Installing stable BART R package from source

- Our **BART** R package (current version 2.9) is on the Comprehensive R Archive Network (CRAN)
- `https://cran.r-project.org/package=BART`
- Install into your personal vs. global R library
- `~/.Rprofile`
- `# my .Rprofile contains this personal library`
- `.libPaths("~/.R/library")`
- Installing **BART** (which depends on the Rcpp package)
- From source with the Unix command line (from here on Unix means UNIX/Linux/macOS)
- Requires a full C++ toolchain like GNU GCC or Apple Xcode
- C++11 required: GCC 9/Xcode 9 or higher recommended
- `$ R CMD INSTALL BART_2.9.tar.gz`
- From the R prompt for Windows and Unix

```
> options(repos=c(CRAN="https://cran.r-project.org"))
> install.packages("BART", dependencies=TRUE)
```

Installing development BART3 R package from source

- Our **BART3** R package (current version 4.4)
- `https://github.com/rsparapa/bnptools`
- Installing **BART3** (which depends on the Rcpp package)
- From source with the Unix command line
- `$ R CMD INSTALL BART3_4.4.tar.gz`
- From the R prompt for Windows and Unix

```
> options(repos=c(CRAN="https://cran.r-project.org"))
> install.packages("Rcpp", dependencies=TRUE)
> library(remotes)
> install_github("rsparapa/bnptools", subdir="BART3")
```

Rcpp for seamless R and C++ integration

- Eddelbuettel and Francois 2011 *JSS*
- `https://cran.r-project.org/package=Rcpp`
- Circa 02/2022: dependency for 2500 (13%) CRAN packages
- Seamless passing of objects from R to C++ and vice versa including R objects such as vectors, matrices and lists
- Facilitates passing by pointer for optimal performance
- Convenient constructors/destructors for R objects for automated creation and garbage collection
- Integration with the R pseudo-random number generator including the standard distributions
- C++ namespace for the standalone Rmath library C functions
- A portable C++ header-only library
  i.e., no compiled binaries are needed
- Side effect: installation/compilation is typically faster but there may be outliers/edge-cases
- See Package Dependencies at `http://cran.r-project.org/doc/manuals/r-release/R-exts.html`

BART software with a `predict` function

		R packages		
Debut	Language	Stable (CRAN)	Development	Multi-threading
2006	C++	**BayesTree**	None	None
2013	Java	**bartMachine**		Java
2014	C++	**dbarts**		forking
2014	C++	MPI BART source code		MPI
2017	C++	**BART** 2.9*	**BART3***	OpenMP/forking
2019	C++	**rbart** 1.0*	**hbart***	OpenMP
2019	C++	None	**mxBART***	OpenMP/forking
2021	C++	None	**mBART***	OpenMP/forking
2021	C++	**nftbart** 1.2*	**nftbart***	OpenMP
		*Descendents of MPI BART		

Development on `github.com` by users `rsparapa` (me), `cspanbauer` (Charley Spanbauer) and `remcc` (Rob McCulloch)

Special thanks to Rob (**BART**), Matt Pratola for **rbart**
Hugh Chipman, Robert Gramacy, the R Core team,
the Rcpp Core team and so many others in the FOSS community

BART software features: descendents of MPI BART

Stable Development	**BART** **BART3**	**nftbart** **nftbart**	**rbart** **hbart**	 **mBART**
`github.com` user		`rsparapa`		`remcc`
`predict` function	Yes	Yes	Yes	**BART**
heteroskedastic	No	Yes	Yes	No
monotonic	No	No	No	Yes
continuous	Yes	Yes	Yes	Yes
binary/categorical	Yes	No	No	No
right censoring	Yes	Yes	No	No
left censoring	No	Yes	No	No
competing risks	Yes	No	No	No
recurrent events	Yes	No	No	No
sparse prior	Yes	No	No	No
marginal effects	**BART3**	Yes	No	No
missing imputation	Yes	Yes	No	No
advanced tree proposals	No	Yes	Yes	No
nonparametric error	No	Yes	No	No
C++ header-only	**BART3**	No	**hbart**	No

A brief history of multi-processing/-threading

- 1961: Burroughs B5000 asymmetric multi-processing
- 1962: Burroughs D825 symmetric multi-processing (SMP)
- 1967: Amdahl's law: $((1-b)/C+b)^{-1}$
- 1970: Fork system call appears in UNIX
- Early 1990's: Message Passing Interface for *shared nothing* distributed processing across multiple computers
- Late 1990's: paradigm shift towards multi-threading where multiple CPU cores can process a thread simultaneously
- 1997-8: OpenMP for *shared memory/storage* multi-processing within a single computer
- 2000: AMD64 architecure debuts: native execution of 32-bit x86 legacy code as well as new 64-bit x86 instructions
- 2003: Linux kernel 2.6 unleashes SMP support
- 2010-11: affordable, mass-produced, multi-core CPUs released as Intel Xeon and AMD Opteron supporting 16 threads
- 2014: **parallel** package included in R supporting shared memory multi-threading via forking on UNIX/Linux/macOS

Modern multi-threading software frameworks

- Message Passing Interface (MPI) for multiple nodes
- Pratola, Chipman, Gattiker, Higdon, McCulloch, Rust. Parallel Bayesian Additive Regression Trees. JCGS 2014;23:830-852 `https://arxiv.org/abs/1309.1906`
- OpenMP for single nodes: used by **BART** for `predict`
- detected by the GNU autotools when **BART** installed
- defines a C pre-processor macro (or not): `_OPENMP`
- macOS: see `https://mac.r-project.org/openmp`
- Windows: OpenMP/GNU autotools generally unavailable
- Forking for single nodes: **parallel** R package
- Forking not supported on Windows
- see the help page: `?mcparallel`
- Forking used by **BART** for posterior sampling
- `mc.gbart`, `mc.surv.bart`, `mc.recur.bart`, etc.
- can be used by the `predict` function instead of OpenMP (when OpenMP is unavailable)

Multi-threading: can I run multiple threads?

- Windows: not currently supported by R or CRAN
- Unix only at this point including Linux and macOS
- OpenMP

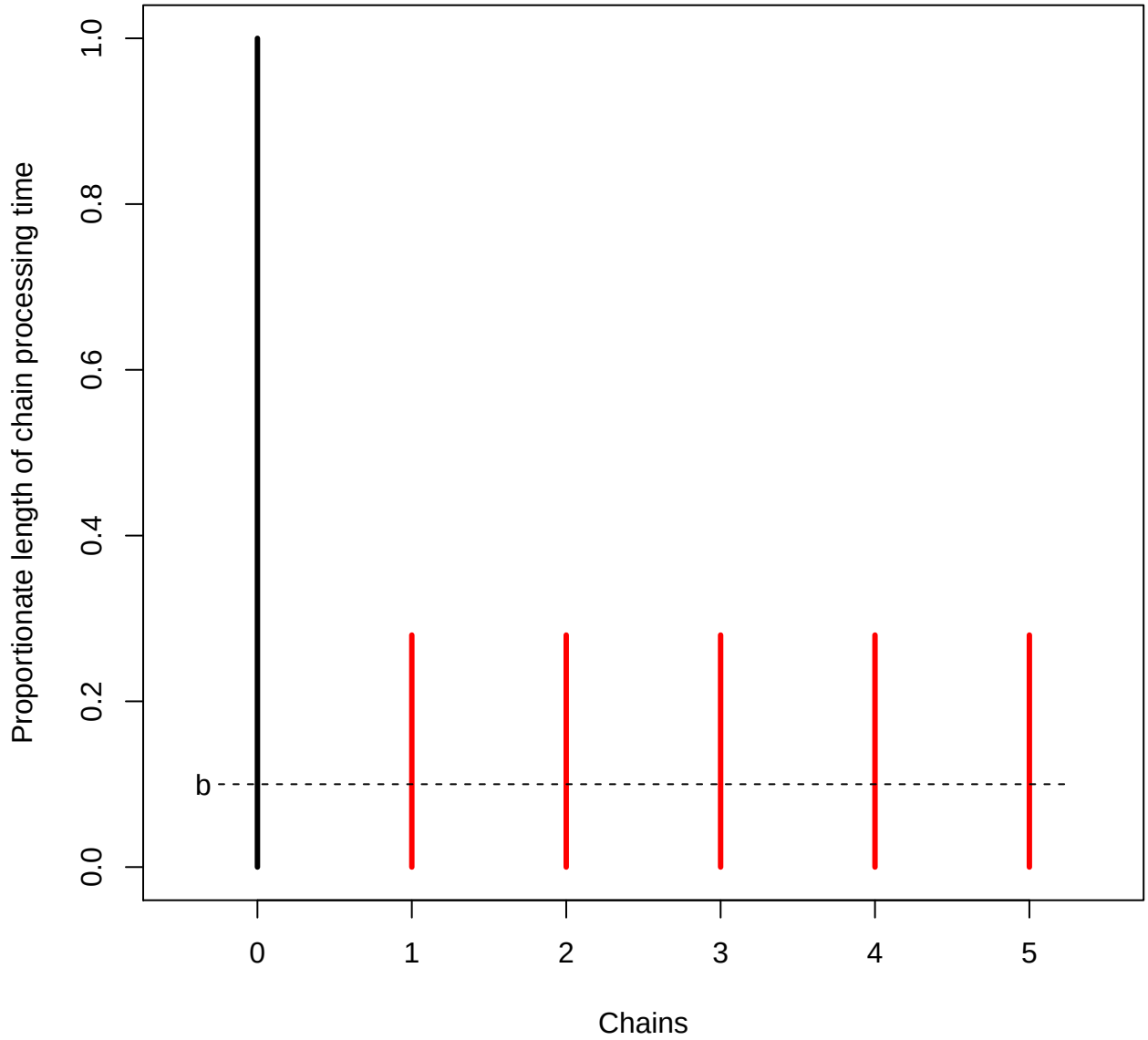
```
> library(BART)
> mc.cores.openmp()
```

- Returns 0 if OpenMP not available; 1 if it is
- Forking

```
> library(parallel)
> detectCores()
```

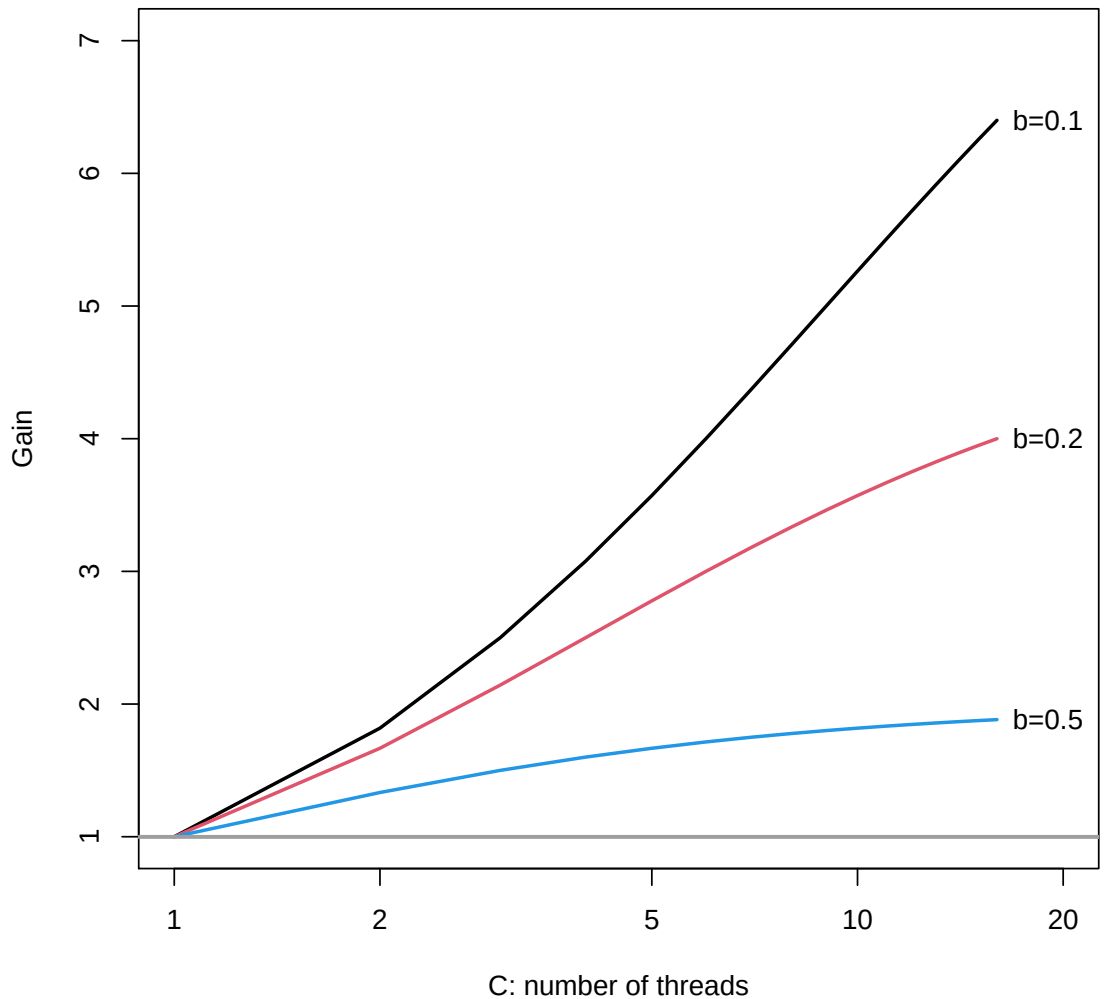
- Returns 1 or more (and occasionally NA)

MCMC is “embarrassingly” parallel



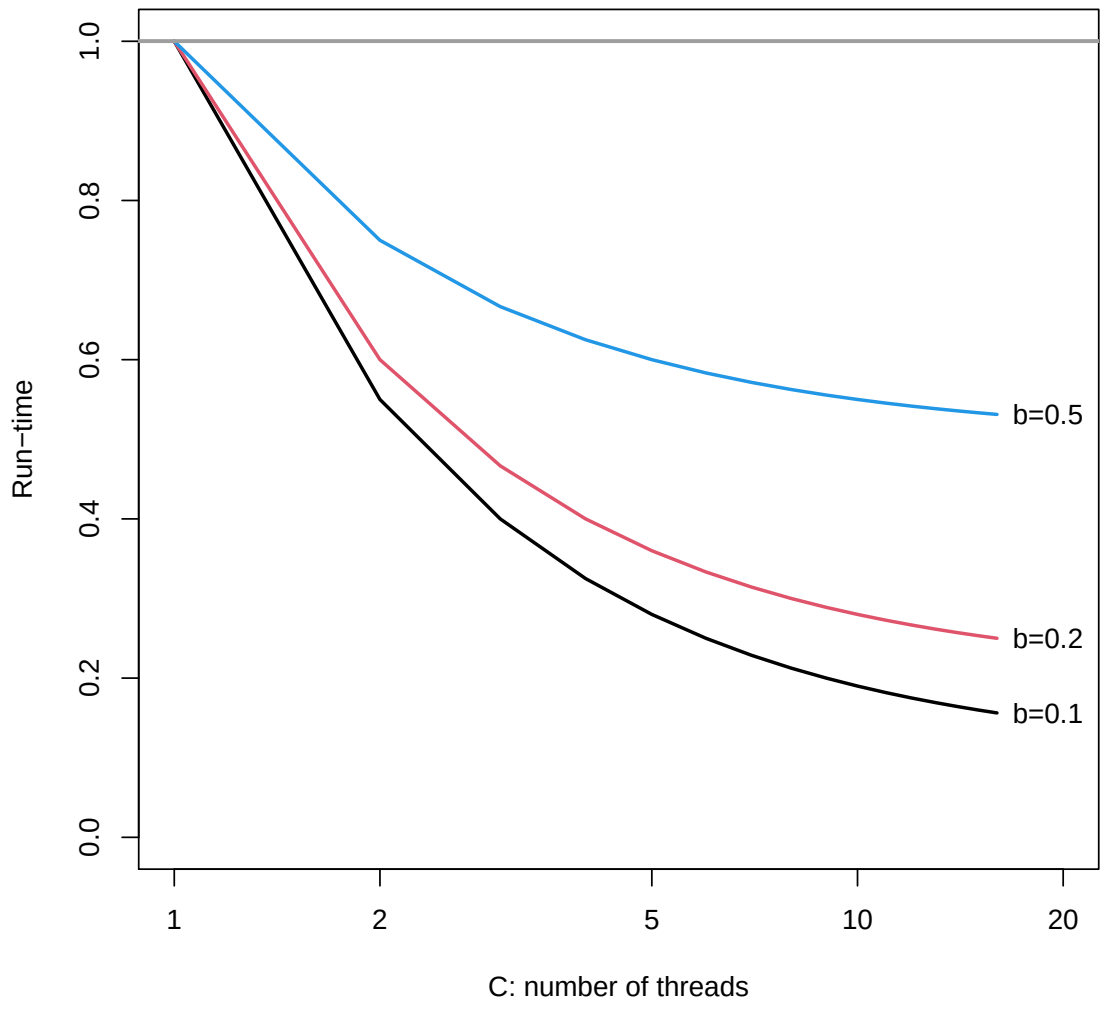
Amdahl's Law and the MCMC Corollary

- Gain $= \frac{(1-b)+b}{(1-b)/C+b} = \frac{1}{(1-b)/C+b}$ and b is the burn-in fraction



Amdahl's Law and Run-time

- Run-time $=$ 1/Gain $= \{(1-b)/C + b\}$



Multi-threading: random access memory (RAM) I

- IEEE 754 specifies that every double-precision number consumes 8 bytes (64 bits) so you can estimate your needs
- If A is $m \times n$, then $\mathbf{RAM}(A) = 8 \times m \times n$ bytes
- If you consume all of the physical RAM, the system will *swap* segments out to virtual RAM which are disk files:
  this will degrade performance and possibly crash the system
- On Unix, you can monitor memory and swap usage with `top`
- Within R, you can determine the size of an object with the `object.size` function

Multi-threading: random access memory (RAM) II

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- R matrices are column-major: $[a_{11}, a_{21}, \ldots, a_{12}, a_{22}, \ldots]$
- C++ matrices are row-major: $[a_{11}, a_{12}, \ldots, a_{21}, a_{22}, \ldots]$
- This is easily addressed with a transpose
  instead of passing A from R to C++, we pass A^t
- R passes objects by pointer, but it is copy-on-write
- All objects in the parent thread can be read by the child thread from the pointer without a copy, but when an object is altered/written by the child, then a new copy is created
- $\mathbf{RAM}(A) = 8 \times m \times n \times C$ and C is the no. of children
- If the parent transposes, we avoid the copy: `A <- t(A)`

Multi-threading: interactive vs. batch processing

- Interactive jobs must take precedence over batch jobs to prevent the user experience from suffering high latency
- Examples of interactive activity: typing at the command line, editing files with Emacs, reading email, browsing the web
- In the **tools** R package, there is the `psnice` function
- Paraphrased from the `?psnice` help page

  Unix has a concept of process priority. Priority is assigned values from 0 to 39 with 20 being the normal priority and (counter-intuitively) larger numeric values denoting lower priority. Adding to the complexity, there is a "nice" value, the amount by which the priority exceeds 20. Processes with higher nice values will receive less CPU time than those with normal priority. Generally, processes with nice 19 are only run when the system would otherwise be idle.

- by default, the **BART** package children have nice set to 19

Hot/Cold-decking missing imputation

- Mainframe era legend: the US Census was on punch cards
- Often items were either mistakenly or intentionally left blank
- If the collected data is (NOT) Missing Completely at Random, you can (NOT) drop missing records from statistical analysis
- Hot-decking is the substitution of a nearby neighbor's value to replace a missing value on the resident's form
- For regression, compute the distance between two records via $\boldsymbol{y}$'s
- But, what does "nearby" mean?
  What about dichotomous or censored time-to-event outcomes?
- For one or more missing covariates, record-level cold-decking imputation can be employed that is biased towards the null
- Non-missing values from another record are randomly selected regardless of the $\boldsymbol{y}$'s (effectively, simple random sampling)
- This missing data imputation method is sufficient for data sets with relatively few missing values
- More pervasive missing data require advanced techniques such as Sequential BART (Xu, Daniels et al. 2016 *Biostatistics*) `https://cran.r-project.org/src/contrib/Archive/sbart`

Cold-decking missing imputation and multiple imputation

- Suppose we have the following 5 variables for childhood growth: age, gender, race/ethnicity, waist circumference, weight
- It is reasonable to assume that these variables have a relationship between them
- Suppose record i has the observed/missingness pattern
  A_i B_i NA NA NA
- And we randomly draw record j to replace its values
  C_j D_j NA E_j F_j
- Now, record i looks like this
  A_i B_i NA E_j F_j
- So, we randomly draw again: record k
  G_k NA H_k I_k NA
- So, record i looks like this
  A_i B_i H_k E_j F_j
- Each MCMC chain can have its own imputation
  providing multiple imputation rather than single imputation

Continuous outcomes with `gbart` and `mc.gbart`

```
set.seed(99); post <- gbart(x.train, y.train, ...,
        ndpost=M, keepevery=1)
## or multi-threaded: BART3 options(mc.cores=2)
post <- mc.gbart(x.train, y.train, ...,
        ndpost=M, keepevery=1, mc.cores=2, seed=99)
```

Matrices/data.frames: `x.train` and, optionally, `x.test`: x_i

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

Output object, `post`, of type `wbart` which is essentially a list

Matrices: `post$yhat.train` and `post$yhat.test`: $\hat{y}_{im} = f_m(x_i)$

$$\begin{bmatrix} \hat{y}_{11} & \cdots & \hat{y}_{N1} \\ \vdots & \cdots & \vdots \\ \hat{y}_{1M} & \cdots & \hat{y}_{NM} \end{bmatrix}$$

`predict` input and output

```
## BART3: options(mc.cores=2)
pred <- predict(post, x.test, mc.cores=1, ...)
```

`post` object of type `wbart` (continuous), `pbart` (binary probit), `lbart` (binary logistic), `survbart` (survival analysis), `criskbart` (competing risks) or `recurbart` (recurrent events)

Input matrices: `x.test`: x_i

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_Q \end{bmatrix}$$

Output matrix for `wbart`: $\hat{y}_{im} = f_m(x_i)$

$$\begin{bmatrix} \hat{y}_{11} & \cdots & \hat{y}_{Q1} \\ \vdots & \cdots & \vdots \\ \hat{y}_{1M} & \cdots & \hat{y}_{QM} \end{bmatrix}$$

Pseudo-code: parallel R package and `mc.gbart`

```
mc.gbart <- function(..., nice=19, transposed=FALSE) {
RNGkind("L'Ecuyer-CMRG")
set.seed(seed)
parallel::mc.reset.stream()
if(!transposed) {
    x.train <- t(x.train)
    x.test <- t(x.test)
}
mc.cores <- min(c(mc.cores, parallel::detectCores()))
...
for(i in 1:mc.cores)
        parallel::mcparallel({psnice(value=nice);
                    gbart(..., transposed=TRUE)},
                    silent=(i!=1))
                    ## to avoid duplication of output
                    ## capture from first child only
post.list <- parallel::mccollect()
```

Multi-threading demos

- With the `system.file` function, you can find where R installed the **BART** package as well as files and sub-directories
- `system.file(package='BART')`
- `system.file('demo', package='BART')`
- `system.file('demo/friedman.wbart.R', package='BART')`
- For the demos, you can use the `demo` function
- `demo(package='BART')`
- `demo('friedman.wbart', package='BART')`
- But then you have to press Return after each plot

Creating a BART executable with C++ sans R

- **Rcpp** allows C++ code to rely on the R PRNG
- In our package, you can build C++ BART without R/**Rcpp**
- C++ BART is built with the standalone Rmath library which is part of the R project and contained in the R source code
- Rmath provides an R compliant PRNG and all of the useful R functions like `pnorm`
- You can optionally build with the PRNG provided by the C++ `random` class from the Standard Template Library (STL)
- `system.file('cxx-ex', package='BART')`
- `system.file('cxx-ex/Makefile', package='BART')`