x <- factor(c("signal", "signal",
              "background", "signal"),
            levels = c("signal", "background"))
x
```

```
## [1] signal     signal     background signal
## Levels: signal background
```

Factors

```
unclass(x)
```

```
## [1] 1 1 2 1
## attr(,"levels")
## [1] "signal"     "background"
```

Missing values

Missing values are denoted by `NA` or `NaN` for undefined mathematical operations:

```
x <- 0/0
is.na(x) # Is x NA?
```

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

```
is.nan(x) # Is x NaN?
```

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

- A `NaN` value is also `NA` but the converse is not true
- Systematic use of NaNs in programming languages was introduced by the IEEE 754 floating-point standard in 1985

Data frames

Data frames are used to store tabular data, and are like pandas DataFrames and similar to ROOT Ntuples

```
x <- data.frame(foo = 1:3, bar = c(T, T, F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
```

```
colnames(x) # Names of the variables
```

```
## [1] "foo" "bar"
```

```
rownames(x) # To identify specific observations
```

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

Names

R objects can also have names, which is very useful for writing readable code and self-describing objects:

```
x <- 1:3
names(x) <- c("foo", "bar", "qux")
x
```

```
## foo bar qux
##   1   2   3
```

```
m <- matrix(1:4, nrow = 2, ncol = 2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m
```

```
##   c d
## a 1 3
## b 2 4
```

Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- `[` always returns an object of the same class as the original; can be used to select more than one element
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
- `$` is used to extract elements of a list or data frame by name; semantics are similar to that of `[[`
- The `[[` operator can be used with computed indices; `$` can only be used with literal names
- The `[[` operator can take an integer sequence
- Partial matching of names is allowed with `[[` and `$`

Subsetting

```r
x <- c("A", "B", "C", "D", "E")
x[1] # This is the FIRST element!
```

```
## [1] "A"
```

```r
x[1:5]
```

```
## [1] "A" "B" "C" "D" "E"
```

```r
x[x > "A"]
```

```
## [1] "B" "C" "D" "E"
```

Subsetting a matrix

```
x <- matrix(1:6, 2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
x[1, 2]
```

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

```
x[1, ] # Note missing index
```

```
## [1] 1 3 5
```

Subsetting lists

```
x <- list(foo = 1:4, bar = 0.6)
x[1]
```

```
## $foo
## [1] 1 2 3 4
```

```
class(x[1])
```

```
## [1] "list"
```

```
x[[1]]
```

```
## [1] 1 2 3 4
```

```
class(x[[1]])
```

```
## [1] "integer"
```

Subsetting lists

```
x$bar
```

```
## [1] 0.6
```

```
class(x["bar"])
```

```
## [1] "list"
```

```
class(x[["bar"]])
```

```
## [1] "numeric"
```

Removing missing values

A common task is to remove missing values (NAs)

```
x <- c(1, 2, NA, 4, NA, 5)
is.na(x)
```

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

```
x[!is.na(x)]
```

```
## [1] 1 2 4 5
```

Removing missing values

```
airquality[3:6, ] # A toy dataset
```

```
##   Ozone Solar.R Wind Temp Month Day
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
good <- complete.cases(airquality) # Logical vector
airquality[good, ][3:6, ] # Remove NAs
```

```
##   Ozone Solar.R Wind Temp Month Day
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

Reading data

There are a few principal functions reading data into R:

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (inverse of dump)
- `dget`, for reading in R code files (inverse of dput)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

There are analogous functions for writing data to files:
`write.table`, `writeLines`, `dump`, `dput`, `save`, `serialize`

read.csv and read.table

The `read.csv` function is one of the most commonly used functions for reading data:

```
data <- read.csv("foo.txt")
```

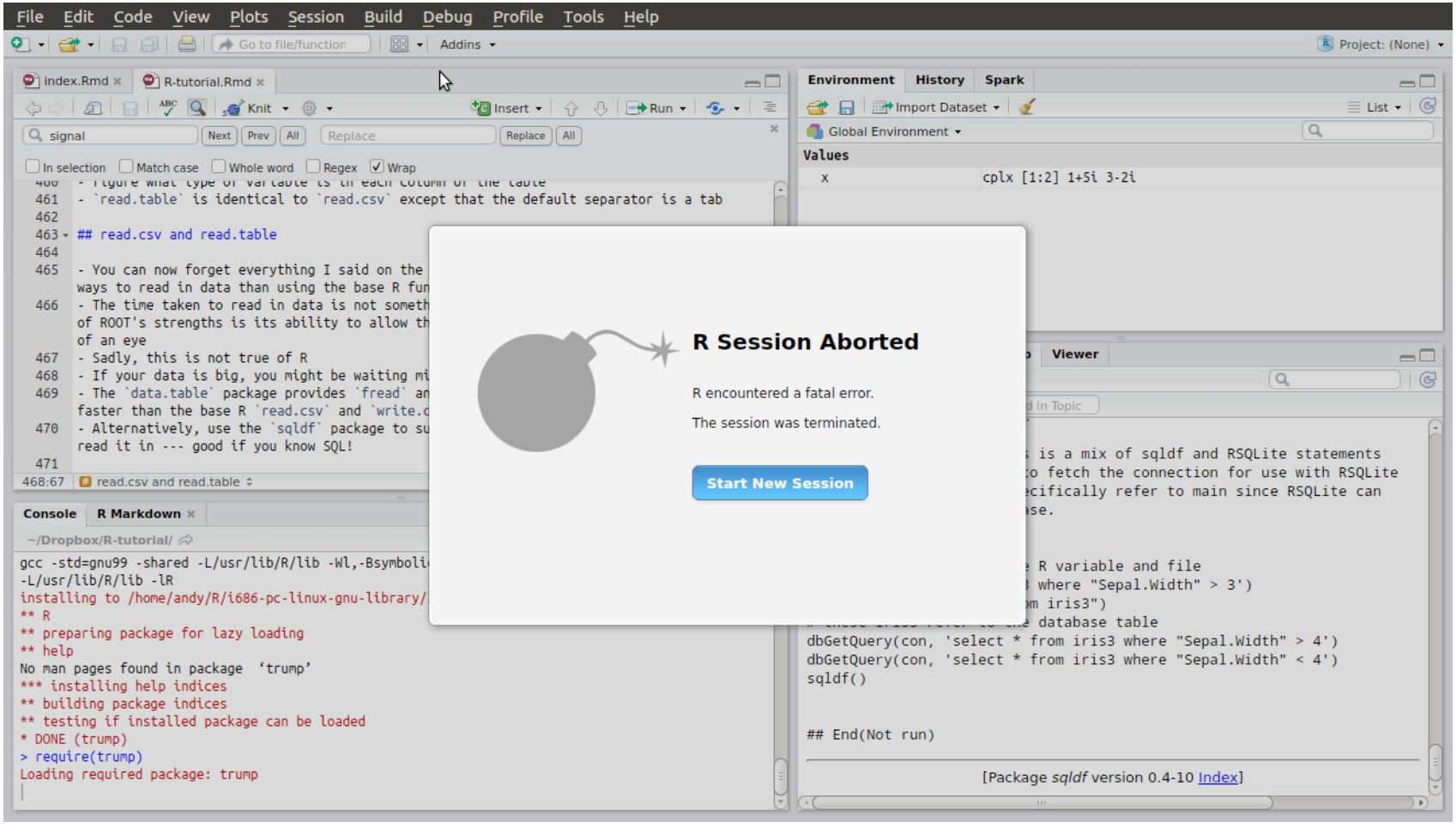
R will automatically

- skip lines that begin with a #
- figure out how many rows there are
- figure what type of variable is in each column of the table
- `read.table` is identical to `read.csv` except that the default separator is a tab

Faster methods for reading and writing data

- The time taken to read in data is not something we normally have to consider in HEP; one of ROOT's strengths is its ability to allow the user access to huge datasets in the blink of an eye
- Sadly, this is not true of R
- If your data is big, you might be waiting minutes or even hours with `read.csv`
- The `data.table` package provides `fread` and `fwrite`; both are often **many times** faster than the base R `read.csv` and `write.csv` methods
- Alternatively, use the `sqldf` package to subset your data with a SQL query before you read it in — good if you know SQL!

Bang!



This is what happens if you try to read in more data than will fit in RAM.

RAM considerations

- Know Thy System
- A 64-bit OS will usually allow you to squeeze more out of RAM than a 32-bit OS
- What other applications are in use?
  - Close those umpteen Chrome tabs!
- You can delete objects you no longer need with `rm(foo)`

Stored Object Caches for R

I use the SOAR package to store objects out of memory in a stored object cache on disk

```
Sys.setenv(R_LOCAL_CACHE=".R_Test") # Store stuff here
dummy <- rnorm(1e6, mean = 2) # 1e6 random normals
ls() # What's in my environment?
```

```
## [1] "dummy"
```

```
Store(dummy) # Cache dummy data
ls() # Gone from RAM:
```

```
## character(0)
```

```
mean(dummy) # But we can read it
```

```
## [1] 1.999484
```

Interfaces to the outside world

Data are read in using connection interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

Type, for example, `?file` to find out more

Reading ROOT data

- At some point, you're going to want/need to do this
- I use Adam Lyon's `RootTreeToR` package
- Debuted at useR! 2007 conference:
  `http://user2007.org/program`
- No longer maintained, sadly
  - Latest version 6 years old
- You need ROOT installed, but no need to run ROOT
  - Launch R/RStudio from a terminal where ROOTSYS is set
- Haven't checked if this works with ROOT 6
- What if this stops working with newer ROOT versions?
  - Trivial to write a stand-alone C++ utility to parse command line strings, like those you would pass to Draw(), and pass them to TTreePlayer and use it to dump out the data into a text file
  - Or use a R↔Python interface package to run `root_numpy`, then use `feather` to write and then read back the data into R

RootTreeToR

```
devtools::install_github("lyonsquark/RootTreeToR")
require(RootTreeToR)
# Open and load ROOT tree:
rt <- openRootChain("TreeName", "FileName")
N <- nEntries(rt) # Number of rows of data
# Names of branches:
branches <- RootTreeToR::getNames(rt)
# Read in a subset of branches (vars), M rows:
df <- toR(rt, vars, nEntries = M) # df: a data.frame
```

Other packages for reading ROOT data

- **AlphaTwirl**: a Python library for summarising event data in ROOT Trees (https://github.com/TaiSakuma/AlphaTwirl)

  "The library contains a set of Python classes which can be used to loop over event data, summarize them, and store the results for further analysis or visualization. Event data here are defined as any data with one row (or entry) for one event; for example, data in ROOT TTrees are event data when they have one entry for one proton-proton collision event. Outputs of this library are typically not event data but multi-dimensional categorical data, which have one row for one category. Therefore, the outputs can be imported into R or pandas as data frames. Then, users can continue a multi-dimensional categorical analysis with R, pandas, and other modern data analysis tools."

- For more details, ask the author Tai Sakuma (he's here)

Control structures

- Structures that will be familiar to C++ programmers include: `if`, `else`, `for`, `while`, `break`, and `return`. `repeat` executes an infinite loop, `next` skips an iteration of a loop.

```
for(i in 1:5) print(letters[i])
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

- Although these structures exist, vectorisation means that loops are not as common as in other languages
- It's usually faster to use a function from the *`apply` family

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class `function`.

```
my_function <- function(foo, bar, ...) {
  # Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated

Function arguments

- R functions arguments can be matched positionally or by name
- In addition to not specifying a default value, you can also set an argument value to `NULL`:

```
f <- function(a, b = 1, c = 2, d = NULL) { # stuff }
```

- Use `...` to indicate a variable number of arguments to pass to an inner function:

```
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...)
}
```

- Useful when the number of arguments passed to the function cannot be known in advance

Inspecting function implementations

- Strip the parentheses from a function to see its implementation:

```
myplot
```

```
## function(x, y, type = "l", ...) {
##   plot(x, y, type = type, ...)
## }
```

Interlude: simplify your code with pipes

%>%

Data pipelines

R is a functional language, which means that your code often contains a lot of parentheses. And complex code often means nesting those parentheses together, which make code hard to read and understand:

```
foo <- do_more_stuff(
  do_something(
    read_some_data(data), some.args,
  )
)
print(foo)
```

Using %>%

- The `dplyr` package provides powerful functions for operating on data, and relies on the %>% pipeline operator (provided by `magrittr`), which takes the results of the previous operation and passes it as the first argument of the next:

```
read_some_data(data) %>% do_something(some.args) %>%
  do_more_stuff %>% print
```

- Unwrap nested function calls (I call this *dematryoshkafication*)
- Data manipulation with %>% mirrors the way we think about processing data: like on a production line, performing actions on an object sequentially, in a stepwise manner
- This results in more readable code, and can be used to reduce the creation of intermediate data objects, which saves RAM
- Takes a mental switch to do analysis this way, but utterly addictive once you get used to it

End of interlude

Vectorised operations

Many operations in R are vectorised:

```
x <- 1:4; y <- 5:8 # Two statements on one line
x * y
```

```
## [1]  5 12 21 32
```

Vectorised matrix operations

```
a <- rep(10, 4) # 10 repeated 4 times
x <- matrix(1:4, 2, 2); y <- matrix(a, 2, 2)
x * y # element-wise multiplication
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```
x %*% y # true matrix multiplication
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```

Loop functions

The *`apply` family of functions enable looping at the command line:

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as lapply but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of lapply

Loop functions: lapply

`lapply` always returns a list, regardless of the class of the input:

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.1878958
```

- The actual looping is done internally in C code

Loop functions: sapply

`sapply` will try to simplify the result of lapply if possible:

```
sapply(x, mean)
```

```
##          a          b
## 3.0000000 0.1878958
```

Loop functions: apply

- `apply` is used to a evaluate a function (often an anonymous one) over the margins of an array
- Usage: `apply(X, MARGIN, FUN, ...)`
  - `X` is an array
  - `MARGIN` is an integer vector indicating which margins should be "retained".
  - `FUN` is a function to be applied
  - `...` is for other arguments to be passed to `FUN`

```
x <- matrix(rnorm(12), 4, 3)
apply(x, 2, mean)
```

```
## [1] -0.002790676  1.185769439  0.339651320
```

Loop functions: apply

A more complicated example:

```
x <- matrix(rnorm(120000), 4, 30000)
apply(x, 1, FUN = function(x) {
  quantile(x, probs = c(0.025, 0.975))
})
```

```
##              [,1]      [,2]      [,3]      [,4]
## 2.5%  -1.942963 -1.943087 -1.969441 -1.967983
## 97.5%  1.971000  1.961710  1.945407  1.958806
```

Some useful functions for exploring your data: str

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 1
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Some useful functions for exploring your data: summary

```
summary(mtcars[1:3])
```

```
##       mpg             cyl             disp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8  
##  Median :19.20   Median :6.000   Median :196.3  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0
```

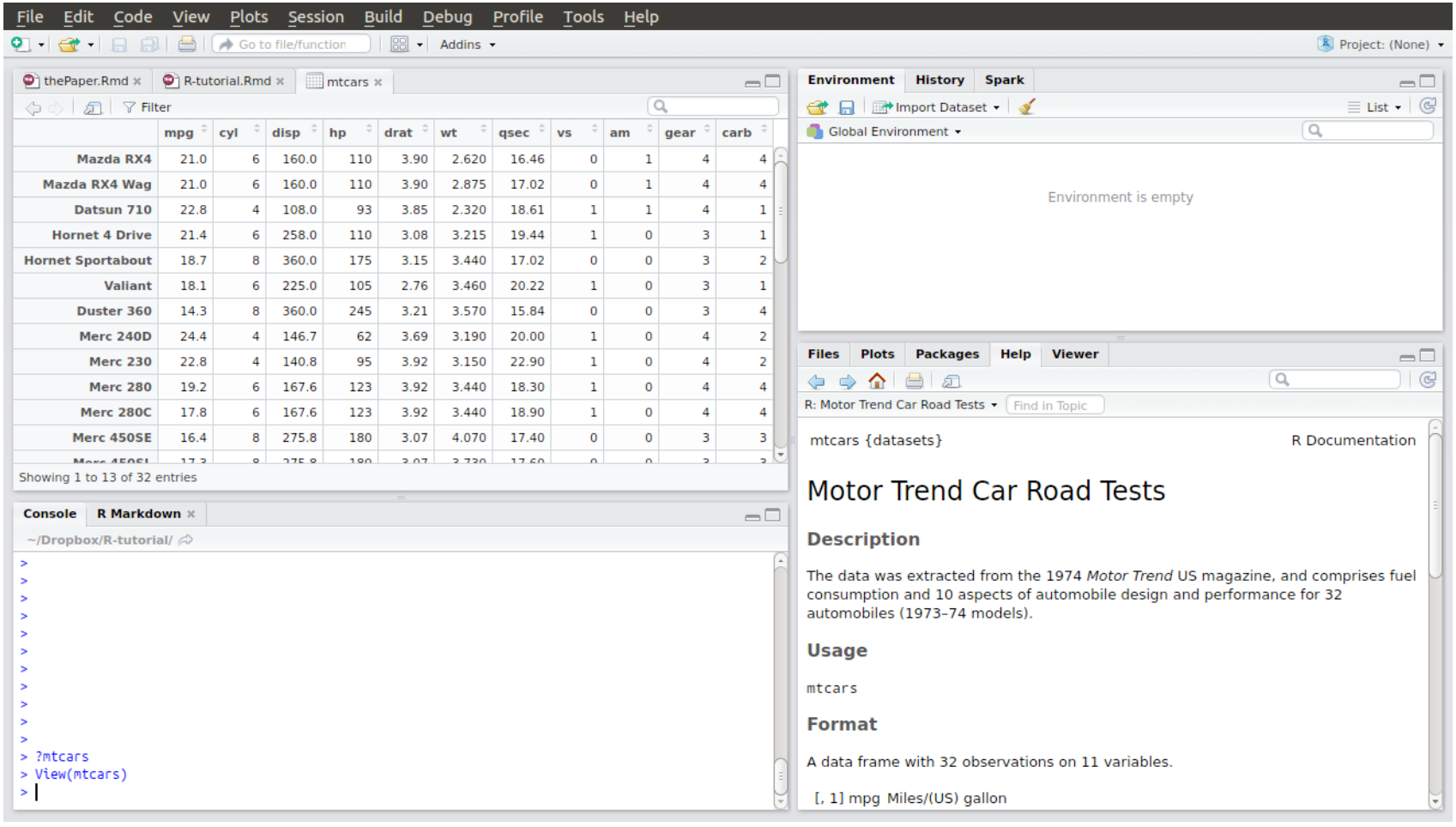
Some useful functions for exploring your data: head

```
head(mtcars[1:5])
```

```
##                    mpg cyl disp  hp drat
## Mazda RX4         21.0   6  160 110 3.90
## Mazda RX4 Wag     21.0   6  160 110 3.90
## Datsun 710        22.8   4  108  93 3.85
## Hornet 4 Drive    21.4   6  258 110 3.08
## Hornet Sportabout 18.7   8  360 175 3.15
## Valiant           18.1   6  225 105 2.76
```

Invoke the RStudio viewer

```
View(mtcars)
```



Simulations

Functions for probability distributions in R

- `rnorm`: generate random Normal variates with a given mean and standard deviation
- `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- `pnorm`: evaluate the cumulative distribution function for a Normal distribution
- `rpois`: generate random Poisson variates with a given rate

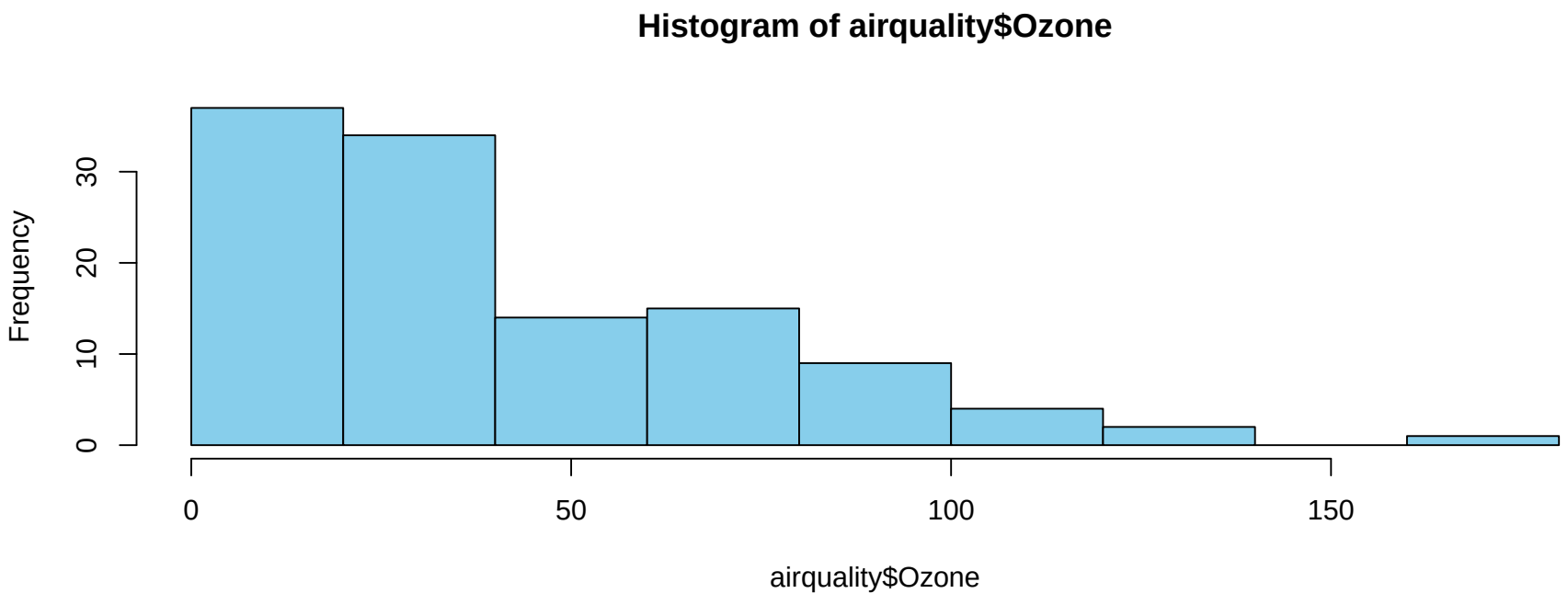
Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

- `d` for density
- `r` for random number generation
- `p` for cumulative distribution
- `q` for quantile function

Graphics

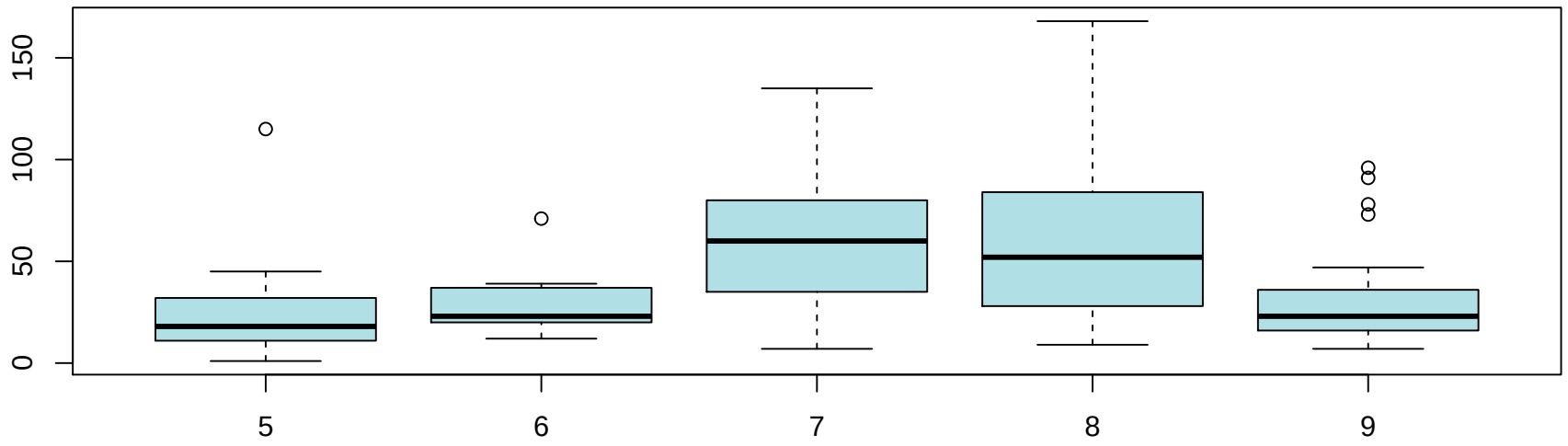
Plotting using R base plots: histograms

```
hist(airquality$Ozone, col = "skyblue")
```



Plotting using R base plots: (Tukey) boxplots

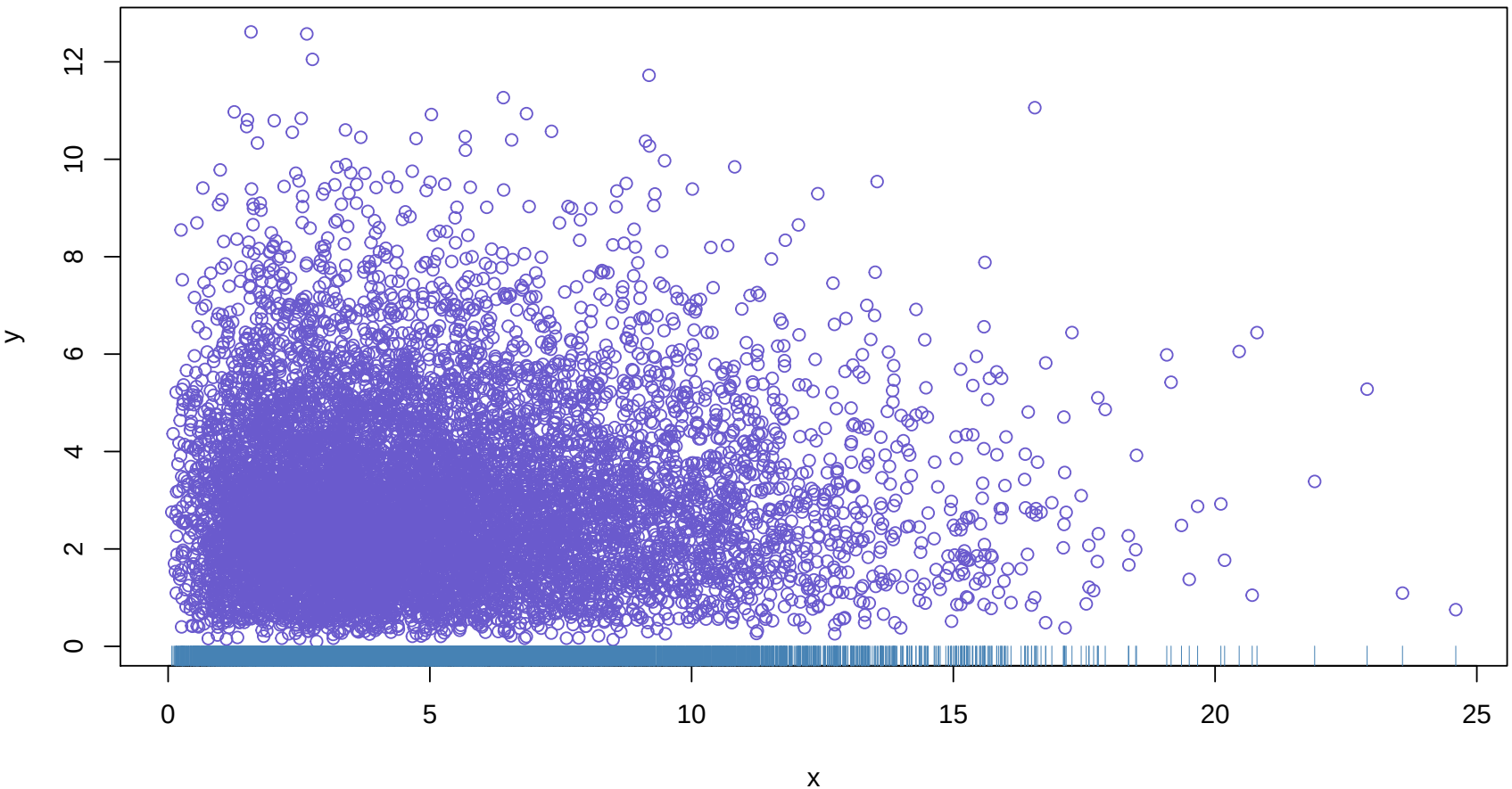
```
airquality <- transform(airquality,
                        Month = factor(Month))
boxplot(Ozone ~ Month, airquality, col = "powderblue")
```



- Box: interquartile range (IQR)
- Line: median
- Whiskers: 1.5×IQR
- Outliers: in a Tukey boxplot, points beyond 1.5×IQR

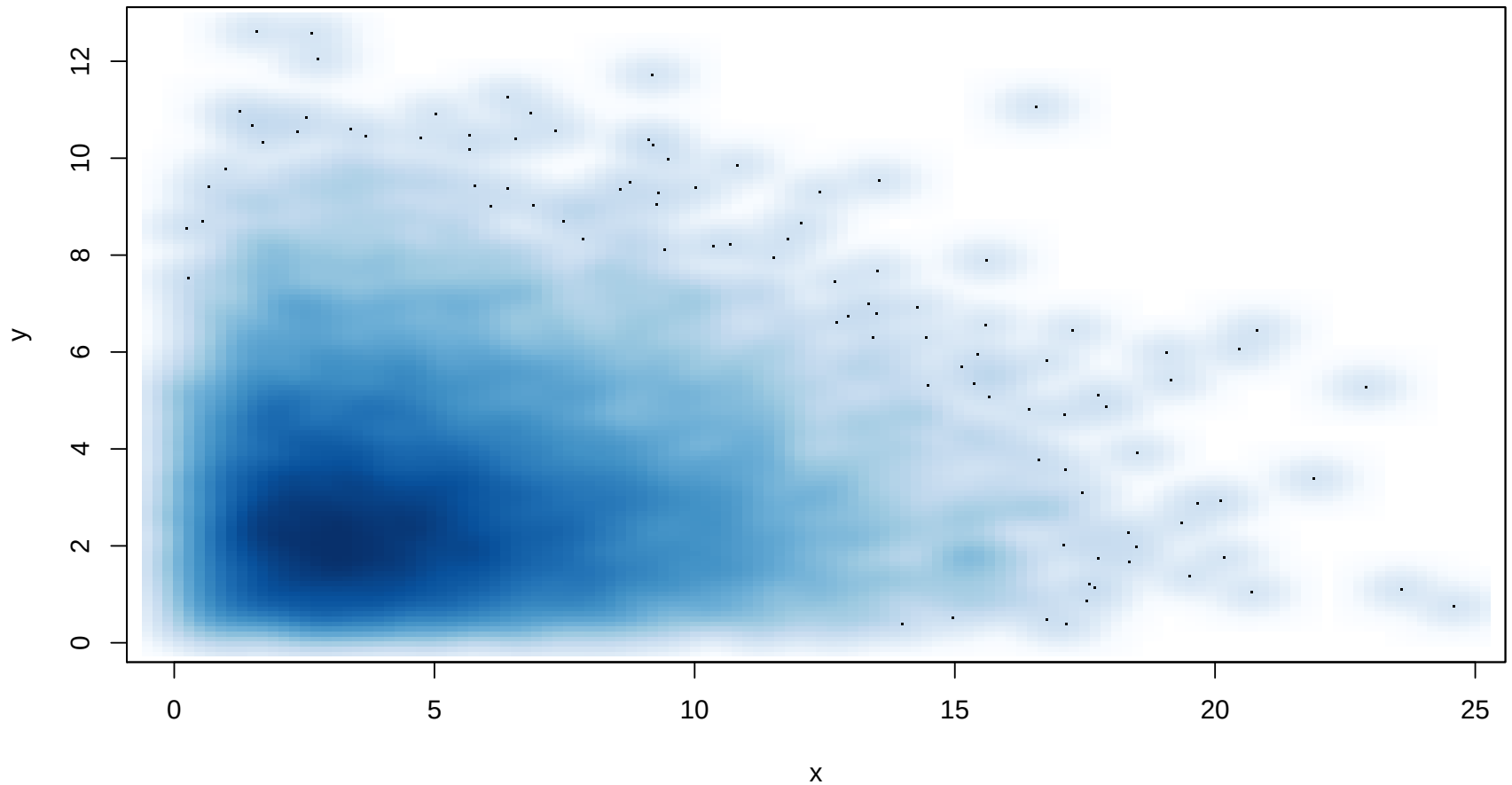
Plotting using R base plots: scatter plots

```
x <- rchisq(10000, df = 5)
y <- rgamma(10000, shape = 3)
plot(x, y, pty = 19, col = "slateblue")
rug(x, col = "steelblue")
```



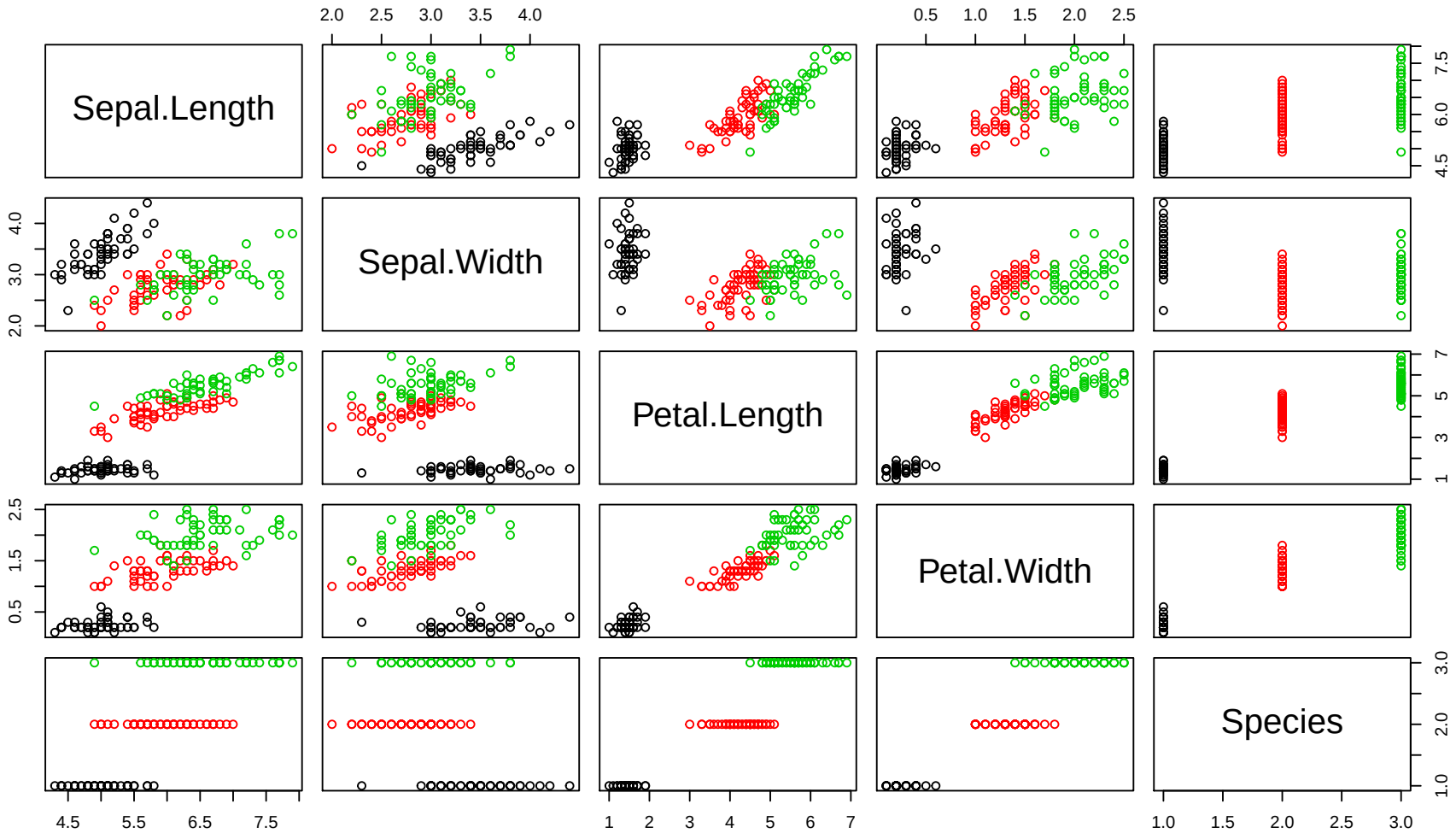
Plotting using R base plots: scatter plots

```
smoothScatter(x, y)
```



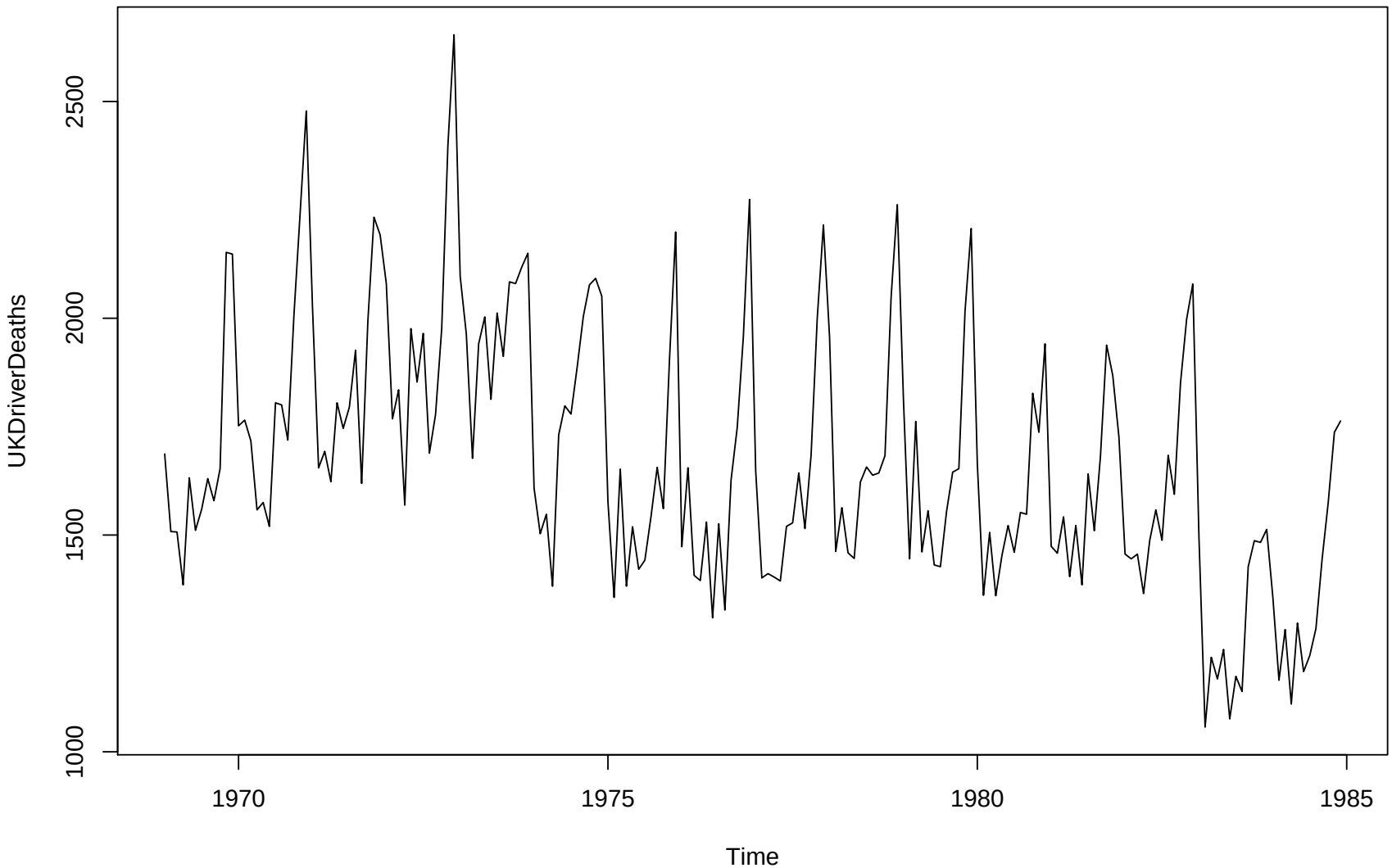
Plotting using R base plots: scatter plot matrices

```
plot(iris, col = iris$Species)
```



Plotting using R base plots: line graphs

```
data("UKDriverDeaths")
plot(UKDriverDeaths)
```

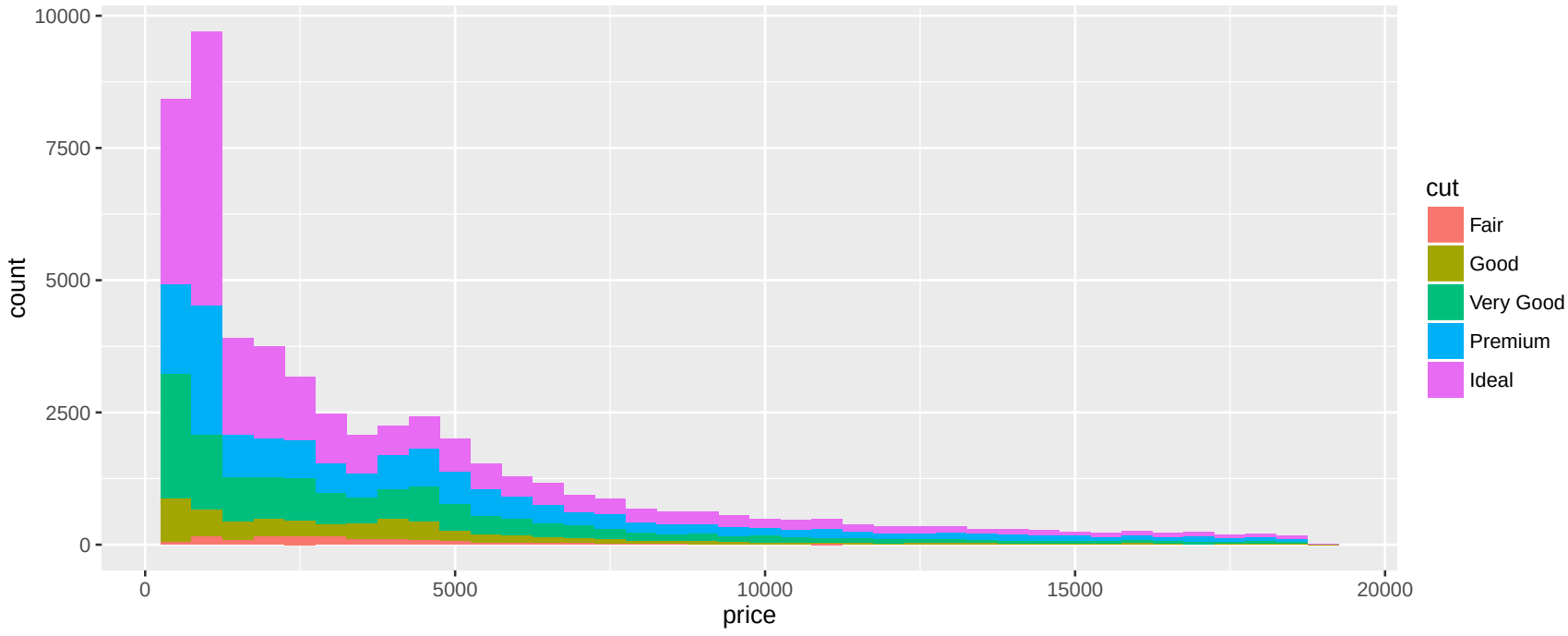


Plotting with ggplot2

- An implementation of the *Grammar of Graphics* by Leland Wilkinson
- Written by Hadley Wickham
- Grammar of graphics represents an abstraction of graphic ideas/objects
- Think "noun", "verb", "adjective" for graphics
- Plots are made up of aesthetics (size, shape, colour) and geoms (points, lines)
- See the ggplot2 cheatsheet (*Help* $\rightarrow$ *Cheatsheets* in RStudio)
- The range of plots you can make is **huge**; I'll show only a small selection
- See my talk in the jet tagging session for more examples

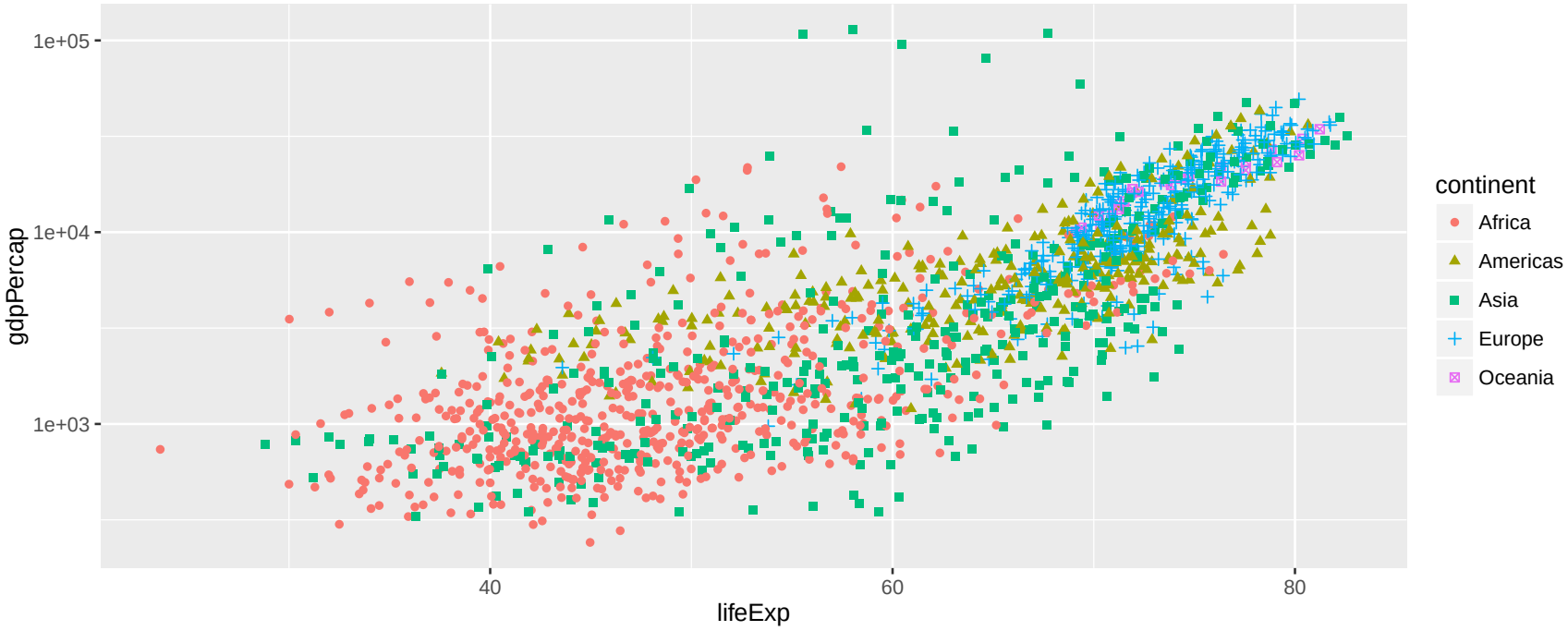
Example ggplot2 plot: histogram

```
require(ggplot2, quietly = TRUE)
ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(binwidth = 500)
```



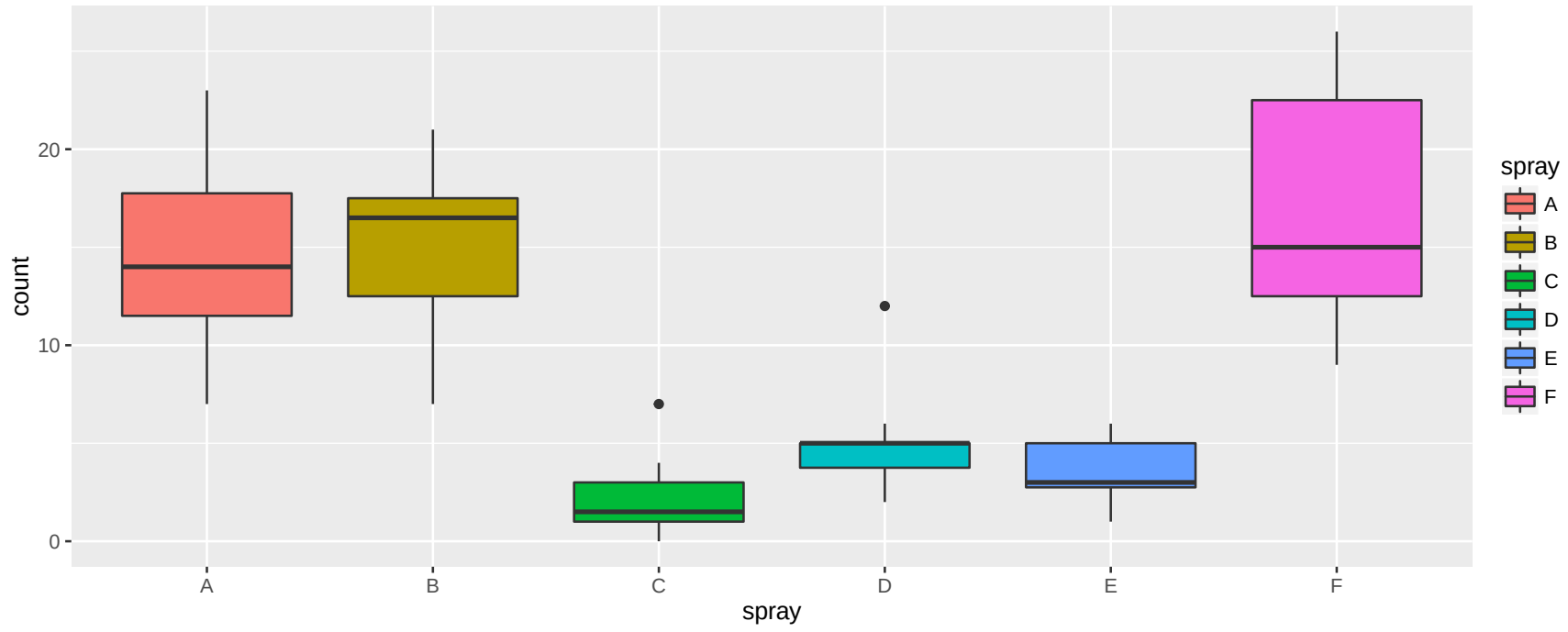
Example ggplot2 plot: scatter plot

```
require(gapminder, quietly = TRUE)
ggplot(
  data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point(
    aes(color = continent, shape = continent)) +
  scale_y_log10()
```



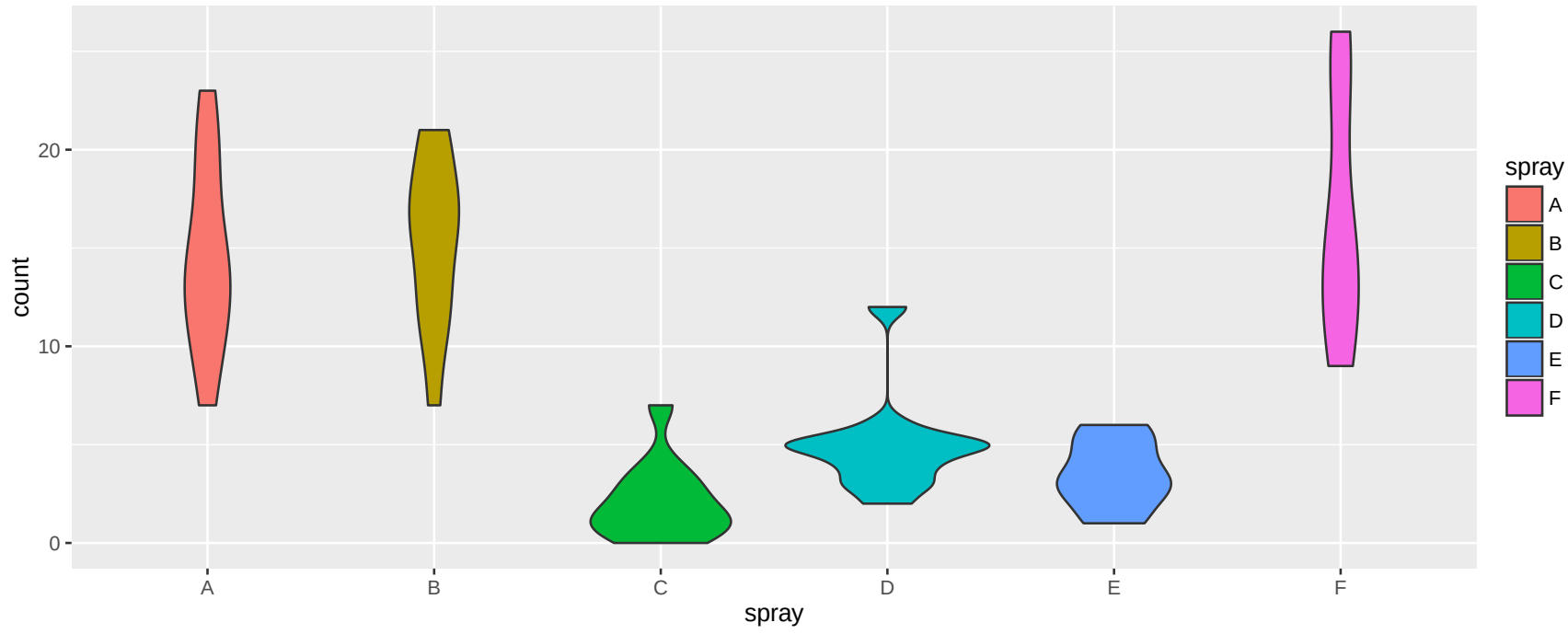
Example ggplot2 plot: box plot

```
ggplot(InsectSprays,
       aes(x = spray, y = count, fill = spray)) +
  geom_boxplot()
```



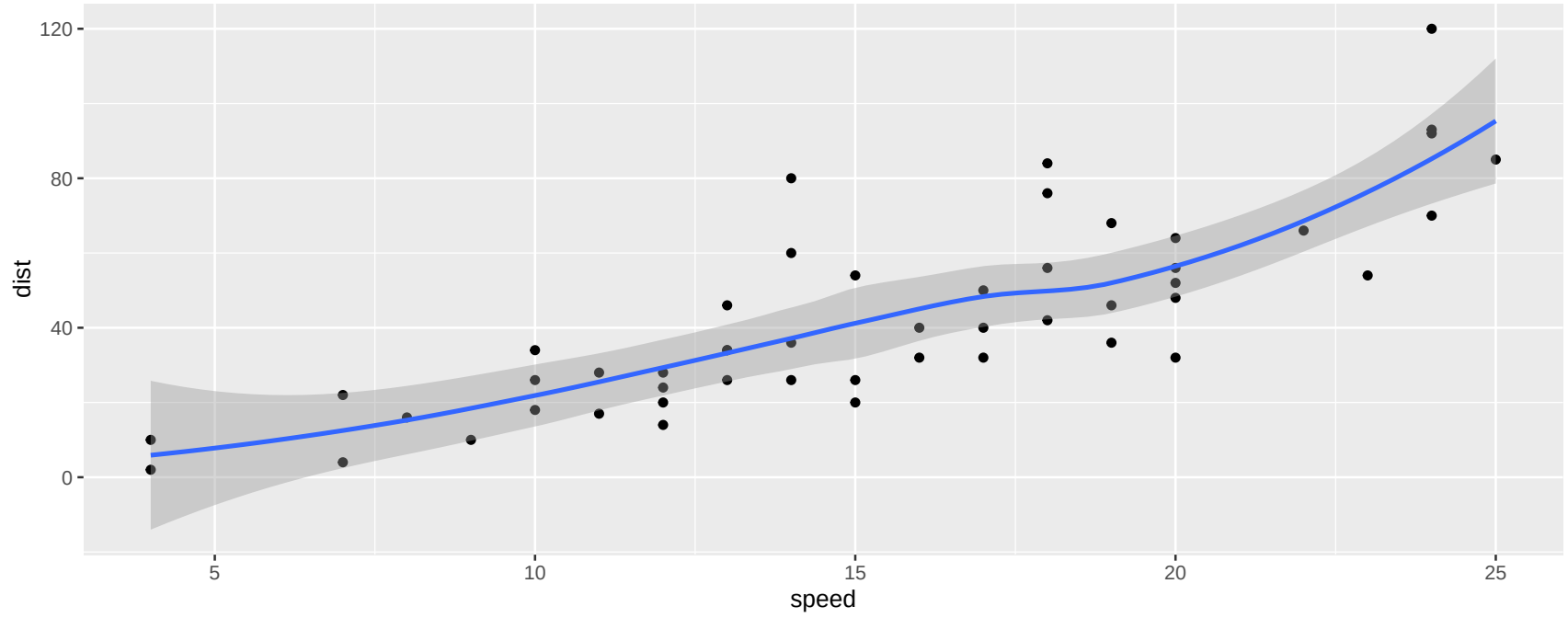
Example ggplot2 plot: violin plot

```
ggplot(InsectSprays,
       aes(x = spray, y = count, fill = spray)) +
  geom_violin()
```



Example ggplot2 plot: scatter plot with a LOESS (locally weighted scatterplot smoothing) fit and 95% confidence interval band

```
qplot(speed, dist, data = cars,
      geom = c("point", "smooth"))
```



Machine learning

Machine learning in R

- There are a huge number of packages available for machine learning in R
  - See CRAN Machine Learning task view
- Rather than interacting with algorithms directly, it's easier to drive them from a framework that provides a unified interface for performing common operations
- These provide functions for data preprocessing (cleaning), data splitting (creating partitions and resampling), training and testing functions, and utilities for generating confusion matrices and ROC plots
- Two excellent frameworks for machine learning are the `caret` and `mlr` packages
- Another useful package is H_2O, which includes algorithms implemented in Java (for speed) for deep learning and boosted decision trees, and has an interface to Spark for parallelised machine learning on large datasets — well worth checking out!

Machine learning in R

- Caret and mlr are to R what scikit-learn is to Python
- These are powerful tools that I can't cover in the available time
- Instead, I'll show a few examples of what you can do with them

Caret

Load data and partition for training and testing:

```
library(mlbench, quietly = T); data(Sonar)
library(caret, quietly = T)
set.seed(42)

inTraining <- createDataPartition(
  Sonar$Class, p = 0.75, list = FALSE)
training <- Sonar[  inTraining, ]
testing  <- Sonar[ -inTraining, ]
```

Caret

Create a grid of hyperparameters to tune over:

```
gbmGrid <- expand.grid(interaction.depth = c(1, 5, 9),
                       n.trees = (1:30) * 50,
                       shrinkage = 0.1,
                       n.minobsinnode = 20)
```

Caret

Create train control for 5-fold CV, train GBM:

```
fitControl <- trainControl(# 5-fold CV
  method = "cv",
  number = 5)
# Using R's formula language: Class ~ .
# fit Class using all features in the data
gbmFit <- train(Class ~ ., data = training,
                method = "gbm",
                trControl = fitControl,
                tuneGrid = gbmGrid,
                verbose = FALSE)
```

Caret

Predict on held-out test data and generate confusion matrix:

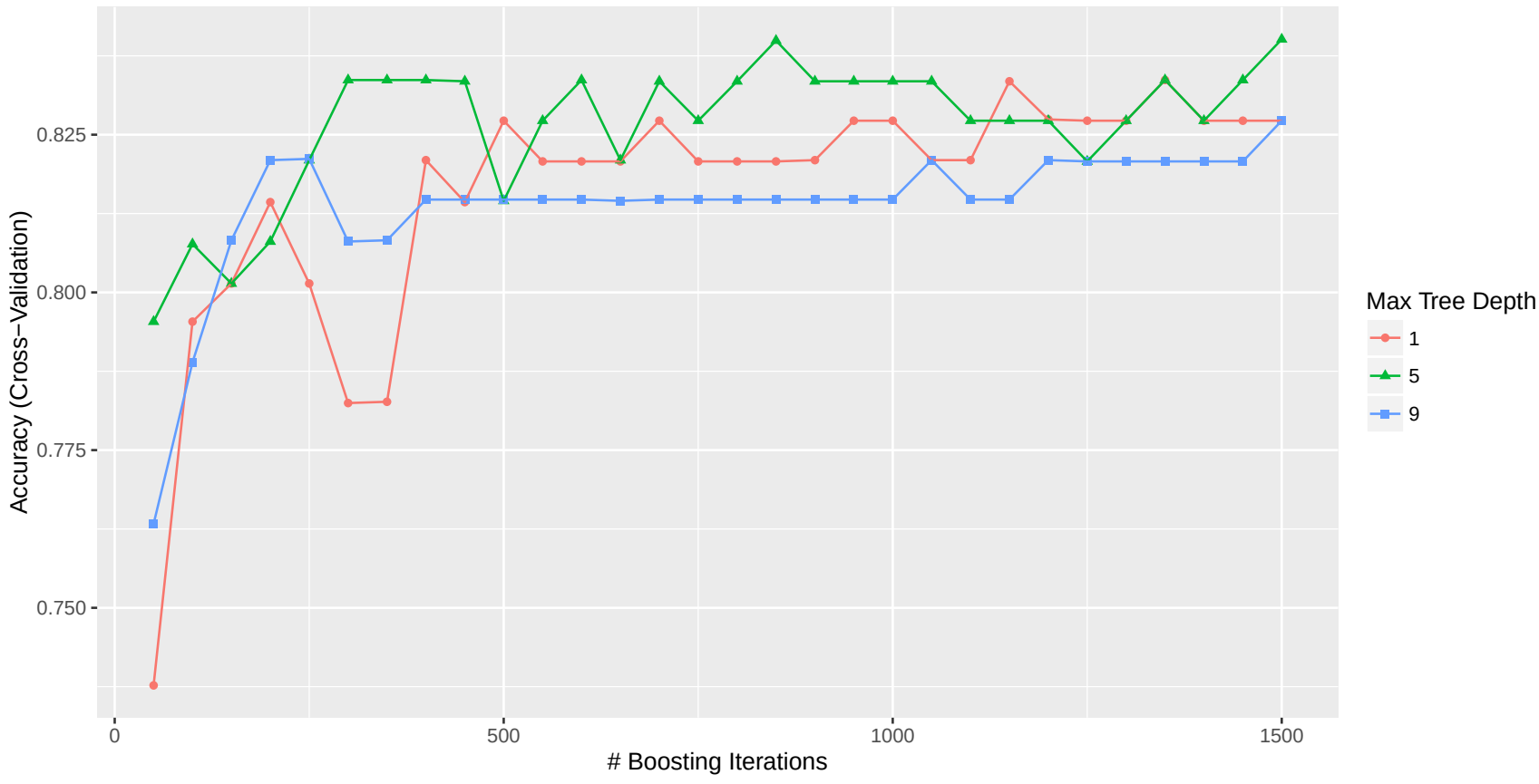
```
predictions <- predict(gbmFit, newdata = testing)
confusionMatrix(predictions, testing$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 25  6
##          R  2 18
##
##                Accuracy : 0.8431
##                  95% CI : (0.7141, 0.9298)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 2.534e-06
##
##                   Kappa : 0.6822
```

Caret

Examining the effect of hyperparameter settings:

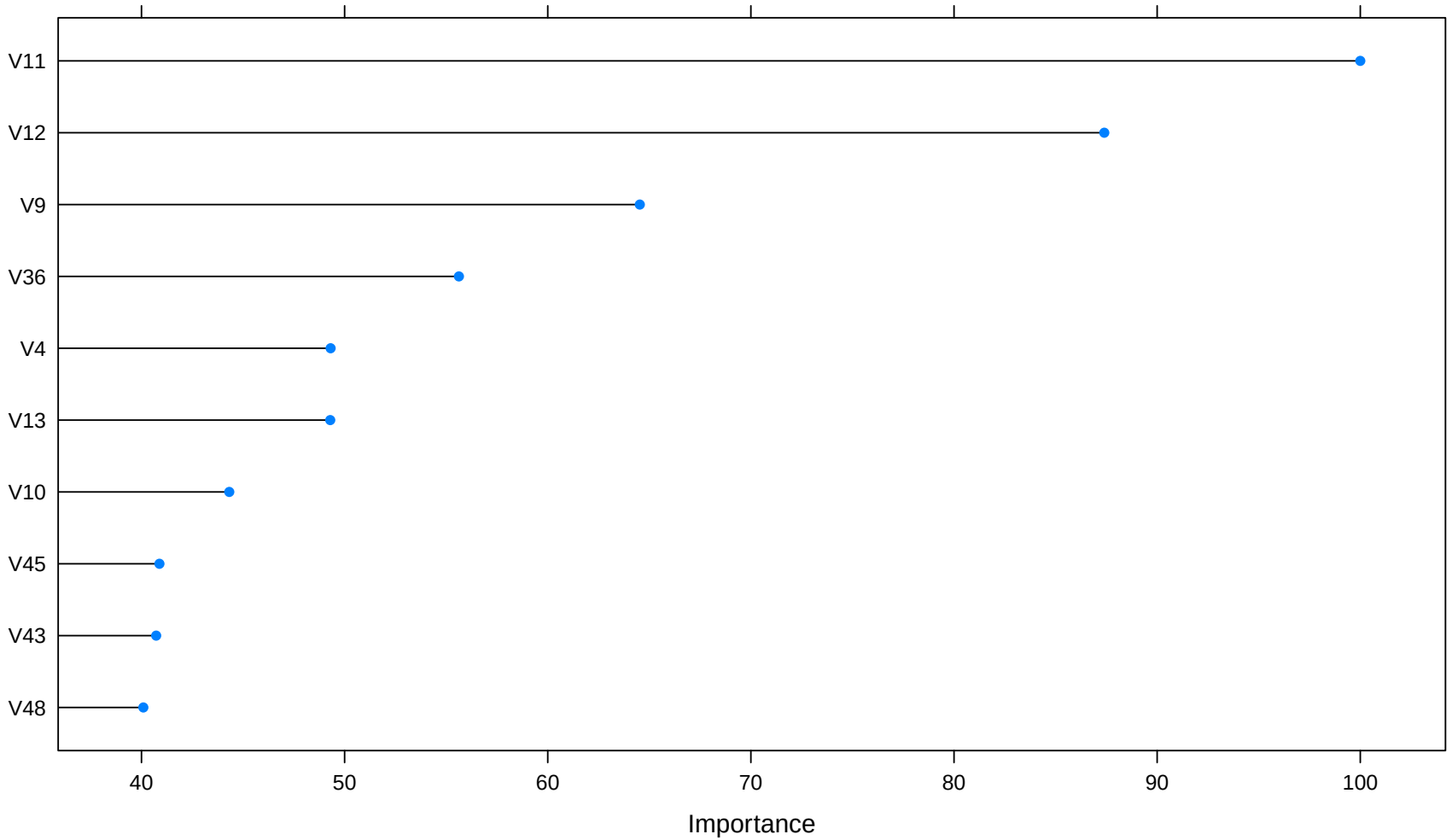
```
ggplot(gbmFit)
```



Caret

Plot variable importance for top 10 variables:

```
imp <- varImp(gbmFit); plot(imp, top = 10)
```



mlr

A simple stratified cross-validation of linear discriminant analysis with `mlr`:

```
require(mlr)
data(iris)
# Define the task
task <- makeClassifTask(id = "tutorial",
                        data = iris,
                        target = "Species")
# Define the learner
lrn <- makeLearner("classif.lda")
# Define the resampling strategy
rdesc <- makeResampleDesc(method = "CV",
                          stratify = TRUE)
# Do the resampling
r <- resample(learner = lrn, task = task,
              resampling = rdesc, show.info = FALSE)
```

mlr

Results:

```
# Get the mean misclassification error:
r$aggr
```

```
## mmce.test.mean
##           0.02
```

Deep learning with H_2O

```
require(h2o);
h2o.no_progress()
localH2O <- h2o.init() # Initialise
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:          1 hours 54 minutes
##     H2O cluster version:         3.10.3.6
##     H2O cluster version age:     1 month
##     H2O cluster name:            H2O_started_from_R_andy_
##     H2O cluster total nodes:     1
##     H2O cluster total memory:    0.84 GB
##     H2O cluster total cores:     2
##     H2O cluster allowed cores:   2
##     H2O cluster healthy:         TRUE
##     H2O Connection ip:           localhost
##     H2O Connection port:         54321
```

Deep learning with H_2O

Load prostate cancer dataset, partition into training and test sets:

```
prosPath <- system.file("extdata",
                        "prostate.csv",
                        package = "h2o")
prostate.hex <- h2o.importFile(
  path = prosPath,
  destination_frame = "prostate.hex")
prostate.split <- h2o.splitFrame(data = prostate.hex,
                                 ratios = 0.75)
prostate.train <- as.h2o(prostate.split[[1]])
prostate.test <- as.h2o(prostate.split[[2]])
```

Deep learning with H_2O

Train deep learning neural net with 5 hidden layers, ReLU with dropout, 10000 epochs, then predict on held-out test set:

```
model <- h2o.deeplearning(
  x = setdiff(colnames(prostate.train),
              c("ID","CAPSULE")),
  y = "CAPSULE",
  training_frame = prostate.train,
  activation = "RectifierWithDropout",
  hidden = c(10, 10, 10, 10, 10),
  epochs = 10000)
predictions <- h2o.predict(model, prostate.test)
```

Deep learning with H_2O

Calculate AUC:

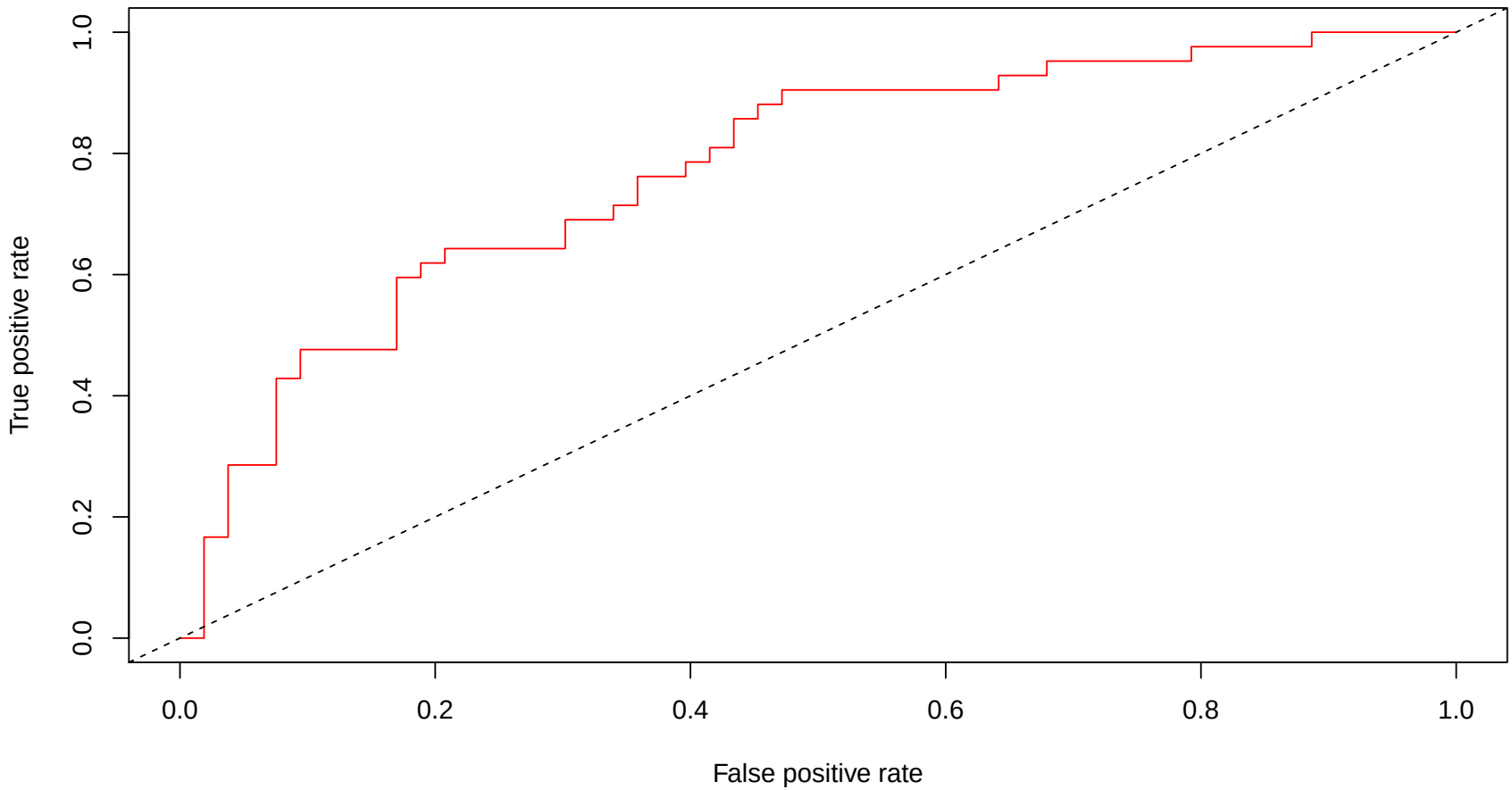
```
suppressMessages(require(ROCR, quietly = T))
preds <- as.data.frame(predictions)
labels <- as.data.frame(prostate.test[2])
p <- prediction(preds, labels)
auc.perf <- performance(p, measure = "auc")
auc.perf@y.values
```

```
## [[1]]
## [1] 0.7744834
```

Deep learning with H_2O

Plot ROC curve:

```
plot(performance(p,
                 measure = "tpr", x.measure = "fpr"),
     col = "red"); abline(a = 0, b = 1, lty = 2)
```



Reproducible research

Quick live demo (if there is time)

That's probably enough for now!

Thanks for listening!