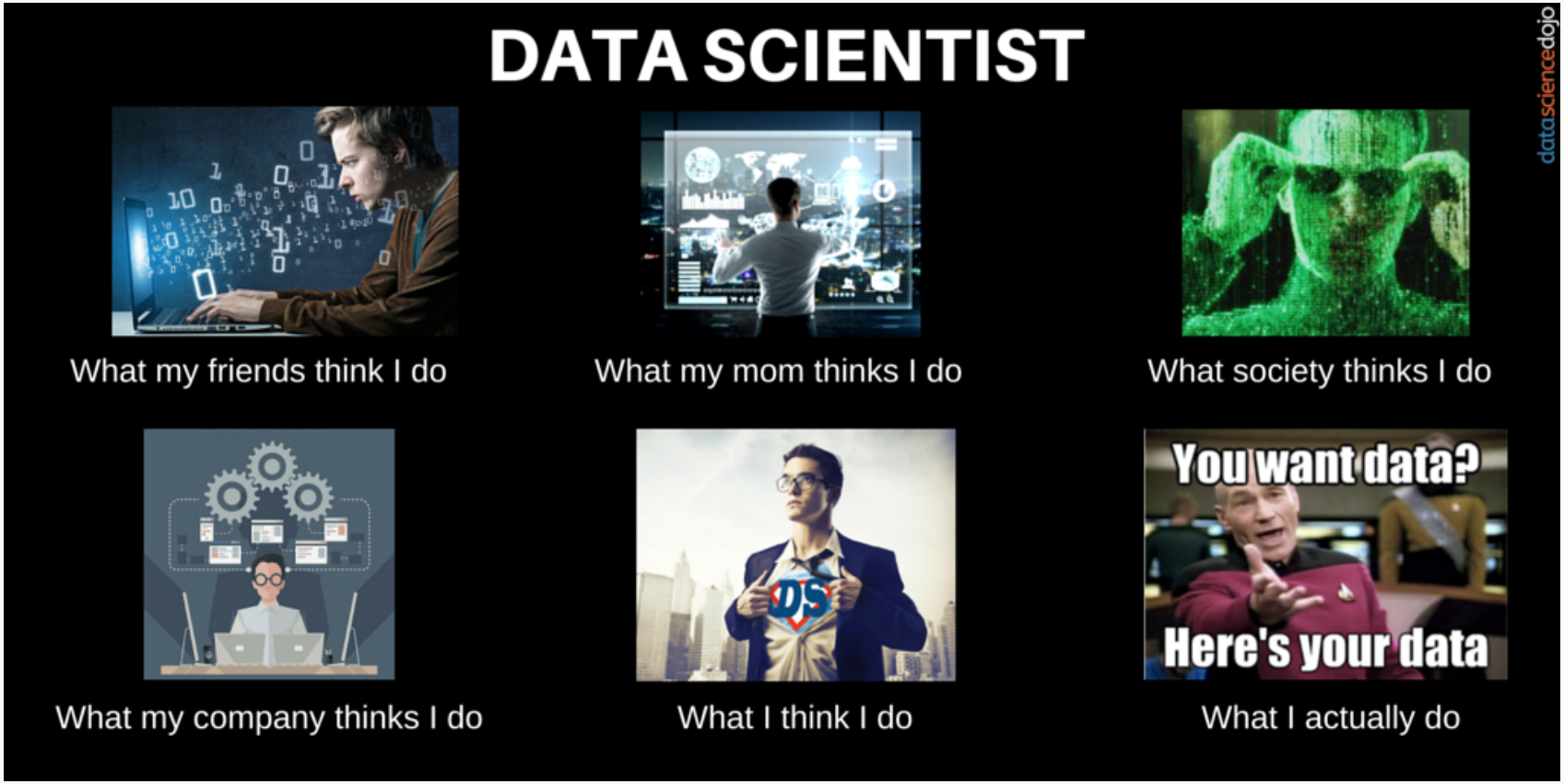
R Data Input and Output

Lecture 11

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

Data Input and Output



Source: https://twitter.com/datasciencedojo/status/627180472104128512

Data Input and Output: Overview

ILLINOIS

Getting Data Into R is part of just about any analysis!

- R excels at this
  - truly broad coverage of file formats
  - as well as 'backends' such as databases
  - or different web-based APIs
- Our focus: read/write of csv data
- Mention other formats: json, xml, ...
- Efficient R-specific storage: rds and fst
- Mention protobuf, msgpack, feather, parquet, ...

R Data Import/Export Manual

ILLINOIS

R Data Import/Export - Google Chrome

R Data Import/Expo

Dirk

https://cran.r-project.org/doc/manuals/r-release/R-data.html

R Data Import/Export

Table of Contents

Chapters in R Data Import/Export Manual

- Introduction
- Spreadsheet-like data
- Importing from other statistical systems
- Relational databases
- Binary files
- Image files
- Connections
- Network interfaces
- Reading Excel spreadsheets
- References

Comes with every R installation, but a little dry to read...

ILLINOIS

Textfiles

- Text files for interchange are common
- File separators have historically been
  - a comma, hence "comma-separated values" or CSV
  - a tab (or other whitespace) in files ending in `.txt`
  - other less common values such as | etc
- Possible issues to encounter:
  - other world regions use a comma where we use a dot
  - values must then be quoted and/or alternate separator used
  - "encodings" can also be an issues

Basic use

- The underlying base R function is `read.table()`
  - it has numerous options so see the help page
  - defaults to *any* so-called white space as separator (*i.e.* one or more tabs, spaces, newlines)
  - also defaults to *no header*
  - can handled missing values and different types
- More frequenly used: `read.csv()`
  - wrapper around `read.table()`
  - different defaults for separator, header, fill
  - we will discuss both, one can focus on this one

`read.table`: work horse behind other functions

```
R> args(read.table)
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
    col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
    encoding = "unknown", text, skipNul = FALSE)
NULL
R>
```

See `help(read.table)` for more complete coverage

ILLINOIS

read.table: Examples

```
# no header, default separator
read.table("file.txt")

# header from 1st line, 'pipe' separator
read.table("file.txt", header=TRUE, sep="|")

# do not convert text variables to factors
read.table("file.txt", stringsAsFactors=FALSE)

# skip 1st column, 2nd column taken as numeric
read.table("file.txt", colClasses=c("NULL", "numeric"))
```

`read.csv`: comma-separated file reader

```
R> args(read.csv)
function (file, header = TRUE, sep = ",", quote = "\"",
    dec = ".", fill = TRUE, comment.char = "", ...)
NULL
R>
```

`header`, `row.names`, `col.names`, `stringsAsFactors`, `sep`, ... are all very important, and some are set differently now.

`...` means other arguments permitted and passed through.

Example: Titanic

- There are a number of titanic datasets in R packages
- Base R package `datasets` has it too
- Here I wanted a file, so this page at Stanford had one

```
R> data <- read.csv("titanic.csv")  # default Arguments
R> str(data)
# 'data.frame': 887 obs. of  8 variables:
#  $ Survived               : int  0 1 1 1 0 0 0 0 1 1 ...
#  $ Pclass                 : int  3 1 3 1 3 3 1 3 3 2 ...
#  $ Name                   : Factor w/ 887 levels "Capt. Edward Gifford Crosby",..: 602 823 172 814 733 464 700 3
#  $ Sex                    : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2 2 1 1 ...
#  $ Age                    : num  22 38 26 35 35 27 54 2 27 14 ...
#  $ Siblings.Spouses.Aboard: int  1 1 0 1 0 0 0 3 0 1 ...
#  $ Parents.Children.Aboard: int  0 0 0 0 0 0 0 1 2 0 ...
#  $ Fare                   : num  7.25 71.28 7.92 53.1 8.05 ...
```

- One commonly use option: `stringAsFactors=FALSE`

```
R> data <- read.csv("titanic.csv", stringsAsFactors=FALSE)
R> str(data)
# 'data.frame': 887 obs. of  8 variables:
#  $ Survived               : int  0 1 1 1 0 0 0 0 1 1 ...
#  $ Pclass                 : int  3 1 3 1 3 3 1 3 3 2 ...
#  $ Name                   : chr  "Mr. Owen Harris Braund" "Mrs. John Bradley (Florence Briggs Thayer) Cumings" "
#  $ Sex                    : chr  "male" "female" "female" "female" ...
#  $ Age                    : num  22 38 26 35 35 27 54 2 27 14 ...
#  $ Siblings.Spouses.Aboard: int  1 1 0 1 0 0 0 3 0 1 ...
#  $ Parents.Children.Aboard: int  0 0 0 0 0 0 0 1 2 0 ...
#  $ Fare                   : num  7.25 71.28 7.92 53.1 8.05 ...
```

Notice the difference?

Save an individual R object *portably*

Very useful for *efficient* (compressed, binary) and *portable* (from machine to machine) data storage

```
# save obj portably and compressed
saveRDS(obj, file="somefile.rds")

# in another session or on another machine
newobj <- readRDS("somefile.rds")
```

```
R> saveRDS(data, file="titanic.rds")  # save to an rds
R> newdata <- readRDS("titanic.rds")  # reload
R>
R> identical(newdata, data)  # check that bitwise identical
[1] TRUE
R>
R> # compressed and portable storage
R> object.size(gs)
118176 bytes
R> file.info("titanic.rds")$size
[1] 18800
```

`fread` (in package `data.table`)

- extremely fast as it reads chunks in parallel
- good heuristics for inferring column types
- generally "just works" and is fast
- also reads from commands
- recommended (and `data.table` is next lecture)
- (and there is also `fwrite`)
- (and we will see both later)

`read_csv` (in package `readr`)

- part of tidyverse, opinionated
- not as fast as `fread`
- not as general (may need columns specified)
- returns as a `tibble` object which or may not be desirable

Add-on Package

- Fastest binary writer / reader
- *Highly* optimized, parallel
- Very fast reads and writers
- Good for large-enough data
- But maybe not worth for small data
- On CRAN as package `fst`
- Website fstpackage.org

fst 0.9.0 | Reference | Changelog

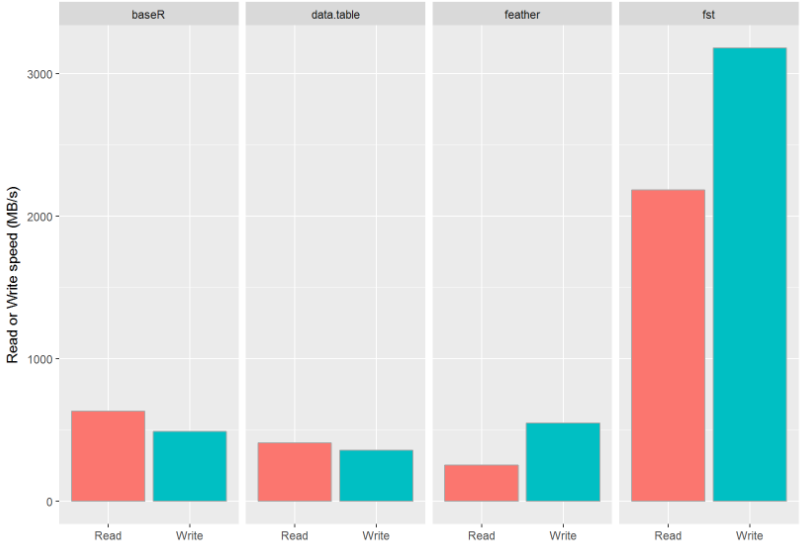
build passing | build passing | License AGPL v3 | CRAN 0.9.0 | codecov 58% | downloads 9595/month

Overview

The *fst package* for R provides a fast, easy and flexible way to serialize data frames. With access speeds of multiple GB/s, *fst* is specifically designed to unlock the potential of high speed solid state disks that can be found in most modern computers. Data frames stored in the *fst* format have full random access, both in column and rows.

The figure below compares the read and write performance of the *fst* package to various alternatives.

Method	Format	Time (ms)	Size (MB)	Speed (MB/s)	N
readRDS	bin	1577	1000	633	112
saveRDS	bin	2042	1000	489	112
fread	csv	2925	1038	410	232
fwrite	csv	2790	1038	358	241
read_feather	bin	3950	813	253	112
write_feather	bin	1820	813	549	112
read_fst	**bin**	**457**	**303**	**2184**	**282**
write_fst	**bin**	**314**	**303**	**3180**	**291**



Very useful extensions

- In `read.*()` functions, `file=...` generalizes
- We can read from any "Connections" object
  - a URL ie anything "on the web" (or network-reachable)
  - a pipe ie output from another program
  - a compressed file
  - a network socket
- See `help(connections)` and the Data I/O Manual for more.

Example

```
R> # we can read directly via http or https
R> webdata <- read.csv("https://some.site.com/path/data.csv")
R>
R> # we can also read output from other programs directly
R> res <- read.table(pipe("python someOtherScript.py"))
R>
R> # or any other shell program via pipe() ...
```

Databases

DBI Abstraction

- DBI: One of the oldest packages for R (and predecessor S(-Plus)).
- Powerful abstraction of db *backend* via a unified interface.
- Connect to a given db using the corresponding driver.
- Once connected, run query, insert, updates, ... as usual.
- To run code on a different engine, just switch one statement.

DBI Backends

- RMySQL
- ROracle
- RPostgreSQL
- RSQLite
- RSQLServer
- MonetDB.R

and more

DBI Example: SQLite

```
R> library("RSQLite")
Loading required package: DBI
R> con <- dbConnect(RSQLite::SQLite(), ":memory:")
R> dbWriteTable(con, "titanic", data)
[1] TRUE
R> dim(dbGetQuery(con, "select * from sales limit 10"))
[1] 10 8
R>
```

DBI Example: PostgreSQL

```
R> library("RPostgreSQL")
R> drv <- dbDriver("PostgreSQL")
R> con <- dbConnect(drv, "username", "password", "dbname")
R> data <- dbGetQuery(con, "SELECT * from sales")
R> dbDriver(con)
```

Others

- Lots of other connection packages
- RODBC, RJDBC, ...
- NoSQL: redis, mongo, monet, ...
- Google bigquery, Apache Cassandra
- Hadoop, Spark, ...
- If something has data, there will be an R package somewhere.

Spreadsheets

Reading xls/xlxs

Several CRAN packages offer functionality:

- `readxl` is a more recent and fast reader, not writing
- `openxlsx` is a C++-based bi-directional package
- `XLConnect` allows bi-directional read + write
- (older, now archived) `xlxs` requires Java, reliable, bi-directional, allows creation of full-featured spreadsheets

Writing xls/xlxs

Several CRAN packages offer functionality:

- `writexl` can export to xlsx, fast, no dependency
- `openxlsx` is a C++-based bi-directional package
- `XLConnect`, also bi-directional
- `xlxs` (archived) requires Java, reliable, bi-directional, allows creation of full-featured spreadsheets

Other

XML, JSON, HDF5, NetCDF, ...

Choices:

- Several XML readers: `XML`, `xml2`, ...
- Several JSON readers: `RJSONIO`, `jsonlite`, ...
- Other readers for YAML, TOML, ...
- Scientific data formats such as HDF5 and NetCDF ...
- 'Big data' formats like Parquet via `arrow` or `miniparquet`
- Plus more domain-specific readers

In general, if a format exists, there may be a reader for it.

Anything 'web'-related

ILLINOIS

The excellent CRAN Task View on Web Technologies has sections on

- Parsing Data from the Web
- Curl, HTTP, FTP, HTML, XML, SOAP
- Authentication
- Web Frameworks
- JavaScript
- Code Sharing

plus a truly exhaustive list of *Data Sources on the Web Accessible via R* organised by discipline

Summary

R can read or write

- just about anything from text files
- just about any existing database backend
- specialised files (`xls`, `xlsx`, statistics packages)
- via connections from programs, URLs, and more
- higher-performance readers and writers