Exploring data #3

forcats

forcats

Hadley Wickham has developed a package called `forcats` that helps you work with categorical variables (factors). I'll show some examples of its functions using the `worldcup` data set:

```
library(forcats)
library(faraway)
data(worldcup)
```

forcats

The `fct_recode` function can be used to change the labels of a function (along the lines of using `factor` with `levels` and `labels` to reset factor labels).

One big advantage is that `fct_recode` lets you change labels for some, but not all, levels. For example, here are the team names:

```
worldcup %>%
  filter(str_detect(Team, "^US")) %>%
  slice(1:3) %>% select(Team, Position, Time)
```

```
## # A tibble: 3 x 3
##     Team   Position  Time
##   <fctr>     <fctr> <int>
## 1    USA Midfielder    10
## 2    USA   Defender   390
## 3    USA   Defender   200
```

If you just want to change "USA" to "United States", you can run:

```
worldcup <- worldcup %>%
  mutate(Team = fct_recode(Team, `United States` = "USA"))
worldcup %>%
  filter(str_detect(Team, "^Un")) %>%
  slice(1:3) %>% select(Team, Position, Time)
```

```
## # A tibble: 3 x 3
##            Team   Position  Time
##          <fctr>     <fctr> <int>
## 1 United States Midfielder    10
## 2 United States   Defender   390
## 3 United States   Defender   200
```

forcats

You can use the `fct_lump` function to lump uncommon factors into an "Other" category. For example, to lump the two least common positions together, you can run (`n` specifies how many categories to keep outside of "Other"):

```
worldcup %>%
  mutate(Position = fct_lump(Position, n = 2)) %>%
  count(Position)
```

```
## # A tibble: 3 x 2
##     Position     n
##       <fctr> <int>
## 1   Defender   188
## 2 Midfielder   228
## 3      Other   179
```

forcats

You can use the `fct_infreq` function to reorder the levels of a factor from most common to least common:

```
levels(worldcup$Position)
```

```
## [1] "Defender"   "Forward"    "Goalkeeper" "Midfielder"
```

```
worldcup <- worldcup %>%
  mutate(Position = fct_infreq(Position))
levels(worldcup$Position)
```

```
## [1] "Midfielder" "Defender"   "Forward"    "Goalkeeper"
```

forcats

If you want to reorder one factor by another variable (ascending order), you can use `fct_reorder` (e.g., homework 3). For example, to re-level `Position` by the average shots on goals for each position, you can run:

```
levels(worldcup$Position)
```

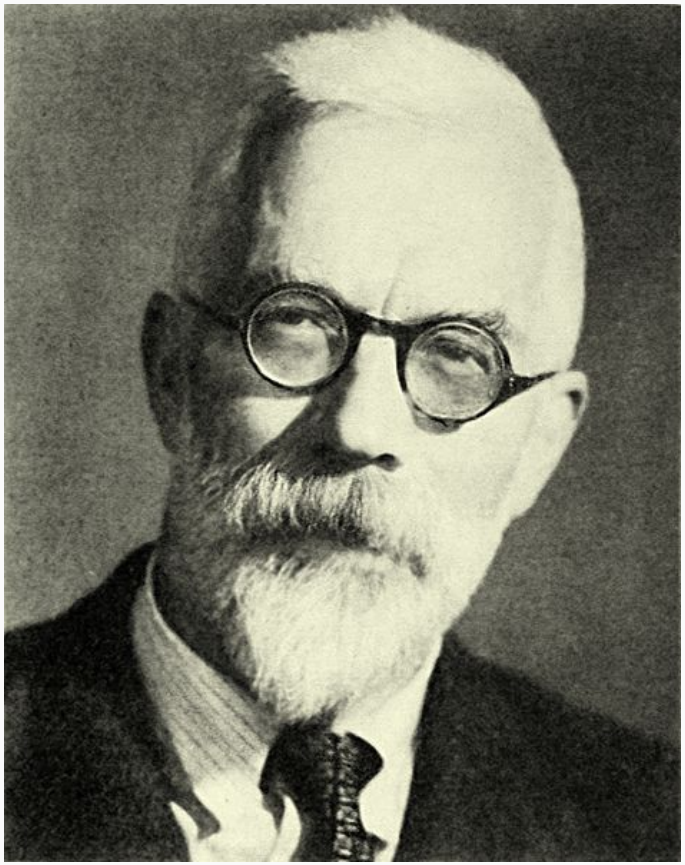
```
## [1] "Midfielder" "Defender"   "Forward"    "Goalkeeper"
```

```
worldcup <- worldcup %>%
  group_by(Position) %>%
  mutate(ave_shots = mean(Shots)) %>%
  ungroup() %>%
  mutate(Position = fct_reorder(Position, ave_shots))
levels(worldcup$Position)
```

```
## [1] "Goalkeeper" "Defender"   "Midfielder" "Forward"
```

Simulations

The lady tasting tea



Source: Flikr commons, https://www.flickr.com/photos/internetarchivebookimages/20150531109/

The lady tasting tea

"Dr. Muriel Bristol, a colleague of Fisher's, claimed that when drinking tea she could distinguish whether milk or tea was added to the cup first (she preferred milk first). To test her claim, Fisher asked her to taste eight cups of tea, four of which had milk added first and four of which had tea added first." — Agresti, Categorical Data Analysis, p.91

The lady tasting tea

Question:

- If she just guesses, what is the probability she will get all cups right?
- What if more or fewer cups are used in the experiment?

The lady tasting tea

One way to figure this out is to run a *simulation*.

In R, `sample` can be a very helpful function for simulations. It lets you randomly draw values from a vector, with or without replacement.

```
## Generic code
sample(x = [vector to sample from],
       size = [number of samples to take],
       replace = [logical-- should values in the
                  vector be replaced?],
       prob = [vector of probability weights])
```

The lady tasting tea

Create vectors of the true and guessed values, in order, for the cups of tea:

```
n_cups <- 8
cups <- sample(rep(c("milk", "tea"), each = n_cups / 2))
cups
```

```
## [1] "milk" "tea"  "milk" "milk" "tea"  "tea"  "tea"  "milk"
```

```
guesses <- sample(rep(c("milk", "tea"), each = n_cups / 2))
guesses
```

```
## [1] "milk" "milk" "tea"  "milk" "tea"  "milk" "tea"  "tea"
```

The lady tasting tea

For this simulation, determine how many cups she got right (i.e., guess equals the true value):

```
cup_results <- cups == guesses
cup_results
```

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

```
n_right <- sum(cup_results)
n_right
```

```
## [1] 4
```

The lady tasting tea

Right a function that will run one simulation. It takes the argument `n_cups`— in real life, they used eight cups (`n_cups = 8`). Note that this function just wraps the code we just walked through.

```
sim_null_tea <- function(n_cups){
  cups <- sample(rep(c("milk", "tea"), each = n_cups / 2))
  guesses <- sample(rep(c("milk", "tea"), each = n_cups / 2))
  cup_results <- cups == guesses
  n_right <- sum(cup_results)
  return(n_right)
}
sim_null_tea(n_cups = 8)
```

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

The lady tasting tea

Now, we need to run a lot of simulations, to see what happens on average if she guesses. You can use the `replicate` function to do that.

```
## Generic code
replicate(n = [number of replications to run],
          eval = [code to replicate each time])
```

```
tea_sims <- replicate(5, sim_null_tea(n_cups = 8))
tea_sims
```

```
## [1] 6 2 6 2 4
```

The lady tasting tea

This call gives a vector with the number of cups she got right for each simulation. You can replicate the simulation many times to get a better idea of what to expect if she just guesses, including what percent of the time she gets all cups right.

```
tea_sims <- replicate(1000, sim_null_tea(n_cups = 8))
mean(tea_sims)
```

```
## [1] 4.014
```

```
quantile(tea_sims, probs = c(0.025, 0.975))
```

```
##  2.5% 97.5%
##     2     6
```

```
mean(tea_sims == 8)
```

```
## [1] 0.014
```

The lady tasting tea

Now we'd like to know, for different numbers of cups of tea, what is the probability that the lady will get all right?

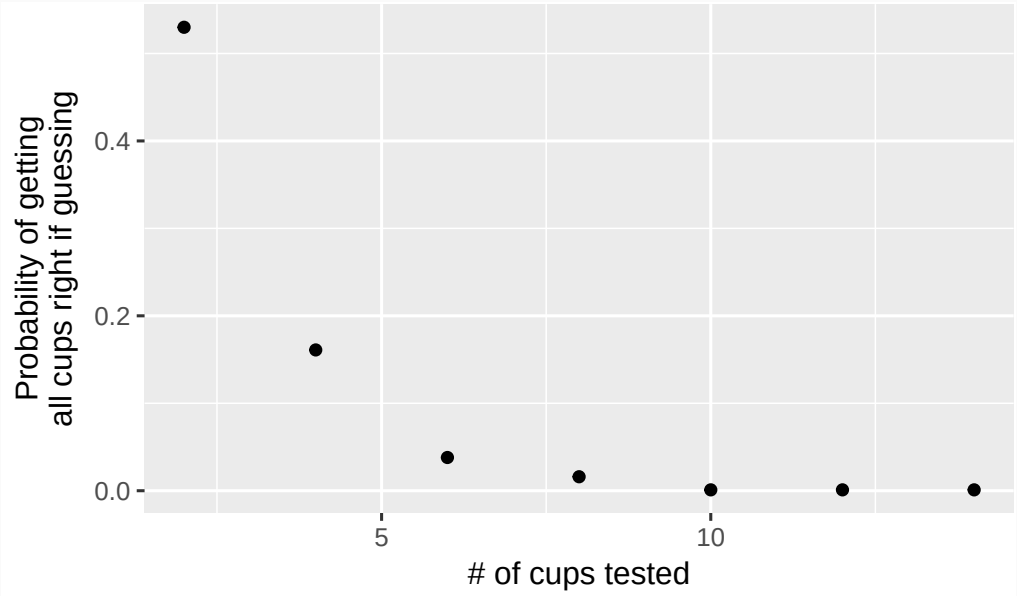
For this, we can apply the replication code across different values of `n_cups`:

```
n_cups <- seq(from = 2, to = 14, by = 2)
perc_all_right <- sapply(n_cups, FUN = function(n_cups){
  cups_right <- replicate(1000, sim_null_tea(n_cups))
  out <- mean(cups_right == n_cups)
  return(out)
})
perc_all_right
```

```
## [1] 0.530 0.161 0.038 0.016 0.001 0.001 0.001
```

The lady tasting tea

```
tea_sims <- data_frame(n_cups, perc_all_right)
ggplot(tea_sims, aes(x = n_cups, y = perc_all_right)) +
  geom_point() + xlab("# of cups tested") +
  ylab("Probability of getting\nall cups right if guessing")
```



The lady tasting tea

You can answer this question analytically using the hypergeometric distribution:

$$P(n_{11} = t) = \frac{\binom{n_{1+}}{t}\binom{n_{2+}}{n_{+1}-t}}{\binom{n}{n_{+1}}}$$

	Guessed milk	Guessed tea	Total
Really milk	n_{11}	n_{12}	$n_{1+} = 4$
Really tea	n_{21}	n_{22}	$n_{2+} = 4$
Total	$n_{+1} = 4$	$n_{+2} = 4$	$n = 8$

The lady tasting tea

In R, you can use `dhyper` to get the density of the hypergeometric function:

```
dhyper(x = [# of cups she guesses have milk first that do],
       m = [# of cups with milk first],
       n = [# of cups with tea first],
       k = [# of cups she guesses have milk first])
```

The lady tasting tea

Probability she gets three "milk" cups right if she's just guessing and there are eight cups, four with milk first and four with tea first:

```
dhyper(x = 3, m = 4, n = 4, k = 4)
```

```
## [1] 0.2285714
```

Probability she gets three or more "milk" cups right if she's just guessing:

```
dhyper(x = 3, m = 4, n = 4, k = 4) +
  dhyper(x = 4, m = 4, n = 4, k = 4)
```

```
## [1] 0.2428571
```

The lady tasting tea

Other density functions:

- `dnorm`: Normal
- `dpois`: Poisson
- `dbinom`: Binomial
- `dchisq`: Chi-squared
- `dt`: Student's t
- `dunif`: Uniform

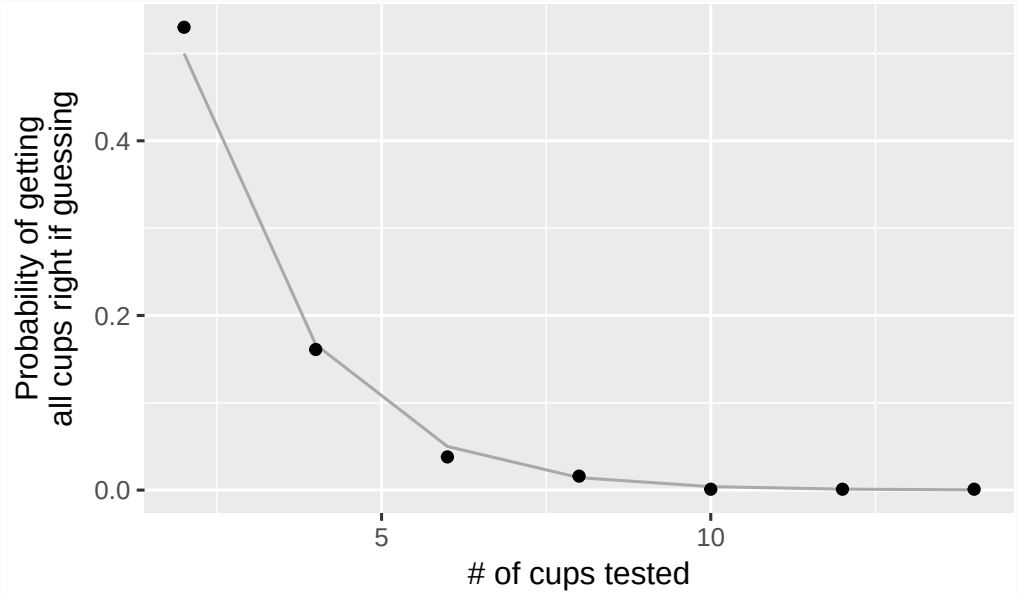
The lady tasting tea

You can get the analytical result for each of the number of cups we simulated and compare those values to our simulations:

```
analytical_results <- data_frame(n_cups = seq(2, 14, 2)) %>%
  mutate(perc_all_right = dhyper(x = n_cups / 2,
                                 m = n_cups / 2,
                                 n = n_cups / 2,
                                 k = n_cups / 2))
```

The lady tasting tea

```
ggplot(analytical_results, aes(x = n_cups, y = perc_all_right))
  geom_line(color = "darkgray") +
  geom_point(data = tea_sims) + xlab("# of cups tested") +
  ylab("Probability of getting\nall cups right if guessing")
```



The lady tasting tea

The Lady Tasting Tea: How Statistics Revolutionized Science in the Twentieth Century. David Salsburg.

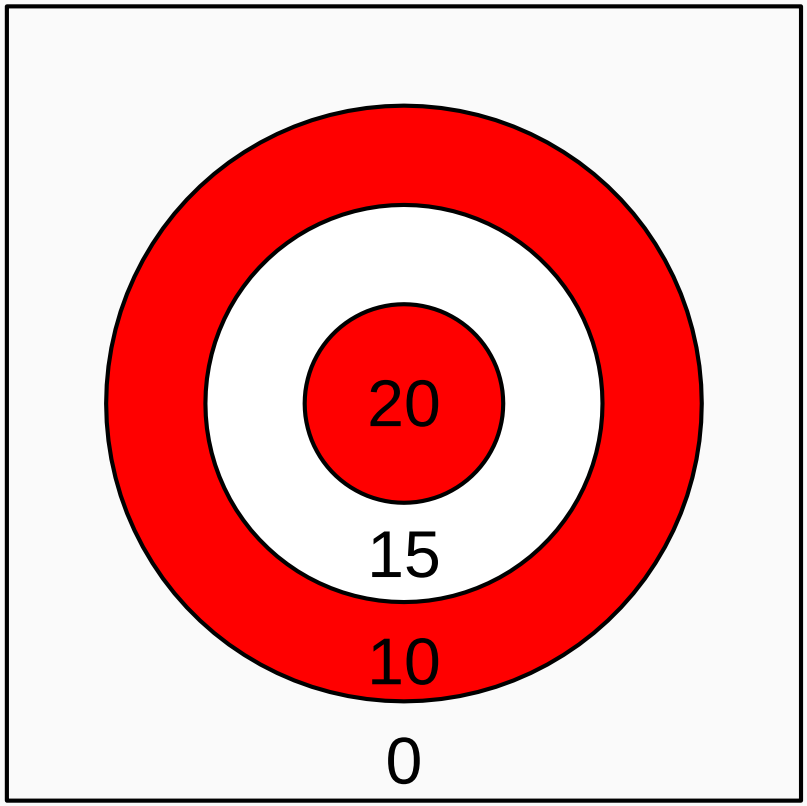
The Design of Experiments. Ronald Fisher.

`https://priceonomics.com/why-the-father-of-modern-statistics-didnt-believe/`

Playing darts

Research question: Is a person skilled at playing darts?

Here's our dart board– the numbers are the number of points you win for a hit in each area.



Playing darts

First, what would we expect to see if the person we test has no skill at playing darts?

Questions to consider:

- *What would the dart board look like under the null (say the person throws 20 darts for the experiment)?*
- *About what do you think the person's mean score would be if they had no skill at darts?*
- *What are some ways to estimate or calculate the expected mean score under the null?*

Playing darts

Let's use R to answer the first question: what would the null look like?

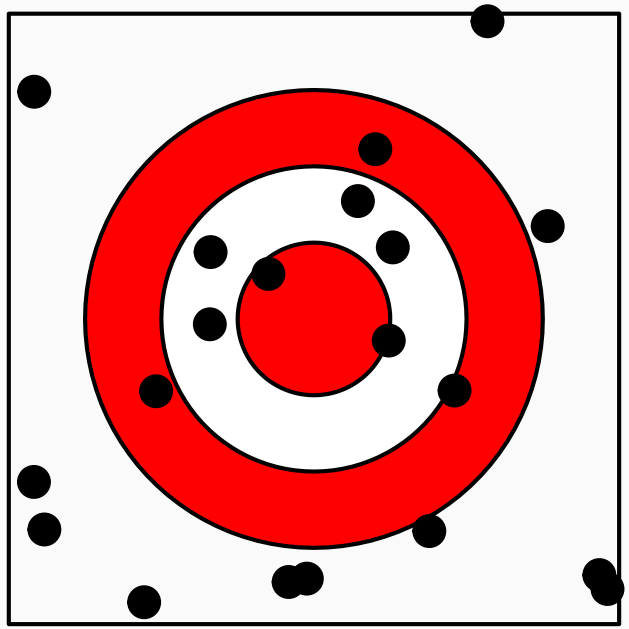
First, create some random throws (the square goes from -1 to 1 on both sides):

```
n.throws <- 20
throw.x <- runif(n.throws, min = -1, max = 1)
throw.y <- runif(n.throws, min = -1, max = 1)
head(cbind(throw.x, throw.y))
```

```
##          throw.x    throw.y
## [1,]  0.2583810  0.2343604
## [2,] -0.8833795 -0.6897532
## [3,] -0.9175728 -0.5340623
## [4,] -0.3386803  0.2190105
## [5,]  0.7660194  0.3041857
## [6,]  0.2011457  0.5560537
```

Playing darts

```
plot(c(-1, 1), c(-1,1), type = "n", asp=1,
     xlab = "", ylab = "", axes = FALSE)
rect( -1, -1, 1, 1)
draw.circle( 0, 0, .75, col = "red")
draw.circle( 0, 0, .5, col = "white")
draw.circle( 0, 0, .25, col = "red")
points(throw.x, throw.y, col = "black", pch = 19)
```



Playing darts

Next, let's tally up the score for this simulation of what would happen under the null.

To score each throw, we calculate how far the point is from (0, 0), and then use the following rules:

- **20 points**: $0.00 \leq \sqrt{x^2 + y^2} \leq .25$
- **15 points**: $0.25 < \sqrt{x^2 + y^2} \leq .50$
- **10 points**: $0.50 < \sqrt{x^2 + y^2} \leq .75$
- **0 points**: $0.75 < \sqrt{x^2 + y^2} \leq 1.41$

Playing darts

Use these rules to "score" each random throw:

```
throw.dist <- sqrt(throw.x^2 + throw.y^2)
head(throw.dist)
```

```
## [1] 0.3488345 1.1207671 1.0616790 0.4033236 0.8242054 0.59131
```

```
throw.score <- cut(throw.dist,
                   breaks = c(0, .25, .5, .75, 1.5),
                   labels = c("20", "15", "10", "0"),
                   right = FALSE)
head(throw.score)
```

```
## [1] 15 0  0  15 0  10
## Levels: 20 15 10 0
```

Playing darts

Now that we've scored each throw, let's tally up the total:

```
table(throw.score)
```

```
## throw.score
## 20 15 10  0
##  1  5  3 11
```

```
mean(as.numeric(as.character(throw.score)))
```

```
## [1] 6.25
```

Playing darts

So, this just showed *one* example of what might happen under the null. If we had a lot of examples like this (someone with no skill throwing 20 darts), what would we expect the mean scores to be?

Questions to consider:

- *How can you figure out the expected value of the mean scores under the null (that the person has no skill)?*
- *Do you think that 20 throws will be enough to figure out if a person's mean score is different from this value, if he or she is pretty good at darts?*
- *What steps do you think you could take to figure out the last question?*
- *What could you change about the experiment to make it easier to tell if someone's skilled at darts?*

Playing darts

How can we figure this out?

- **Theory.** Calculate the expected mean value using the expectation formula
- **Simulation.** Simulate a lot of examples using R and calculate the mean of the mean score from these.

Playing darts

The expected value of the mean, $E[\bar{X}]$, is the expected value of X, $E[X]$. To calculate the expected value of X, use the formula:

$$E[X] = \sum_x xp(x)$$

$$E[X] = 20 * p(X = 20) + 15 * p(X = 15) + 10 * p(X = 10) + 0 * p(X = 0)$$

So we just need to figure out $p(X = x)$ for $x = 20, 15, 10$.

Playing darts

(In all cases, we're dividing by 4 because that's the area of the full square, 2^2.)

- $p(X = 20)$: Proportional to area of the smallest circle, $(\pi * 0.25^2)/4 = 0.049$
- $p(X = 15)$: Proportional to area of the middle circle minus area of the smallest circle, $\pi(0.50^2 - 0.25^2)/4 = 0.147$
- $p(X = 10)$: Proportional to area of the largest circle minus area of the middle circle, $\pi(0.75^2 - 0.50^2)/4 = 0.245$
- $p(X = 0)$: Proportional to area of the square minus area of the largest circle, $(2^2 - \pi * 0.75^2)/4 = 0.558$

As a double check, if we've done this write, the probabilities should sum to 1:

$$0.049 + 0.147 + 0.245 + 0.558 = 0.999$$

Playing darts

$$E[X] = \sum_x xp(x)$$

$$E[X] = 20 * 0.049 + 15 * 0.147 + 10 * 0.245 + 0 * 0.558$$

$$E[X] = 5.635$$

Remember, this also gives us $E[\bar{X}]$.

Playing darts

Now it's pretty easy to also calculate $var(X)$ and $var(\bar{X})$:

$$Var(X) = E[(X - \mu)^2] = E[X^2] - E[X]^2$$

$$E[X^2] = 20^2 * 0.049 + 15^2 * 0.147 + 10^2 * 0.245 + 0^2 * 0.558 = 77.18$$

$$Var(X) = 77.175 - (5.635)^2 = 45.42$$

$$Var(\bar{X}) = \sigma^2/n = 45.42/20 = 2.27$$

Playing darts

Now that we can use the Central Limit Theorem to calculate a 95% confidence interval for the mean score when someone with no skill (null hypothesis) throws 20 darts:

```
5.635 + c(-1, 1) * qnorm(.975) * sqrt(2.27)
```

```
## [1] 2.682017 8.587983
```

Playing darts

We can check our math by running simulations– we should get the same values of $E[\bar{X}]$ and $Var(\bar{X})$ (which we can calculate directly from the simulations using R).

```
n.throws <- 20
n.sims <- 10000

x.throws <- matrix(runif(n.throws * n.sims, -1, 1),
                   ncol = n.throws, nrow = n.sims)
y.throws <- matrix(runif(n.throws * n.sims, -1, 1),
                   ncol = n.throws, nrow = n.sims)
dist.throws <- sqrt(x.throws^2 + y.throws^2)
score.throws <- apply(dist.throws, 2, cut,
                   breaks = c(0, .25, .5, .75, 1.5),
                   labels = c("20", "15", "10", "0"),
                   right = FALSE)
```

Playing darts

```
dist.throws[1:3,1:5]
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.7947828 0.4095746 1.0260358 0.2683331 1.2395218
## [2,] 0.4294334 0.6737229 0.8628415 0.2513240 0.9532781
## [3,] 0.3762356 0.9058099 0.6788584 0.8690350 0.8505576
```

```
score.throws[1:3,1:5]
```

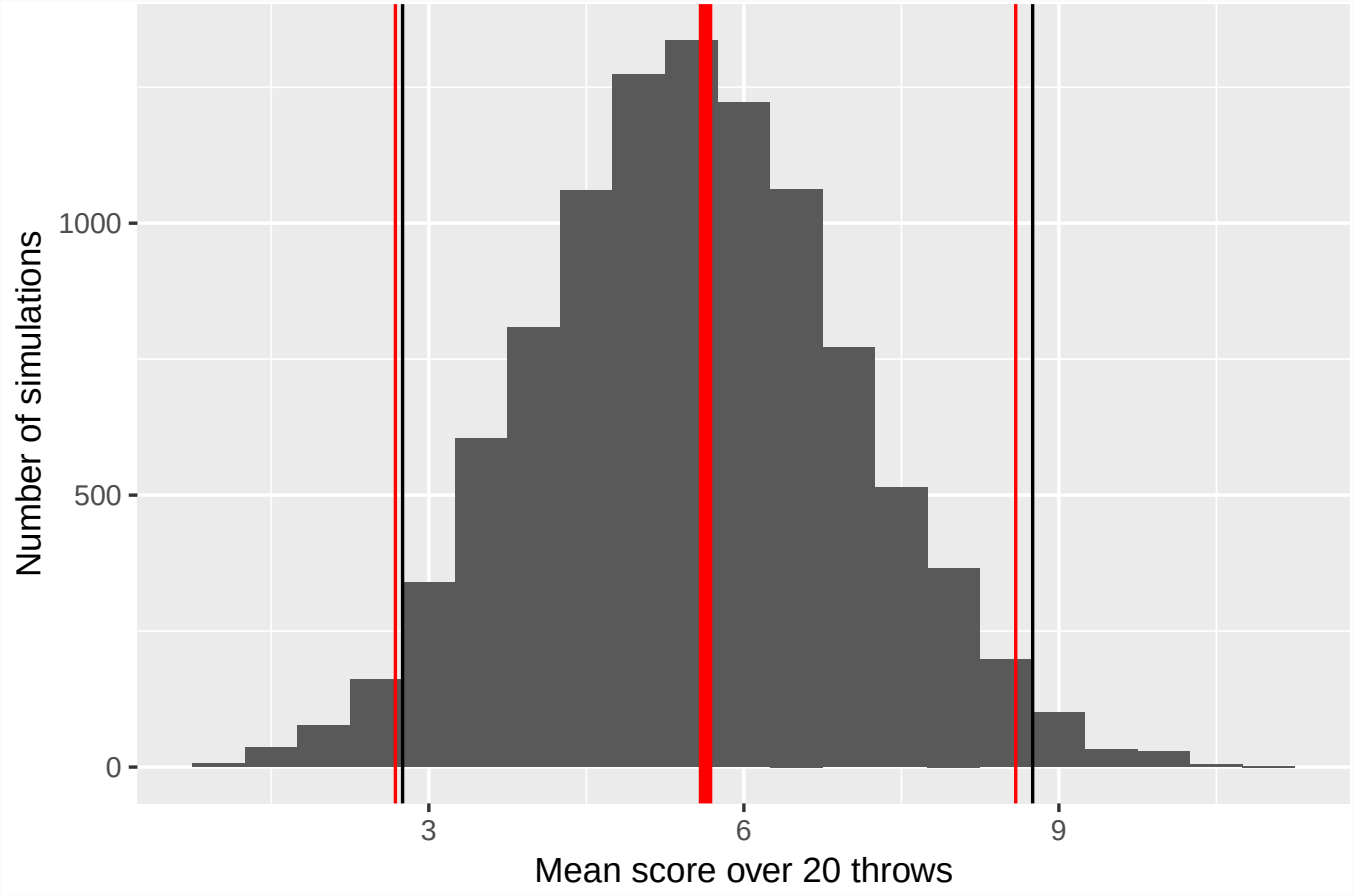
```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "0"  "15" "0"  "15" "0"
## [2,] "15" "10" "0"  "15" "0"
## [3,] "15" "0"  "10" "0"  "0"
```

Playing darts

```
mean.scores <- apply(score.throws, MARGIN = 1,
                     function(x){
                             out <- mean(as.numeric(
                                     as.character(x)))
                             return(out)
                     })
head(mean.scores)
```

```
## [1] 6.75 8.25 6.50 6.00 7.25 7.00
```

Playing darts



Playing darts

Let's check the simulated mean and variance against the theoretical values:

```
mean(mean.scores) ## Theoretical: 5.635
```

```
## [1] 5.63725
```

```
var(mean.scores) ## Theoretical: 2.27
```

```
## [1] 2.236911
```

Simulations in research

Simulations in the wild (just a few examples):

- The Manhattan Project
- US Coast Guard search and rescue
- Infectious disease modeling

Other computationally-intensive approaches

Bootstrap and friends

- **Bootstraping:** Sample the data set with replacement and re-estimate the statistical parameter(s) each time.
- **Jackknifing:** Rake out one observation at a time and re-estimate the statistical parameter(s) with the rest of the data.
- **Permutation tests:** See how unusual the result from the data is compared to if you shuffle your data (and so remove any relationship in observed data between variables).
- **Cross-validation:** See how well your model performs if you pick a subset of the data, build the model just on that subset, and then test how well it predicts for the rest of the data, and repeat that many times.

Bayesian analysis

Suggested books for learning more about Bayesian analysis in R:

- *Doing Bayesian Data Analysis, Second Edition: A Tutorial with R, JAGS, and Stan.* John Kruschke.
- *Statistical Rethinking: A Bayesian Course with Examples in R and Stan.* Richard McElreath.
- *Bayesian Data Analysis, Third Edition.* Andrew Gelman et al.

R can tap into software for Bayesian analysis:

- BUGS
- JAGS
- STAN

Ensemble models and friends

- **Bagging:** Sample data with replacement and build a tree model. Repeat many times. To predict, predict from all models and take the majority vote.
- **Random forest:** Same as bagging, for picking each node of a tree, only consider a random subset of variables.
- **Boosting:** Same as bagging, but "learn" from previous models as you build new models.
- **Stacked models:** Build many different models (e.g., generalized linear regression, Naive Bayes, k-nearest neighbors, random forest, ...), determine weights for each, and predict using weighted predictions combined from all models

Ensemble models and friends

For more on these and other machine learning topics, see:

An Introduction to Statistical Learning. Gareth James, Robert Tibshirani, and Trevor Hastie.

The `caret` package: `http://topepo.github.io/caret/index.html`

For many examples of predictive models like this built with R (and Python): `https://www.kaggle.com`

Speeding up R

When to worry about this

Not until you need it!

$$\text{Total time} = \text{Time to write code} + \text{Time to run code}$$

Optimizing code means it takes longer to write the code. Often, even if it's slow to compute, it's still faster overall to not take the time to write faster code.

When to worry about this

- Data with $>$ 1,000,000 observations (memory + code speed)
- Running complex models many, many times (code speed)
- Large data created by the analytic methods (memory + code speed)

Compiled code

Compiled languages:

- C
- C++: `Rcpp`
- Java: `rJava`
- Fortran subroutine

Compiled code

All require a *compiler* (e.g., gcc) on your computer to compile the code. You can get what you need with:

- Windows: RTools
- Mac: XCode Tools

Compiled code

Typical steps in using compiled code in R:

1. Create a shared object file (.so)
2. Use `dyn.load` to load and `dyn.unload` to unload shared objects.
3. Call function (`.Fortran`, `.Call`)
4. Write a function in R that loads and calls the function, so you have a nicer interface for doing this.

Profiling R code

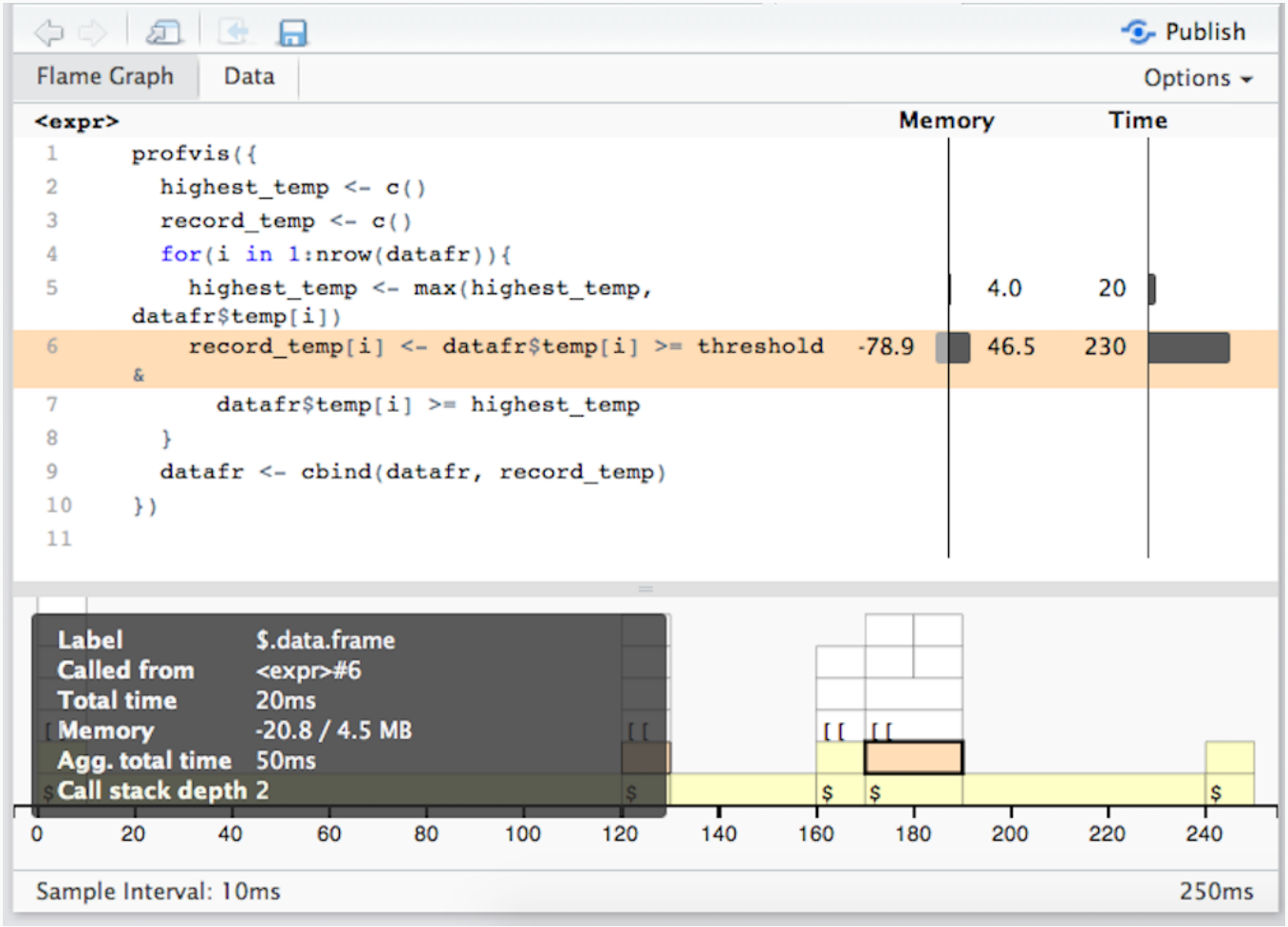
Strategy: Look for bottlenecks in the code, and then rewrite small sections of code that are bottlenecks in a compiled language.

You can use *code profiling* to identify bottlenecks in your code. The best option is `profvis` (currently in development version of RStudio).

```
## Generic code
library(profvis)
profvis({
  [Code you want to profile]
})
```

Profiling R code

The `profvis` package allows you to profile R code, and results are much easier to navigate and interpret than older methods of profiling R code.



Rcpp

Currently, the best tool for using code from a compiled language in R is `Rcpp`, which works with C++ code. Advantages include:

- Stay in R the whole time– no need to compile from the command line
- Compiles and loads to R with one functions (`sourceCpp`)
- Under active development

Get a bigger machine

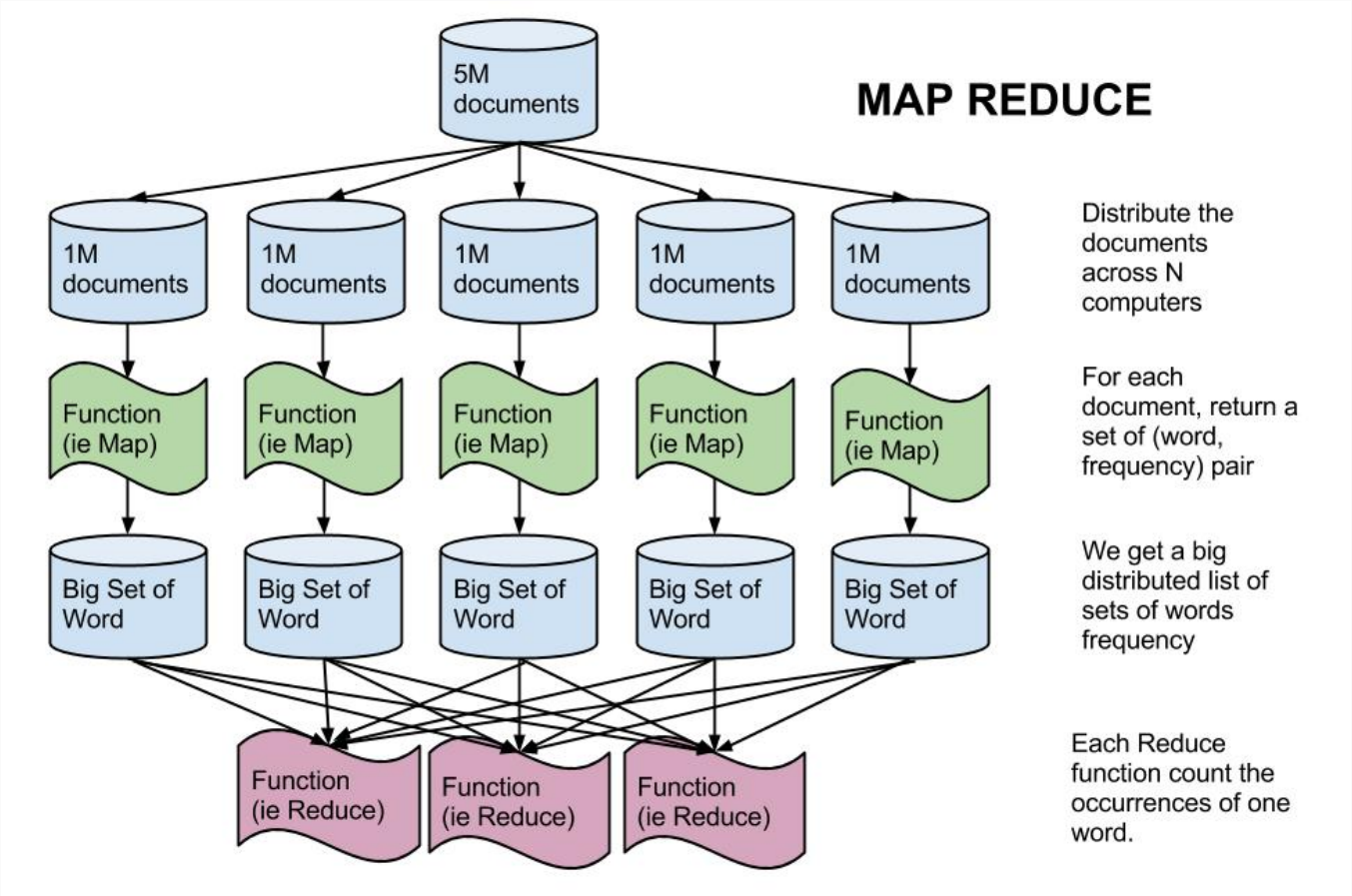
Another way to improve R speed is to improve hardware. Options include:

- Personal computer with more memory and speed
- Server (R Studio server)
- Cluster
- Cloud services (Amazon Web Services, EC2 Instance; `RAmazonS3`)

Many of these options may require you to run R in *batch mode* rather than interactively. In other words, you'll need to write all your R code in an R script and then source the script using something like `R CMD BATCH` or `Rscript` from the command line, rather than running code line-by-line.

Parallel computing

You can also improve speed by running code in parallel. This is particularly powerful on a large cluster or cloud services.



Source: `http://gerardnico.com/wiki/algorithm/map_reduce`

Parallel computing

In R, there are some packages for setting up parallel processing for your custom code. Examples include:

- `parallel`
- `foreach`

Some packages include the option to run internal code in parallel. For example:

- `caret`
- `GAMBoost`

sparklyr

The `sparklyr` package was just released by RStudio earlier this fall: `http://spark.rstudio.com`.

It allows you to connect R with Spark (open source framework for cluster computing), but still use `dplyr` for all your code. It also connects to some distributed machine learning systems (e.g., H2O Sparkling Water), so you can build large ensemble models with big data.

GPUs

Graphics Processing Units (GPUs)— historically, closely tied to computer gaming systems.

> *"GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications. Pioneered in 2007 by NVIDIA, GPU accelerators now power energy-efficient data centers in government labs, universities, enterprises, and small-and-medium businesses around the world. They play a huge role in accelerating applications in platforms ranging from artificial intelligence to cars, drones, and robots." –Nvidia*

Source:
`http://www.nvidia.com/object/what-is-gpu-computing.html`

GPUs

Code for GPUs is written in a different programming language (e.g., CUDA). There are a few packages that provide an R interface:

- `gputools`
- `cudaBayesreg` (application: analyzing fMRI data)
- `gmatrix`
- `gpuR`

This is a growing area of development in R, so look for more packages in the future.

Working with large datasets

data.table

`data.table` is a package in R that can efficiently read in and manipulate large data sets. It offers a **substantial** speed improvement over the classic `data.frame` when working with large data sets.

Example: US precipitation

As an example, I have a file with daily precipitation measures for every US county from 1979 through 2011:

- 365 days * 33
- ~3,000 counties

This file has $>$ 37,000,000 lines. The total file size is 2.26 GB.

Reading in a large text file

`fread` is the `data.table` equivalent of the `read.table` family of functions:

```
library(data.table)
system.time(precip <- fread(paste0(precip_dir,
                        "nasa_precip_export_2.txt"),
                       header = TRUE,
                       select = c("county",
                                  "year_month_day",
                                  "precip"),
                       verbose = FALSE))
dim(precip)
```

Reading in a large text file

`fread` can also read a file directly from http and https URLs, if you'd prefer to not save the flat file locally.

Manipulating a `data.table`

The `data.table` class has a series of conventions for summarizing and indexing that runs much, much faster than if you tried to use "classic" R functions.

The general form is:

```
precip[i, j, by]
```

where `i` filters by row, `j` selects or calculates on columns, and `by` groups by some grouping variable when selecting or calculating using columns.

Manipulating a `data.table`

You can use the first element to filter to certain rows. For example, to pull out just values for Larimer County, CO, run:

```
precip[county == 8069 &
         year_month_day %in%
         c(19970727, 19970728), ]
```

Manipulating a `data.table`

You can use the `order` function in the first element to sort the data:

```
head(precip[order(-precip), ])
```

Manipulating a `data.table`

You can run calculations on columns using the second element:

```
precip[ , max(precip)]
precip[ , quantile(precip,
                   probs = c(0.99, 0.999,
                             0.9999))]
```

Manipulating a `data.table`

You can combine filtering by rows and calculating on columns. For example, to figure out how many counties there were in 2011:

```
precip[year_month_day == 20110101,
       length(precip)]
```

Note: If you want to count rows, you can also use `.N`:

```
precip[year_month_day == 20110101,
       .N]
```

Grouped analysis

You can also group by a variable before you run an analysis. For example, to get the highest recorded precipitation in each county:

```
highest_precip <- precip[ , .(max.precip = max(precip)),
                          by = .(county)]
head(highest_precip, 3)
```

Highest precipitation by county

Chaining operation with `data.table`

If you want to, you can chain together several operations. For example, to determine the number of days over the 99.9th percentile in each county:

```
extreme_precip <- precip[ , .N, .(precip >
                                    quantile(precip,
                                             probs = 0.999),
                                  county)][
                    precip == TRUE,
                  ]
```

Extreme precipitation by county

Chaining operation with `data.table`

To plot trends by month within states:

```
ts_precip <- precip[ , .(precip = precip,
                         state = substring(sprintf("%05d",
                                                   county),
                                           1, 2),
                         month = as.numeric(
                           substring(year_month_day,
                                           5, 6)))][
                       , .(precip = mean(precip)),
                       keyby = .(state, month)
                       ]
```

Precipitation by month and state

Databases

R also allows you to work with data stored in a database. There are many different types of databases; the `DBI` package provides a generic interface to many of them.

Outside of memory data

One way to work with really large data is to avoid loading it into R. Some R packages help with this:

- `ff`
- `ffbase`
- `biglm`
- `bigmemory`

Some file formats allow you to read just part of the data into memory. For example:

- NetCDF: `RNetCDF`, `ncdf4`
- HDF5: `h5`, `rhdf5` (Bioconductor)

Find out more

- *Advanced R*. Hadley Wickham. (Free online, see especially the chapter on `Rcpp`: `http://adv-r.had.co.nz/Rcpp.html`)
- *Seamless R and C++ Integration with Rcpp.* Dirk Eddelbuettel. (Free online through CSU library)
- CRAN High Performance Computing Task View: `https://cran.r-project.org/web/views/HighPerformanceComputing.html`
- Rcpp website: `http://www.rcpp.org`
- *Statistical Computing in C++ and R.* Eubank and Kupresanin.