Learning *R*

Carl James Schwarz

StatMathComp Consulting by Schwarz
cschwarz.stat.sfu.ca @ gmail.com

Split-Apply-Combine Paradigm

Advanced usage of the *plyr* package

Table of Contents I

Split - Apply - Combine

Performing the same analysis to multiple chunks of your data

Advanced usage of *plyr* package.

Recall

```
res <- plyr::ddply(cereals, "Shelf", plyr::summarize,
        mean.fat=mean(fat, na.rm=TRUE)
        ...
        )
```

Split - Apply - Combine - *ddply()* + your own function

Passing a function to *ddply()* function

- Sometimes computations are too complex to use *plyr::summarize*
- The chunk is passed as DATA.FRAME and traditionally called *x* and is the first argument of the function
- The function can be defined separately or as part of the call (a.k.a. anonymous function)

Split - Apply - Combine - *ddply()* + your own function

Here is a simple function - what does it do?

```
mysummary <- function(x){
   # compute the mean fat and calories and their ratio
   mean.fat = mean(x$fat, na.rm=TRUE)
   mean.calories=mean(x$calories, na.rm=TRUE)
   ratio = mean.calories / mean.fat
   data.frame(mean.fat, mean.calories, ratio, stringsAsFacto
}
```

We test the function:

```
> mysummary(cereal)
  mean.fat mean.calories    ratio
1 1.012987      105.0649 103.7179
```

As always, functions should be *self-contained* and seldom refer to variable not passed as arguments or in the calling environment!

Split - Apply - Combine - *ddply()* + your own function

We pass the function to *plyr::ddply()*

```
report <- plyr::ddply(cereal, "shelf", mysummary)
report
```

We get a separate summary for each shelf

```
> report <- plyr::ddply(cereal, "shelf", mysummary)
> report
  shelf mean.fat mean.calories     ratio
1     1     0.60      100.5000 167.50000
2     2     1.00      107.6190 107.61905
3     3     1.25      106.1111  84.88889
```

As always, functions should be *self-contained* and seldom refer to variable not passed as arguments or in the calling environment!

Split - Apply - Combine - *ddply()* + your own function

Rather than cluttering up the environment with functions that are only used once, it is quite common to use
it anonymous functions. Simply replace the function name above by the actual function definition:

```
plyr::ddply(dataframe, byvars, function(x)
         { # don't forget the opening brace
         func definition
         res1 <- ..
         res2 <- ...
         res <- data.frame(res1,res2,
                  stringsAsFactors=FALSE)
         return(res)
         } # don't forget the closing brace
         ) # don't forget the closing )
```

Split - Apply - Combine - *ddply()* + your own function

Rather than cluttering up the environment with functions that are only used once, it is quite common to use
it anonymous functions. Simply replace the function name above by the actual function definition:

```
report <- plyr::ddply(cereal, "shelf", function(x){
   # compute the mean fat and calories and their ratio
   mean.fat = mean(x$fat, na.rm=TRUE)
   mean.calories=mean(x$calories, na.rm=TRUE)
   ratio = mean.calories / mean.fat
   data.frame(mean.fat, mean.calories, ratio, stringsAsFact
})
```

We get the same results. Be careful to match up braces and parentheses and don't forget to refer to the chunk as *x*.

Split - Apply - Combine - *ddply()* + your own function - Exercise

Fit a separate regression line for each shelf and report the

- intercept
- slope
- RMSE available from *summary(fit)$sigma*

Use an external function definition and an anonymous function

Split - Apply - Combine - *ddply()* + your own function - Exercise

```
library(plyr)
sumstats <- plyr::ddply(cereal, "shelf", function(x) {
        result <- lm(calories ~ fat, data=x)   # notice use
        intercept <- coef(result)[1]
        slope    <-   coef(result)[2]
        sigma   <-   summary(result)$sigma
        res <- data.frame(intercept, slope, rmse,
                    stringsAsFactors=FALSE)
        return(res)
      })
sumstats
```

```
> sumstats
  shelf intercept      slope     rmse
1     1 100.27778  0.3703704 11.76983
2     2  96.78571 10.8333333  9.09777
3     3  91.36752 11.7948718 25.82378
```

Split - Apply - Combine - Exercise

Refer back to the accidents dataset. For each day. compute

- Number of accidents
- Proportion of fatalities
- MEAN weather severity (*Weather_ Conditions*). Not really valid but a close approximation)
- Day of the week (0=Sunday)

Use *plyr::summarize* and write your own (anonymous) function.

Plot number of accident over the year with the SIZE of point related to mean weather conditions.
Add loess curve.

Split - Apply - Combine - Exercise

```
accidents <- read.csv(file.path(... , 'road-accidents-2010.c
                  as.is=TRUE, strip.white=TRUE)
# Convert date to internal date format
accidents$mydate <- as.Date(accidents$Date,
                            format="%d/%m/%Y")
# Create the fatality variable
accidents$Fatality <- accidents$Accident_Severity == 1
```

Split - Apply - Combine - Exercise VI

Using *ddply()* and *summarize()*

```
naccidents <- plyr::ddply(accidents, "mydate", plyr::summar
                   freq=length(mydate),
                   pfatal=mean(Fatality),
                   mean.weather=mean(Weather_Conditions),
                   dow=format(mydate, "%w")[1])
naccidents[1:5,]
```

```
> naccidents[1:5,]
      mydate freq      pfatal mean.weather dow
1 2010-01-01  282 0.014184397     2.262411   5
2 2010-01-02  293 0.030716724     2.740614   6
3 2010-01-03  273 0.014652015     2.857143   0
4 2010-01-04  401 0.002493766     2.518703   1
5 2010-01-05  379 0.002638522     2.936675   2
```

Split - Apply - Combine - Exercise VI

Using *ddply()* and an explicit function

```
naccidents <- plyr::ddply(accidents, "mydate", function(x){
               freq <- nrow(x)
               mean.weather <- mean(x$Weather_Conditions)
               pfatal <- mean(x$Fatality)
               dow=format(x$mydate, "%w")[1]
               res <- data.frame(freq, mean.weather, pfatal,
                         dow, stringsAsFactors=FALSE)
               return(res)
               })
naccidents[1:5,]
```

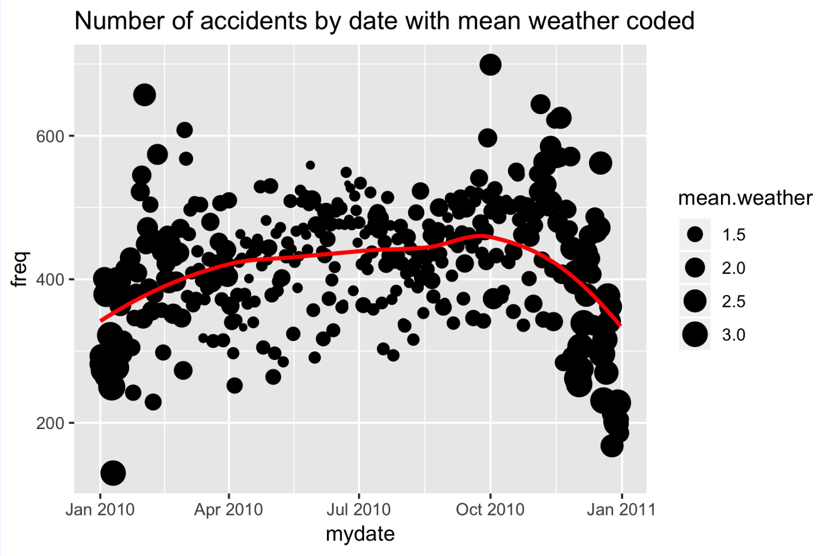
```
> naccidents[1:5,]
      mydate freq mean.weather      pfatal dow
1 2010-01-01  282     2.262411 0.014184397   5
2 2010-01-02  293     2.740614 0.030716724   6
3 2010-01-03  273     2.857143 0.014652015   0
4 2010-01-04  401     2.518703 0.002493766   1
5 2010-01-05  379     2.936675 0.002638522   2
```

Split - Apply - Combine - Exercise VI

Make the plots

```
newplot <- ggplot(data=naccidents,
                  aes(x=mydate, y=freq ))+
  ggtitle("Number of accidents by date with mean weather co
  geom_point( aes(size=mean.weather))+
  geom_smooth(method="loess", color="red", se=FALSE)
newplot
```

Split - Apply - Combine - Exercise VI



Split - Apply - Combine - Exercise VI

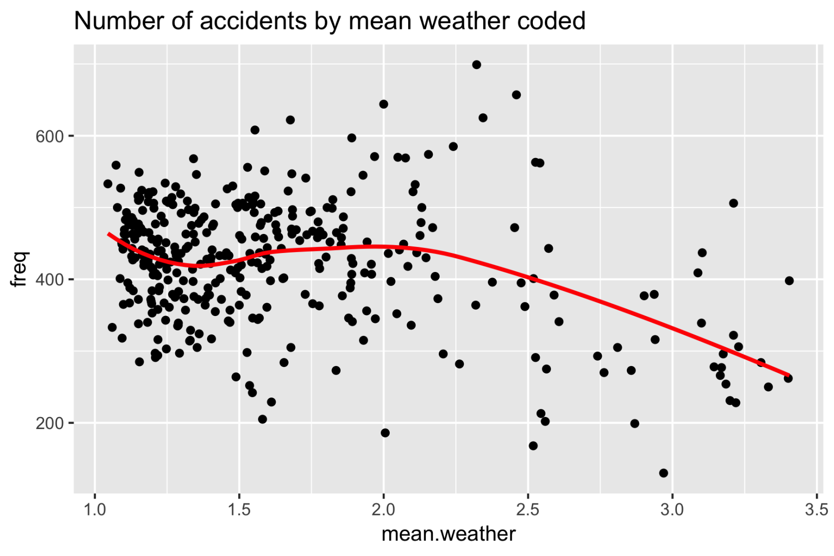
Plot number of accidents vs. mean weather conditions;
Add loess curve

Split - Apply - Combine - Exercise VI

Plot number of accidents vs. mean weather conditions;

```
newplot <- ggplot(data=naccidents,
                  aes(x=mean.weather, y=freq))+
  geom_point( ) +
  geom_smooth(method="loess", color="red", se=FALSE) +
  ggtitle("Number of accidents by mean weather coded")
newplot
```

Split - Apply - Combine - Exercise VI



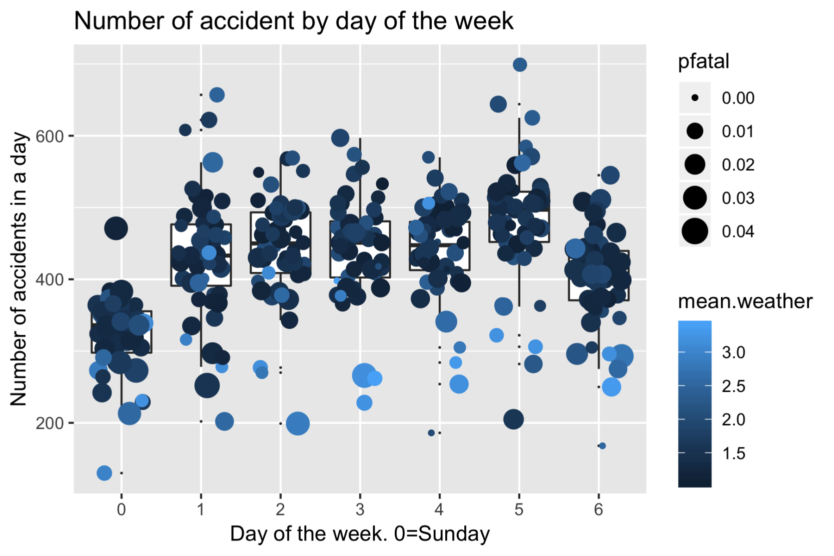
Split - Apply - Combine - Exercise VII

Accident data.

Make a box-plot of number of accident by day of the week coded using proportion of fatalities by the size of the symbol and the mean weather condition by a color gradient.

```
newplot <- ggplot(data=naccidents, aes(x=dow, y=freq))+
  geom_boxplot( ) +
  geom_jitter(aes(size=pfatal, color=mean.weather),
              position=position_jitter(w=.3, h=.0))+
  ggtitle("Number of accident by day of the week")+
  xlab("Day of the week. 0=Sunday") +
  ylab("Number of accidents in a day")
newplot
```

Split - Apply - Combine - Exercise VII



Split - Apply - Combine - Summary (Simple)

VERY COMMON PARADIGM IN *R*.

- Virtually unnecessary to use *for* loops in *R* if computations for each chunk are independent and do not depend on other chunks.
- Makes it easy to parallelize your work (routines are set up to use multiple machines)
- Most common usage is *ddply()*

Split - Apply - Combine - Summary (Simple)

Most simple usage is with *ddply()* and *summarize()*,
Sometimes it is more convenient write your own function

```
sumstat <- plyr::ddply( dataframe, "chunking variable",
                        function(x){
                            res1 <- function of x$...
                            res2 <- function of x$...
                            res <- data.frame(res1, res2, ...,
                                      stringsAsFactors=FALSE)
                            return(res)
                       })
```

Passing additional arguments to *ddply()* functions.
Sometimes you need to pass additional variables other than the chunk to be processed.

Example:Refer back to the cereal dataset. For each shelf group (and for an "arbitrary" variable), compute

- Number of observations
- Number of observations with missing values
- Mean of the variable
- SD of the variable

```r
sumstat <- function(x, var){
  # Compute some summary statistics for a data frame
  values <- x[,var]  # extract the variable values
  n <- length(values)
  nmiss <- sum(is.na(values))
  mean  <- mean(values, na.rm=TRUE)
  sd    <- sd(values, na.rm=TRUE)
  res  <- data.frame(n,nmiss,mean,sd,
              stringsAsFactors=FALSE)
  return(res)
} # end of sumstat


sumstat(cereal, "calories")
sumstat(cereal, "weight")
```

Split - Apply - Combine - Advanced

But how are the second (and additional arguments passed to *ddply()*?

```
res <- plyr::ddply( dataframe, "chunking variable",
                             functionname,
                             y=xxx, z=xxxx)

res <- plyr::ddply( dataframe, "chunking variable",
                    function(x, y, z){
                        res1 <- function of $x$, $y$,and $z
                        res2 <- function of $x$, $y$, and $
                        res <- data.frame(res1, res2, ...,
                                  stringsAsFactors=FALSE)
                        return(res)
                   }, y=xxx, z=xxxx)

plyr::ddply(cereal, "shelf", sumstat, var="calories")
plyr::ddply(cereal, "shelf", sumstat, var="weight")
```

Notice the placement of the variables in the function header and the calling location.

Split - Apply - Combine - Advanced - Exercise

Write a function that takes a data frame and a variable name and

- Find the sample size, number of missing values, mean, its se, and a 95% normal-based confidence interval.
- Bonus - allow for different size of confidence limits, e.g. 90% confidence interval.

Either use the *t.test()* or *lm(y ~1)* or code yourself using sample size and t-distribution

Split - Apply - Combine - Advanced - Exercise

Sample output:

```
> my.simple.summary(cereal, "fat")
  variable  n nmiss     mean       sd        se conflevel
1      fat 77     0 1.012987 1.006473 0.1146982      0.95 0
> my.simple.summary(cereal, "fat", conflevel=.90)
  variable  n nmiss     mean       sd        se conflevel
1      fat 77     0 1.012987 1.006473 0.1146982       0.9 0
> plyr::ddply(cereal, "shelf", my.simple.summary, variable='
  shelf variable  n nmiss     mean         sd         se cor
1     1   weight 20     0 0.991500 0.03801316 0.00850000
2     2   weight 21     0 1.015714 0.07201190 0.01571429
3     3   weight 36     2 1.062353 0.21450519 0.03678734
> plyr::ddply(cereal, "shelf", my.simple.summary, variable='
  shelf variable  n nmiss mean        sd        se confleve
1     1      fat 20     0 0.60 0.7539370 0.1685854       0.9
2     2      fat 21     0 1.00 0.7745967 0.1690309       0.9
3     3      fat 36     0 1.25 1.1801937 0.1966989       0.9
```

Split - Apply - Combine - Advanced

Using *dlply()* and *ldply()*

- It is convenient to do ALL computations for a chunk, return a list, and then extract from the list a needed rather than having several different function.
- Some output (like plots) cannot be stored in data frames.

Example: Refer back to the cereal dataset. For each shelf group (and for two "arbitrary" variable), compute

- Plot of Y vs. X (use *aes_string()* in *ggplot()*
- Regression of Y on Y. Use *lm(x[, Yvar]* $\sim$ *x[, Xvar])*
- Return both in a list

```
sumstat(cereal, "calories", "fat")
```

should return a list with 2 elements.

Split - Apply - Combine - Advanced

Using *dlply()* and *ldply()*

```
sumstat <- function(x, Yvar, Xvar){
   # Do the plot (use aes_string)
   plot <- ggplot(data=x, aes_string(x=Xvar, y=Yvar))+
      ggtitle(paste("Scatterplot of ", Yvar, " vs. ", Xvar,
      geom_point(position=position_jitter(h=.1, w=.1))+
      geom_smooth(method="lm", se=FALSE)
   fit <- lm( x[,Yvar] ~ x[, Xvar], data=x)
   list(plot=plot, fit=fit)
}

res<- sumstat(cereal, "calories", "fat")
length(res)
```

Using *dlply()* and *ldply()*

- Now use *dlply()* to make a list of lists, once for each shelf

```
res <- plyr::dlply(cereal, "shelf", sumstat,
           Yvar="calories", Xvar="fat")
length(res)
res[[1]]
res[[2]]
res[[3]]
```

Using *dlply()* and *ldply()*

- Now use *ldply()*to make a data frame of slope and se

```
se.summary <- plyr::ldply(res, function(x){
    # x is now the list of the flot and the fit
    slope <- summary(x$fit)$coefficients[2,]
    slope
})
se.summary
```

```
  shelf   Estimate Std. Error   t value     Pr(>|t|)
1     1  0.3703704   3.581444 0.1034137 0.9187781230
2     2 10.8333333   2.626300 4.1249412 0.0005759948
3     3 11.7948718   3.698559 3.1890452 0.0030617682
```

Using *dlply()* and *ldply()*

- Now use *ldply()* to extract the plots

```
plyr::l_ply(res, function(x){
    plot(x$plot) # needed because within a function
})
```

- Notice use of *l_ply()* because no output returned.
- Notice use of *plot()* WITHIN function to force display.
- All plots are sent to the plot window in *Rstudio*.
- It is possible to send the plots to a pdf file (see code).

Create residual displays using *dlply()* and *ldply()*

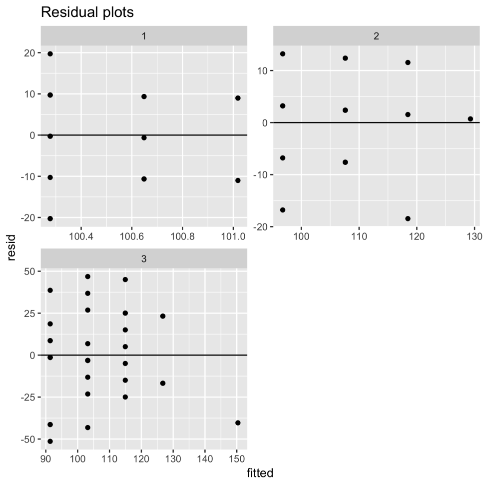
```
resid <- ldply(res, function(x){
    data.frame(resid= resid(x$fit), fitted=fitted(x$fit))
})
head(resid)
```

```
  shelf      resid   fitted
1     1   8.981481 101.0185
2     1 -10.648148 100.6481
3     1   8.981481 101.0185
```

Split - Apply - Combine - Advanced

```
ggplot(data=resid, aes(x=fitted, y=resid))+
  ggtitle("Residual plots")+
  geom_point()+
  geom_hline(yintercept=0)+
  facet_wrap(~shelf, ncol=2, scales="free")
```

Residual plot for each shelf:



Split - Apply - Combine - Advanced - Parallelization

All of the routines in the *plyr* package allow parallelization, i.e. using separate cores on your machine.

- Set up cores to be used
- Set *.parallel=TRUE* in the call
- Close the cores ued.

Split - Apply - Combine - Advanced - Parallelization

Set up the cores

```
doParallel <- TRUE  # should I set up parallel processing o

if(doParallel) {
  library(doMC)  # for parallel model fitting
  # see http://viktoriawagner.weebly.com/blog/five-steps-to
  detectCores()
  cl <- makeCluster(4)
  # Need to export some libraries to the cluster
  # see http://stackoverflow.com/questions/18981932/logging
  clusterEvalQ(cl, library(unmarked))
  registerDoMC(5)
}
```

Run the *plyr::function()* in parallel.

```
Sys.time()
res <- plyr::ddply(cereal, "shelf", plyr::summarize,
                   mean=mean(shelf[1]*(1:100000000), na.rm=T
                   .parallel=FALSE)
Sys.time()

Sys.time()
res <- plyr::ddply(cereal, "shelf", plyr::summarize,
                   mean=mean(shelf[1]*(1:100000000), na.rm=T
                   .parallel=doParallel)
Sys.time()
```

Stop the cores.

```
if(doParallel) stopCluster(cl) # stop parallel processing
```

Split - Apply - Combine - Advanced

Row or Column operations on a MATRIX or ARRAY (less common
Note MATRIX differs from a data.frame because all values must have same type.
ARRAY is a 3+ dimensional object.

```
mat <- matrix(1:30, nrow=6)
mat
```

```
> mat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

Split - Apply - Combine - Advanced

Row or Column operations on a MATRIX or ARRAY.

```
aaply(mat, 1, sum)
aaply(mat, 2, mean)
aaply(mat, 1, function(x){
  res <- prod(sin(x))
  return(res)
})
```

```
> aaply(mat, 1, sum)
 1  2  3  4  5  6
65 70 75 80 85 90
> aaply(mat, 2, mean)
   1    2    3    4    5
 3.5  9.5 15.5 21.5 27.5
> aaply(mat, 1, function(x)....
            1             2             3             4
-0.0046076865  0.6204102446  0.0302615870  0.0002842306 -0.
```

Split - Apply - Combine - Advanced - Exercise VIII

Refer back to the accident dataset.

- For each month, compute the number of days, weekdays and weekends. Hint: Use the *unique()* function on the dates within each month. Why do I want all three values?
- For each month, compute the total number of accidents with injury, those on weekends, and those on weekdays. Again, why do I want all three values?
- For each month, find the ratio of the number of accidents on weekday to weekends.
- Plot these over the year
- Add a suitable comparison line if accidents were uniformly spread over the days of the week. Note that the number of weekends and weekdays differs among months.

Split - Apply - Combine - Advanced - Exercise VIII

```
... read accident data ....
... convert dates to internal R format ...

# get the month for each accident date
accidents$month <- as.numeric(format(
          accidents$mydate, "%m"))
```

Split - Apply - Combine - Advanced - Exercise VIII

```
mysummary <- function(accidents){
# Compute the number of weekend and weekdays in the month (
# Compute the number of accidents on weekend/weekdays
# Report the two ratio.
    DaysOfMonth <- unique(accidents$mydate)
    DaysOfWeeks <- format(DaysOfMonth, "%w")
    nDays      <- length(DaysOfMonth)
    nWeekdays <- sum(DaysOfWeeks %in% 1:5)
    nWeekends <- sum(DaysOfWeeks %in% c(0,6))

    AccDaysOfWeek <- format(accidents$mydate, "%w")
    nAccTotal     <- length(accidents$mydate)
    nAccWeekdays <- sum(AccDaysOfWeek %in% 1:5)
    nAccWeekends <- sum(AccDaysOfWeek %in% c(0,6))

    rAccWdWe <- nAccWeekdays/nAccWeekends
    rDaysWdWe <- nWeekdays/nWeekends

    res <- data.frame(nDays,
             nWeekdays,
```

Split - Apply - Combine - Advanced - Exercise VIII

```
testdata <- subset(accidents, accidents$month == 1)
dim(testdata)

mysummary(testdata)

> mysummary(testdata)
       nDays     nWeekdays     nWeekends     nAccTotal nAccWeek
    31.000000     21.000000     10.000000 10637.000000   7643.00
```

Split - Apply - Combine - Advanced - Exercise VIII

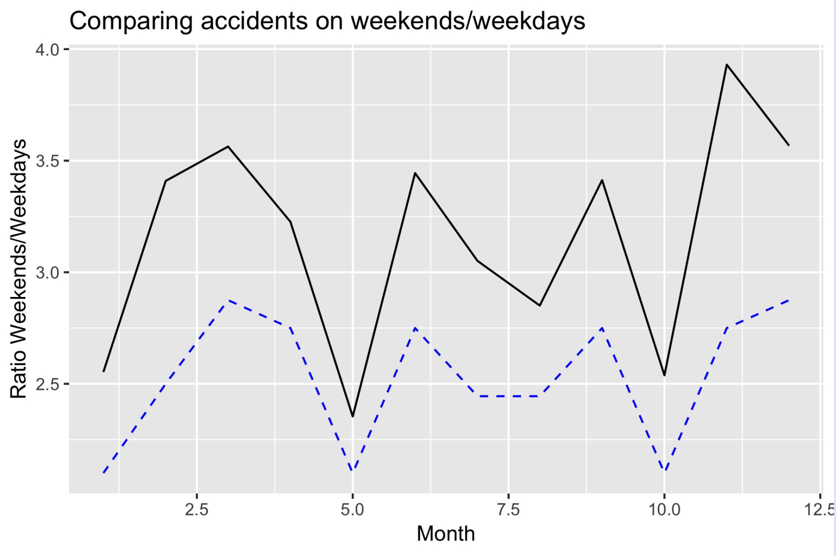
```
results <- plyr::ddply(accidents, "month", mysummary)
results

     month nDays nWeekdays nWeekends nAccTotal nAccWeekdays nA
1        1    31        21        10     10637         7643
2        2    28        20         8     11724         9065
3        3    31        23         8     13165        10280
4        4    30        22         8     12248         9350
5        5    31        21        10     13220         9278
6        6    30        22         8     13644        10574
7        7    31        22         9     13527        10188
8        8    31        22         9     13027         9644
9        9    30        22         8     13904        10753
10      10    31        21        10     14429        10351
11      11    30        22         8     14544        11594
12      12    31        23         8     10345         8080
```

Split - Apply - Combine - Advanced - Exercise VIII

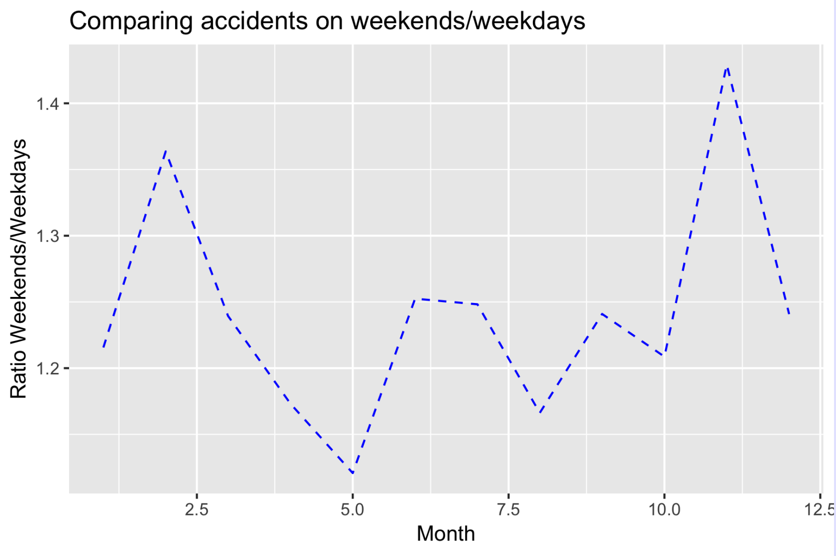
```
newplot <- ggplot(data=results, aes(x=month, y=rDaysWdWe))+
  ggtitle("Comparing accidens on weekends/weekdays")+
  xlab("Month")+ylab("Ratio Weekends/Weekdays")+
  geom_line(group=1, color="blue", linetype=2)+
  geom_line(aes(y=rAccWdWe,group=1))
newplot
```

Split - Apply - Combine - Advanced - Exercise VIII



```
newplot <- ggplot(data=results, aes(x=month, y=rAccWdWe/rDay
  ggtitle("Comparing accidens on weekends/weekdays")+
  xlab("Month")+ylab("Ratio Weekends/Weekdays")+
  geom_line(group=1, color="blue", linetype=2)
newplot
```

Split - Apply - Combine - Advanced - Exercise VIII



General steps for the Split-Apply-Combine paradigm.

- What form is data in? Array, Dataframe, List?
- What form should results be in? Array, Dataframe, List, NULL (for plots)
- Create function to do the application; test with a subset of the data;

```
myfunction <- function() {} # define your function
testdata <- subset( mydata,
                    mydata$variable == testvalue)
myfunction(testdata)
```

- Run the member of the *plyr* package.