NC STATE UNIVERSITY

Programming in R Part II

Justin Post

August 9, 2017

What do we want to be able to do?

- Restructure Data/Clean Data
- Streamline repeated sections of code
- **Improve efficiency of code**
- Write custom functions to simplify code

Efficient Code

For loops inefficient in R

- R interpreted language
- Must figure out how to evaluate code at each iteration of loop
- Slows it down

Efficient Code

For loops inefficient in R

- R interpretted language
- Must figure out how to evaluate code at each iteration of loop
- Slows it down

Vecotrized functions much faster!

- Vectorized function: works on entire vector at once
- Avoids costly computation time

Efficient Code

Some 'built-in' vectorized functions

- `colMeans()`, `rowMeans()`
- `colSums()`, `rowSums()`
- `colSds()`, `colVars()`, `colMedians()` (`matrixStats` package)
- `ifelse()`
- `apply()` family
- Create your own with `Vectorize()`

5/84

Efficient Code

- Find column means for full Batting data set
- `colMeans()` just requires a numeric data frame (array)

```
colMeans(select(Batting, G:GIDP), na.rm = TRUE)
```

```
##          G         AB          R          H        X2B        X3B
##  51.400111 149.970327  19.887038  39.261647   6.637067   1.373361
##         HR        RBI         SB         CS         BB         SO
##   2.949305  17.965163   3.158184   1.324025  13.811484  21.629849
##        IBB        HBP         SH         SF       GIDP
##   1.213234   1.113395   2.457900   1.150122   3.210032
```

Efficient Code

- Compare computational time
- `microbenchmark` package allows for easy recording of computing time

```
install.packages("microbenchmarK")
```

```
library(microbenchmark)
```

Efficient Code

- Compare computational time

```
microbenchmark(
  colMeans(select(Batting, G:GIDP), na.rm = TRUE)
)
```

```
## Unit: milliseconds
##                                             expr     min       lq   mean
##  colMeans(select(Batting, G:GIDP), na.rm = TRUE) 10.5278 11.00802 13.866
##    median       uq      max neval
##  11.46476 15.06337 67.50874   100
```

Efficient Code

- Compare computational time

```
microbenchmark(
  for(i in 1:17){
    mean(Batting[ , i + 5][[1]], na.rm = TRUE)
  }
)
```

```
## Unit: milliseconds
##                                                            expr     min
##  for (i in 1:17) {     mean(Batting[, i + 5][[1]], na.rm = TRUE) } 2.67567
##        lq     mean   median       uq      max neval
##  2.909959 3.303627 3.092535 3.423539 6.953662   100
```

Efficient Code

- With vectorized functions, can easily find cool stuff
- Median number of games played for all players
- Median number of AB for players that batted
- Steps: (think `dplyr` commands!)

1. Group observations by playerID
2. Summarise variables of interest
3. Remove non numeric column
4. Coerce to matrix for use in `colMedians()`
5. Use `colMedians()` function

Efficient Code

```
library(matrixStats) #install if not installed
```

```
##
## Attaching package: 'matrixStats'
```

```
## The following object is masked from 'package:dplyr':
##
##     count
```

```
Batting %>% group_by(playerID) %>%
  summarise(totG = sum(G), totAB = sum(AB)) %>%
  select(-playerID) %>% as.matrix() %>%
  colMedians(na.rm = TRUE)
```

```
## [1] 78 91
```

Efficient Code

- Logical statement - comparison of two quantities
  - resolves as TRUE or FALSE
- Often want to execute code logically
- logical comparison operators
  - `==`, `!=`, `>=`, `<=`, `>`, `<`
  - `&` "and"
  - `|` "or"
- logical functions
  - `is.` family (`is.numeric()`, `is.data.frame()`, etc.)

Efficient Code

If then, If then else

- Often want to execute statements conditionally
- If then else concept

```
if (condition) {
  then execute code
}

#if then else
if (condition) {
  execute this code
} else {
  execute this code
}
```

Efficient Code

If then, If then else

- Often want to execute statements conditionally
- If then else concept

```
#Or more if statements
if (condition) {
  execute this code
} else if (condition2) {
  execute this code
} else if (condition3) {
  execute this code
} else {
  #if no conditions met
  execute this code
}
```

Efficient Code

- Often create new variables

Efficient Code

- Often create new variables
- Built in data set `airquality`
  - daily air quality measurements in New York
  - from May (Day 1) to September (Day 153) in 1973

```
airquality<-tbl_df(airquality)
airquality
```

```
## # A tibble: 153 x 6
##   Ozone Solar.R  Wind  Temp Month   Day
##   <int>   <int> <dbl> <int> <int> <int>
## 1    41     190   7.4    67     5     1
## 2    36     118   8.0    72     5     2
## 3    12     149  12.6    74     5     3
## 4    18     313  11.5    62     5     4
## 5    NA      NA  14.3    56     5     5
## # ... with 148 more rows
```

Efficient Code

Want to code a wind category variable

- high wind days (15mph $\leq$ wind)
- windy days (10mph $\leq$ wind < 15mph)
- lightwind days (6mph $\leq$ wind < 10mph)
- calm days (wind $\leq$ 6mph)

Efficient Code

Want to code a wind category variable

- high wind days (15mph $\leq$ wind)
- windy days (10mph $\leq$ wind < 15mph)
- lightwind days (6mph $\leq$ wind < 10mph)
- calm days (wind $\leq$ 6mph)

Initial plan

- loop through each observation
- use if then else to determine wind status

Efficient Code

```r
#initialize vector to save results
status<-vector()

for (i in 1:(dim(airquality)[1])){
  if(airquality$Wind[i] >= 15){
    status[i] <- "HighWind"
  } else if (airquality$Wind[i] >= 10){
    status[i] <- "Windy"
  } else if (airquality$Wind[i] >= 6){
    status[i] <- "LightWind"
  } else if (airquality$Wind[i] >= 0){
    status[i] <- "Calm"
  } else {
    status[i] <- "Error"
  }
}
```

Efficient Code

```
status
```

```
##   [1] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
##   [6] "Windy"     "LightWind" "Windy"     "HighWind"  "LightWind"
##  [11] "LightWind" "LightWind" "LightWind" "Windy"     "Windy"
##  [16] "Windy"     "Windy"     "HighWind"  "Windy"     "LightWind"
##  [21] "LightWind" "HighWind"  "LightWind" "Windy"     "HighWind"
##  [26] "Windy"     "LightWind" "Windy"     "Windy"     "Calm"
##  [31] "LightWind" "LightWind" "LightWind" "HighWind"  "LightWind"
##  [36] "LightWind" "Windy"     "LightWind" "LightWind" "Windy"
##  [41] "Windy"     "Windy"     "LightWind" "LightWind" "Windy"
##  [46] "Windy"     "Windy"     "HighWind"  "LightWind" "Windy"
##  [51] "Windy"     "LightWind" "Calm"      "Calm"      "LightWind"
##  [56] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
##  [61] "LightWind" "Calm"      "LightWind" "LightWind" "Windy"
##  [66] "Calm"      "Windy"     "Calm"      "LightWind" "Calm"
##  [71] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
##  [76] "Windy"     "LightWind" "Windy"     "LightWind" "Calm"
##  [81] "Windy"     "LightWind" "LightWind" "Windy"     "LightWind"
##  [86] "LightWind" "LightWind" "Windy"     "LightWind" "LightWind"
```

20/84

Efficient Code

- Add it to the data set

```
airquality$status <- status
```

- Find mean temperature for each wind Status

```
airquality$status <- status
airquality %>% group_by(status) %>%
  mutate(avgTemp = mean(Temp))
```

Efficient Code

```
## # A tibble: 153 x 8
## # Groups:   status [4]
##   Ozone Solar.R  Wind  Temp Month   Day    status  avgTemp
##   <int>   <int> <dbl> <int> <int> <int>     <chr>    <dbl>
## 1    41     190   7.4    67     5     1 LightWind 79.43077
## 2    36     118   8.0    72     5     2 LightWind 79.43077
## 3    12     149  12.6    74     5     3     Windy 75.54839
## 4    18     313  11.5    62     5     4     Windy 75.54839
## 5    NA      NA  14.3    56     5     5     Windy 75.54839
## # ... with 148 more rows
```

Efficient Code

- Know for loops not great
- `ifelse()` is vectorized version of `if then else`
- Syntax

```
ifelse(vector_condition, if_true_do_this, if_false_do_this)
```

Efficient Code

```
ifelse(airquality$Wind >= 15, "HighWind",
          ifelse(airquality$Wind >= 10, "Windy",
                 ifelse(airquality$Wind >= 6, "LightWind", "Calm")))
```

```
##   [1] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
##   [6] "Windy"     "LightWind" "Windy"     "HighWind"  "LightWind"
##  [11] "LightWind" "LightWind" "LightWind" "Windy"     "Windy"
##  [16] "Windy"     "Windy"     "HighWind"  "Windy"     "LightWind"
##  [21] "LightWind" "HighWind"  "LightWind" "Windy"     "HighWind"
##  [26] "Windy"     "LightWind" "Windy"     "Windy"     "Calm"
##  [31] "LightWind" "LightWind" "LightWind" "HighWind"  "LightWind"
##  [36] "LightWind" "Windy"     "LightWind" "LightWind" "Windy"
##  [41] "Windy"     "Windy"     "LightWind" "LightWind" "Windy"
##  [46] "Windy"     "Windy"     "HighWind"  "LightWind" "Windy"
##  [51] "Windy"     "LightWind" "Calm"      "Calm"      "LightWind"
##  [56] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
##  [61] "LightWind" "Calm"      "LightWind" "LightWind" "Windy"
##  [66] "Calm"      "Windy"     "Calm"      "LightWind" "Calm"
##  [71] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
##  [76] "Windy"     "LightWind" "Windy"     "LightWind" "Calm"
```

Efficient Code

- Compare speed

```
loopTime<-microbenchmark(
  for (i in 1:(dim(airquality)[1])){
    if(airquality$Wind[i] >= 15){
      status[i] <- "HighWind"
    } else if (airquality$Wind[i] >= 10){
      status[i] <- "Windy"
    } else if (airquality$Wind[i] >= 6){
      status[i] <- "LightWind"
    } else if (airquality$Wind[i] >= 0){
      status[i] <- "Calm"
    } else{
      status[i] <- "Error"
    }
  }
, unit = "us")
```

Efficient Code

- Compare speed

```
vectorTime <- microbenchmark(
  ifelse(airquality$Wind >= 15, "HighWind",
            ifelse(airquality$Wind >= 10, "Windy",
                   ifelse(airquality$Wind >= 6, "LightWind", "Calm")))
, unit = "us")
```

Efficient Code (Note units!)

```
loopTime
```

```
## Unit: microseconds
##
##  for (i in 1:(dim(airquality)[1])) {     if (airquality$Wind[i] >= 15) {         status[i
##       min       lq     mean   median       uq      max neval
##  29039.58 31981.91 34859.96 33797.79 36785.91 61407.75   100
```

```
vectorTime
```

```
## Unit: microseconds
##
##  ifelse(airquality$Wind >= 15, "HighWind", ifelse(airquality$Wind >=      10, "Windy", if
##      min       lq     mean   median       uq      max neval
##  287.385 400.0875 691.6916 555.6225 670.6945 6716.412   100
```

27/84

Efficient Code

- `apply()` family of functions *pretty* fast
- Check `help(apply)`
  - We'll look at `apply()`, `sapply()`, `lapply()`

Efficient Code

- `apply()` family of functions *pretty* fast
- Check `help(apply)`
    - We'll look at `apply()`, `sapply()`, `lapply()`
        - Use `apply()` to find summary for columns of airquality data

```
apply(X = select(airquality, Ozone:Temp), MARGIN = 2,
      FUN = summary, na.rm = TRUE)
```

Efficient Code

- Keeps data numeric, keeps labels!

```
## $Ozone
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##    1.00   18.00   31.50   42.13   63.25  168.00      37 
## 
## $Solar.R
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##     7.0   115.8   205.0   185.9   258.8   334.0       7 
## 
## $Wind
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   1.700   7.400   9.700   9.958  11.500  20.700 
## 
## $Temp
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   56.00   72.00   79.00   77.88   85.00   97.00
```

Efficient Code

- Use `lapply()` to apply function to lists
- Obtain a list object

```
fit <- lm(Ozone ~ Wind, data = airquality)
fit <- list(fit$residuals, fit$effects, fit$fitted.values)
```

Efficient Code

```
fit[[1]]
```

```
##          1          2          3          4          6          7 
## -14.7960653 -16.4655116 -14.9312663 -15.0372815  13.8358563 -26.1349578 
##          8          9         11         12         13         14 
##  -1.2701589  22.7006553 -51.5715267 -27.0289427 -34.8044041 -22.3678352 
##         15         16         17         18         19         20 
##  -5.6007126 -19.0372815   3.7381799  11.2640864  -3.0372815 -32.0289427 
##         21         22         23         24         28         29 
## -42.0289427   6.2724252 -39.0289427   1.7381799  -7.2618201  30.8358563 
##         30         31         38         40         41         44 
##  49.7673658 -18.7960653 -14.0289427  50.7298411   5.9627185 -29.4655116 
##         47         48         49         50         51         62 
##   6.8358563  55.0312090 -25.8044041 -21.0372815 -26.6983889  60.8858892 
##         63         64         66         67         68         69 
##   3.1955959 -13.8044041  -7.3386494   3.6321648   8.4368121  35.0979195 
##         70         71         73         74         76         77 
##  31.7673658  29.2039347  -7.4946974  12.8358563 -10.4946974 -10.5715267 
##         78         79         80         81         82         85 
##  -4.6983889  -0.9020805  10.4368121  29.9627185 -42.5715267  30.8650422 
```

32/84

Efficient Code

```
fit[[2]]
```

```
##  (Intercept)         Wind                                      
## -453.7465588 -212.8004841  -14.7950343  -14.4827016   13.0973609 
##                                                                  
##  -24.4774608   -1.5903064   19.9845153  -49.2674921  -25.7897935 
##                                                                  
##  -33.3750969  -21.5850656   -5.6926704  -18.4827016    4.1026017 
##                                                                  
##    9.1944841   -2.4827016  -30.7897935  -40.7897935    4.8873922 
##                                                                  
##  -37.7897935    2.1026017   -6.8973983   30.0973609   52.5277800 
##                                                                  
##  -16.6821888  -12.7897935   50.4096936    6.5172984  -27.5798248 
##                                                                  
##    6.0973609   52.0868793  -24.3750969  -20.4827016  -25.6874296 
##                                                                  
##   64.2548094    4.6249031  -12.3750969   -4.1598873    4.4149344 
##                                                                  
##   11.4254160   37.6301439   34.5277800   31.3178112   -8.0050031 
```

33/84

Efficient Code

- Apply `mean()` function to each list element

```
lapply(X = fit, FUN = mean)
```

```
## [[1]]
## [1] -5.731915e-16
##
## [[2]]
## [1] -4.333566
##
## [[3]]
## [1] 42.12931
```

Efficient Code

- Use `sapply()` similar but returns a vector if possible

```
sapply(X = fit, FUN = mean)
```

```
## [1] -5.731915e-16 -4.333566e+00  4.212931e+01
```

- `apply()` functions not as good as `colMeans()` type functions

```
air2 <- select(airquality, Ozone:Day)
microbenchmark(apply(X = air2, MARGIN = 2, FUN = mean, na.rm = TRUE))
```

```
## Unit: microseconds
##                                                     expr     min       lq
##  apply(X = air2, MARGIN = 2, FUN = mean, na.rm = TRUE) 135.008 138.9555
##      mean   median       uq      max neval
##  193.8308 142.7055 161.4565 1371.389   100
```

```
microbenchmark(colMeans(air2, na.rm = TRUE))
```

```
## Unit: microseconds
##                             expr    min     lq     mean median    uq     max
##  colMeans(air2, na.rm = TRUE) 58.425 60.004 64.19629 60.794 62.57 187.511
##  neval
##    100
```

36/84

Recap!

- Vectorized functions fast!
- 'Built-in' vectorized functions
  - `colMeans()`, `rowMeans()`
  - `colSums()`, `rowSums()`
  - `colSds()`, `colVars()`, `colMedians()` (`matrixStats` package)
  - `ifelse()`
  - `apply()` family

Activity

- **Vectorized Functions Activity** instructions available on web
- Work in small groups
- Ask questions! TAs and I will float about the room
- Feel free to ask questions about anything you didn't understand as well!

What do we want to be able to do?

- Restructure Data/Clean Data
- Streamline repeated sections of code
- Improve efficiency of code
- **Write custom functions to simplify code**

Writing Functions

- Knowing how to write **functions** vital to custom analyses!
- Function writing syntax

```
nameOfFunction <- function(input1, input2, ...) {
  #code
  #return something with return()
  #or returns last value
}
```

Writing Functions

- Can look at code for functions

```
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
```

41/84

Writing Functions

- Can look at code for functions

```
colMeans
```

```
## function (x, na.rm = FALSE, dims = 1L)
## {
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     if (!is.array(x) || length(dn <- dim(x)) < 2L)
##         stop("'x' must be an array of at least two dimensions")
##     if (dims < 1L || dims > length(dn) - 1L)
##         stop("invalid 'dims'")
##     n <- prod(dn[id <- seq_len(dims)])
##     dn <- dn[-id]
##     z <- if (is.complex(x))
##         .Internal(colMeans(Re(x), n, prod(dn), na.rm)) + (0+1i) *
##             .Internal(colMeans(Im(x), n, prod(dn), na.rm))
##     else .Internal(colMeans(x, n, prod(dn), na.rm))
##     if (length(dn) > 1L) {
##         dim(z) <- dn
```

Writing Functions

- Can look at code for functions

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000018519a90>
## <environment: namespace:base>
```

Writing Functions

- Can look at code for functions

```
mean.default
```

```
## function (x, trim = 0, na.rm = FALSE, ...)
## {
##     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
##         warning("argument is not numeric or logical: returning NA")
##         return(NA_real_)
##     }
##     if (na.rm)
##         x <- x[!is.na(x)]
##     if (!is.numeric(trim) || length(trim) != 1L)
##         stop("'trim' must be numeric of length one")
##     n <- length(x)
##     if (trim > 0 && n) {
##         if (is.complex(x))
##             stop("trimmed means are not defined for complex data")
##         if (anyNA(x))
##             return(NA_real_)
```

Writing Functions

- Goal: Create a `standardize()` function
- Take vector of values
  - subtract mean
  - divide by standard deviation
- z-score idea
- Formula: For value i,

  (value[i] - mean(value)) / sd(value)

45/84

Writing Functions

```
nameOfFunction <- function(input1, input2, ...) {
  #code
  #return something with return()
  #or returns last value
}
```

```
standardize <- function(vector) {
  return((vector - mean(vector)) / sd(vector))
}
```

Writing Functions

- Now use it!

```
data <- runif(5)
```

```
data
```

```
## [1] 0.40420982 0.16632350 0.82994597 0.27298712 0.02134729
```

```
result <- standardize(data)
```

```
result
```

```
## [1]  0.2115897 -0.5598517  1.5922093 -0.2139523 -1.0299950
```

Writing Functions

- Check result has mean 0 and sd 1

```
mean(result)
```

```
## [1] -7.769393e-17
```

```
sd(result)
```

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

Writing Functions

- Goal: Add more inputs
- Make centering optional
- Make scaling optional

```
standardize <- function(vector, center, scale) {
  if (center == TRUE) {
    vector <- vector - mean(vector)
  }
  if (scale == TRUE) {
    vector <- vector / sd(vector)
  }
  return(vector)
}
```

Writing Functions

```
result <- standardize(data, center = TRUE, scale = TRUE)
result
```

```
## [1]  0.2115897 -0.5598517  1.5922093 -0.2139523 -1.0299950
```

```
result <- standardize(data, center = FALSE, scale = TRUE)
result
```

```
## [1] 1.31081187 0.53937041 2.69143144 0.88526981 0.06922711
```

Writing Functions

- Give center and scale default arguments

```
standardize <- function(vector, center = TRUE, scale = TRUE) {
  #center and scale if appropriate
  if (center == TRUE) {
    vector <- vector - mean(vector)
  }
  if (scale == TRUE) {
    vector <- vector / sd(vector)
  }
  return(vector)
}
```

Writing Functions

```
result <- standardize(data, center = TRUE, scale = TRUE)
result
```

```
## [1]  0.2115897 -0.5598517  1.5922093 -0.2139523 -1.0299950
```

```
#same call
result <- standardize(data)
result
```

```
## [1]  0.2115897 -0.5598517  1.5922093 -0.2139523 -1.0299950
```

Writing Functions

- Return more than 1 object by returning a list
- Goal: Also return
  - `mean()` of original data
  - `sd()` of original data

Writing Functions

```r
standardize <- function(vector, center = TRUE, scale = TRUE) {
  #get attributes to return
  mean <- mean(vector)
  stdev <- sd(vector)
  #center and scale if appropriate
  if (center == TRUE) {
    vector <- vector - mean
  }
  if (scale == TRUE) {
    vector <- vector / stdev
  }
  #return a list of objects
  return(list(vector, mean, stdev))
}
```

Writing Function

```
result <- standardize(data)
result
```

```
## [[1]]
## [1]  0.2115897 -0.5598517  1.5922093 -0.2139523 -1.0299950
##
## [[2]]
## [1] 0.3389627
##
## [[3]]
## [1] 0.308366
```

```
result[[2]]
```

```
## [1] 0.3389627
```

Writing Functions

- Fancy up what we return by giving names

```
standardize <- function(vector, center = TRUE, scale = TRUE) {
  #get attributes to return
  mean <- mean(vector)
  stdev <- sd(vector)
  #center and scale if appropriate
  if (center == TRUE) {
    vector <- vector - mean
  }
  if (scale == TRUE) {
    vector <- vector / stdev
  }
  #return a list of objects
  return(list(result = vector, mean = mean, sd = stdev))
}
```

Writing Functions

```
result <- standardize(data, center = TRUE, scale = TRUE)
result
```

```
## $result
## [1]  0.2115897 -0.5598517  1.5922093 -0.2139523 -1.0299950
##
## $mean
## [1] 0.3389627
##
## $sd
## [1] 0.308366
```

```
result$sd
```

```
## [1] 0.308366
```

Writing Functions

- Can bring in unnamed arguments
- Arguments that can be used by functions **inside** your function
- Done already in `apply()`

```
apply
```

```
## function (X, MARGIN, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     dl <- length(dim(X))
##     if (!dl)
##         stop("dim(X) must have a positive length")
##     if (is.object(X))
##         X <- if (dl == 2L)
##             as.matrix(X)
##         else as.array(X)
##     d <- dim(X)
##     dn <- dimnames(X)
##     ds <- seq_len(dl)
```

Writing Functions

```
apply(X = select(airquality, Ozone:Temp), MARGIN = 2,
      FUN = summary, na.rm = TRUE)
```

```
## $Ozone
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.00   18.00   31.50   42.13   63.25  168.00      37
##
## $Solar.R
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     7.0   115.8   205.0   185.9   258.8   334.0       7
##
## $Wind
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.700   7.400   9.700   9.958  11.500  20.700
##
## $Temp
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   56.00   72.00   79.00   77.88   85.00   97.00
```

Writing Functions

- Add unnamed arguments to our function

```
standardize <- function(vector, center = TRUE, scale = TRUE, ...) {
  #get attributes to return
  mean <- mean(vector, ...)
  stdev <- sd(vector, ...)
  #center and scale if appropriate
  if (center == TRUE) {
    vector <- vector - mean
  }
  if (scale == TRUE) {
    vector <- vector / stdev
  }
  #return a list of objects
  return(list(result = vector, mean = mean, sd = stdev))
}
```

Writing Functions

```
sData <- standardize(airquality$Ozone, na.rm = TRUE)
sData$mean
```

```
## [1] 42.12931
```

```
sData$sd
```

```
## [1] 32.98788
```

```
sData$result
```

```
##   [1] -0.03423409 -0.18580489 -0.91334473 -0.73145977          NA
##   [6] -0.42831817 -0.57988897 -0.70114561 -1.03460136          NA
##  [11] -1.06491552 -0.79208809 -0.94365889 -0.85271641 -0.73145977
##  [16] -0.85271641 -0.24643321 -1.09522968 -0.36768985 -0.94365889
##  [21] -1.24680048 -0.94365889 -1.15585800 -0.30706153          NA
##  [26]          NA          NA -0.57988897  0.08702254  2.20901373
##  [31] -0.15549073          NA          NA          NA          NA
```

Recap!

- Function writing opens R up!
- Syntax

```
nameOfFunction <- function(input1, input2, ...) {
  #code
  #return something with return()
  #or returns last value
}
```

- Can set defaults in function definition
- Can return a named list
- Can give unnamed arguments for use

Activity

- **Function Writing Activity** instructions available on web
- Work in small groups
- Ask questions! TAs and I will float about the room
- Feel free to ask questions about anything you didn't understand as well!

What do we want to be able to do?

- Restructure Data/Clean Data
- Streamline repeated sections of code
- **Improve efficiency of code**
- Write custom functions to simplify code

Parallel Computing

- Just basic intro and idea (very complicated, many things to consider!)

Idea:

- Take computations that can be done independently (or close to it)
- Don't run sequentially
- Split up computation
- Run computation simultaneously on
  - different processor cores
  - across many connected computers (i.e. on a cluster)
  - or a few other ways
- Combine results

Parallel Computing

Many applications in data science lend themselves to parallel computing

Examples

- Monte Carlo simulation studies
- Bootstrapping
- Multiple MCMC runs from different starting points
- Cross Validation
- Random Forests and Boosting algorithms

- We'll use `parallel` package (built-in)

Parallel Computing

- `parallel` package function we'll use has syntax similar to `apply()` family
- Problem to parallelize:
    - kmeans clustering
        - group similar observations
    - consider iris data set

```
iris<-tbl_df(iris)
```

Parallel Computing

```
iris
```

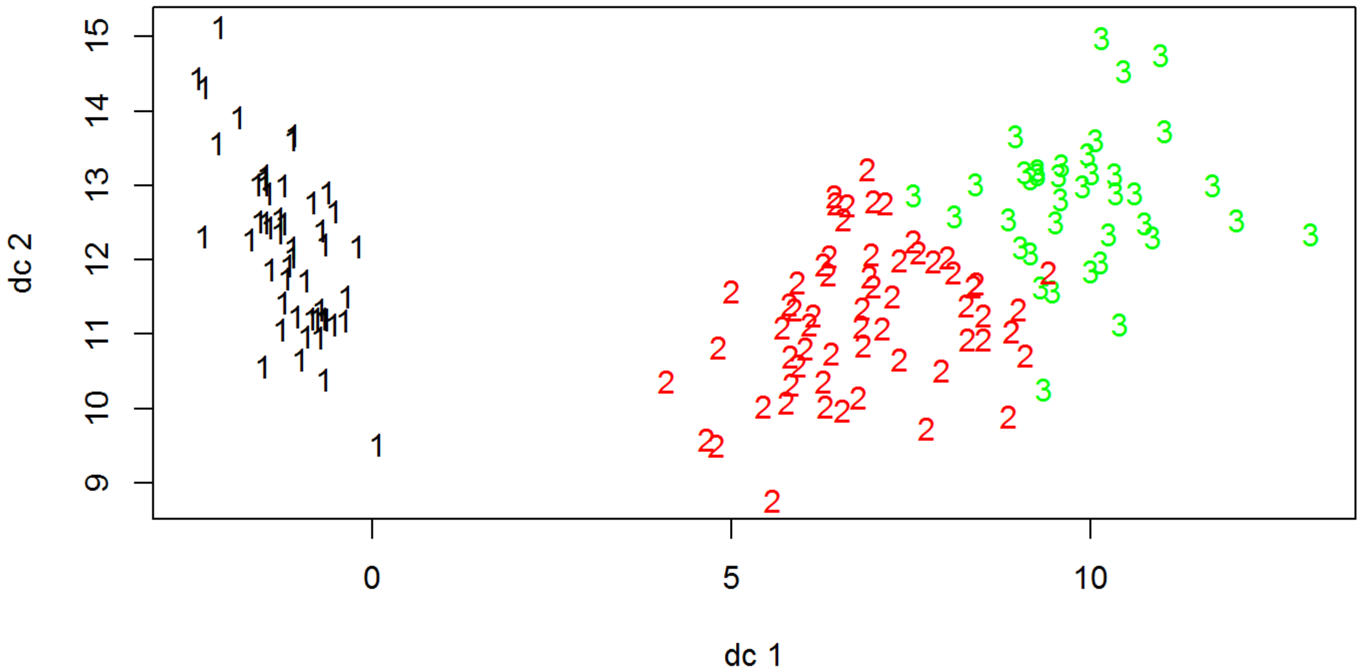
```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## # ... with 145 more rows
```

Parallel Computing

- Problem to parallelize:
  - kmeans clustering
    - group similar observations

```
library(fpc)       #install if needed
iris$Species <- NULL #remove category labels (truly 3 groups)
clus <- kmeans(iris, centers = 3, nstart = 100)
plotcluster(iris, clus$cluster)
```

Parallel Computing



Parallel Computing

Why is this able to be parallelized?

- Code tries to find 3 'cluster' centers using 100 random starting positions
- Result is the starting position that yields the minimal `result$tot.withinss` value
- Each random starting point used is independent of the others (embarrassingly parallel)
- Can assign some of the runs of the algorithm to separate computer cores
- Combine back at the end, look at overall smallest value

Parallel Computing

- How to parallelize this?
- Create a function using lapply to do the kmeans call

```
parallel.function <- function(data, i) {
  kmeans(as.matrix(data), centers = 3, nstart = i)
}
```

Parallel Computing

- Evaluating example with lapply

```
results <- lapply(X = c(25, 25), FUN = parallel.function, data = iris)
```

Parallel Computing

```
results[[1]]
```

```
## K-means clustering with 3 clusters of sizes 38, 50, 62
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     6.850000    3.073684     5.742105    2.071053
## 2     5.006000    3.428000     1.462000    0.246000
## 3     5.901613    2.748387     4.393548    1.433871
##
## Clustering vector:
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [71] 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3 1 1 1
## [106] 1 3 1 1 1 1 1 1 3 3 1 1 1 1 3 1 3 1 3 1 1 3 3 1 1 1 1 1 3 1 1 1 1 3 1
## [141] 1 1 3 1 1 1 3 1 1 3
##
## Within cluster sum of squares by cluster:
## [1] 23.87947 15.15100 39.82097
##  (between_SS / total_SS =  88.4 %)
```

74/84

Parallel Computing

```
results[[2]]
```

```
## K-means clustering with 3 clusters of sizes 38, 50, 62
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     6.850000    3.073684     5.742105    2.071053
## 2     5.006000    3.428000     1.462000    0.246000
## 3     5.901613    2.748387     4.393548    1.433871
##
## Clustering vector:
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [71] 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3 1 1 1
## [106] 1 3 1 1 1 1 1 1 3 3 1 1 1 1 3 1 3 1 3 1 1 3 3 1 1 1 1 1 3 1 1 1 1 3 1
## [141] 1 1 3 1 1 1 3 1 1 3
##
## Within cluster sum of squares by cluster:
## [1] 23.87947 15.15100 39.82097
##  (between_SS / total_SS =  88.4 %)
```

Parallel Computing

- Want best result of the two returned

Parallel Computing

- Want best result of the two returned
- Create a function to determine which call had the best overall result

```
temp.vector <- sapply(results, function(result) {result$tot.withinss})
#take the result for the best one as the final solution
result <- results[[which.min(temp.vector)]]
print(result)
```

```
## K-means clustering with 3 clusters of sizes 38, 50, 62
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     6.850000    3.073684     5.742105    2.071053
## 2     5.006000    3.428000     1.462000    0.246000
## 3     5.901613    2.748387     4.393548    1.433871
##
## Clustering vector:
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

77/84

Parallel Computing

- Set-up is a lot of work!
- For a large problem this could save a lot of time.
- Now parallelize it
  - Set up cores
  - Only code change: use `parLapply()` insead of `lapply()`

Parallel Computing

- Set-up is a lot of work!
- For a large problem this could save a lot of time.
- Now parallelize it
  - Set up cores
  - Only code change: use `parLapply()` insead of `lapply()`

```
library(parallel)
cores <- detectCores()
cluster <- makeCluster(cores - 1)
results <- parLapply(cluster, X = c(250, 250, 250),
                     fun = parallel.function, data = iris)
temp.vector <- sapply(results, function(result) {result$tot.withinss})
result <- results[[which.min(temp.vector)]]
print(result)
```

Parallel Computing

```
## K-means clustering with 3 clusters of sizes 38, 62, 50
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     6.850000    3.073684     5.742105    2.071053
## 2     5.901613    2.748387     4.393548    1.433871
## 3     5.006000    3.428000     1.462000    0.246000
##
## Clustering vector:
##   [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [36] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [71] 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 1 1 1
## [106] 1 2 1 1 1 1 1 1 2 2 1 1 1 1 2 1 2 1 2 1 1 2 2 1 1 1 1 1 2 1 1 1 1 2 1
## [141] 1 1 2 1 1 1 2 1 1 2
##
## Within cluster sum of squares by cluster:
## [1] 23.87947 39.82097 15.15100
##  (between_SS / total_SS =  88.4 %)
##
## Available components:
```

80/84

Parallel Computing

- Compare computation time when parallelized

```
parTime <- microbenchmark({
  library(parallel)
  cores <- detectCores()
  cluster <- makeCluster(cores - 1)
  results <- parLapply(cluster, X = c(25000, 25000, 25000),
                  fun = parallel.function, data = iris)
  temp.vector <- sapply(results, function(result) {result$tot.withinss})
  result <- results[[which.min(temp.vector)]]
}, times = 10, unit = "s")

straightTime <- microbenchmark({
  clus <- kmeans(iris, centers = 3, nstart = 75000)
}, times = 10, unit = "s")
```

Parallel Computing

```
parTime
```

```
## Unit: seconds
##
##  {     library(parallel)     cores <- detectCores()     cluster <- makeCluster(cores - 1)
##       min       lq     mean   median       uq      max neval
##  2.479656 2.499244 2.513047 2.511559 2.529938 2.544348    10
```

```
straightTime
```

```
## Unit: seconds
##                                                          expr      min
##  {     clus <- kmeans(iris, centers = 3, nstart = 75000) } 3.489261
##        lq     mean   median       uq      max neval
##  3.502473 3.560645 3.562536 3.591032 3.647707    10
```

Recap!

- Parallel Computing can speed up computations
- A lot of up front work
- Can only use when process can be done separately
- Many other ways to speed up R as well (see Microsoft R)

What do we want to be able to do?

- Restructure Data/Clean Data
- Streamline repeated sections of code
- Improve efficiency of code
- Write custom functions to simplify code
- Thanks for coming!