# Stat-340/341/342 – Assignment 7
# 2015 Spring Term

## Part 1 - Breakfast cereals - Intermediate

In this part of the assignment, you will learn how to:

- create a factor variable; specifying the order of the levels of the factor;
- dealing with missing values with standard summary functions;
- learn how install a package;
- learn to use the *recode()* function in the *car* package.
- use the *plyr* package and the split-apply-combine paradigm;
- create dot, box, and notched-box plots using the *ggplot()* function;
- improve plots through jittering;
- what is the difference between a factor and a variable and the consequence of mixing these up;
- do a single factor CRD ANOVA using the *lm()* function;
- extract estimated marginal means (LSMEANS) from *lm()* using the *lsmeans* package and make a suitable plot.

We will start again with the cereal data.

1. Read the information about the dataset and breakfast cereals from
   `http://www.stat.sfu.ca/~cschwarz/Stat-340/Assignments/Cereal/Cereal-description.pdf`

2. Download the cereal dataset from
   `http://www.stat.sfu.ca/~cschwarz/Stat-340/Assignments/Cereal/cereal.csv`
   and save it to your computer in an appropriate directory.

3. Input the data into *R* as in Assignment 6.

4. Deal with the coding for missing values as in Assignment 6.

5. Manufacturers pay lots of money to supermarkets to place their product to increase sales. For example, product placement charges are higher for shelves at eye-level rather than on the bottom level. The variable *shelf* indicates the shelf (from the floor) where the product was placed.

   In the rest of this question, we will examine if there is evidence that the the average amount of sugar per serving varies by shelf height?

6. I am always VERY leery of coding qualitative variables (such as *shelf*) using numeric codes. (You will see later why).

   Create a new variable in the data frame, *shelfC* taking the values of *low*, *medium*, *high* depending on the value of shelf (1, 2, or 3). You can do this by indexing the list of character values by the value of shelf using something like

   ```
   c("low","medium","high")[shelf]
   ```

   If you run this code directly, it will fail because *shelf* is not in the workspace (do a *ls()* to verify this). So where is the *shelf* variable? Modify the above code as needed and add the *shelfC* variable to the *cereals* data frame.

   The *recode()* in the *car* package is also very useful. Install and load the *car* package. Try

   ```
   cereals$shelfC <- recode(cereals$shelf,
                      " 1='low'; 2='medium'; 3='high' ")
   xtabs(~shelf+shelfC, data=cereals)
   ```

   Don't get the single and double quotes mismatched! What is the purpose of the *xtabs()* function call?

   Use the *head()* and *tail()* to inspect the first few and last few records, and do a *str()* to verify that you've added the new variable correctly to the data frame.

7. Notice that the data type of the variable *shelfC* is character. One of the most confusing aspects of *R* is that most modeling functions need to create variables of type *factor* before using them in models as classification effects. This was easily done in *SAS* using the *class* statement in the procedures, but not so easily done in *R*.

The *as.is=TRUE* option in the *read.csv()* function PREVENTED the automatic conversion (again, I'm always leery of automatic conversions when you are not an Rexpert because then you are never sure exactly what is going on).

The *as.factor()* and *factor()* functions converts various data types (numeric, characters, etc.) to a **factor** data type. So you would use something like

```
factor(shelfC)
```

to convert the variable *shelfC* to a new variable of type *factor*. Again the code will not run directly for the same reasons as noted just above. Modify this code to create a NEW variable *shelfCF* in the cereal data frame as the factor equivalent of *shelfC*. (I usually add the character *F* to the end of a variable name to remind myself that it is a factor). Again, print out the first 10 records and do a *str()* on the data frame to verity your results.

Notice that when you print the first few records of the data frame that the output of *shelfC* and *shelfCF* LOOK the same, but the *str()* function shows that they are different. Do you see this in the output from the *str()* function? (This would be a good exam question).

8. A factor data type has two components. First a list of the unique levels within the factor. In this case, the *str()* function shows that that the unique levels are:

```
> str(cereals)
'data.frame': 77 obs. of  17 variables:
.......
 $ shelfC  : chr  "high" "high" "high" "high" ...
 $ shelfCF : Factor w/ 3 levels "high","low","medium": 1 1 1 1 1 2 3 1 2 1 ...
```

*high*, *low*, and *medium*. Notice that factors are (usually) sorted (in this case alphabetically).

The second component is an INDEX into the levels of the factor. So the in the first row, the index of the first observation is 1, which points to the *"high"* level of the factor. When you print a factor variable, the level corresponding to the index is printed which is why the values for the two new shelf variables LOOK the same, but internally are quite different.

BE SURE YOU UNDERSTAND HOW A FACTOR DIFFERS FROM A REGULAR VARIABLE. Otherwise you will be confused for the rest of this assignment!

9. In this case, the default action when a factor is created is to sort alphabetically. In this case, this doesn't make any sense as the first level of the factor corresponds to the *high* shelf.

*R* has facilities to reorder the levels of a factor to make plots and other output more meaningful. You should always specify the levels of the factor in meaningful way if needed.

The *factor()* function allows you specify the ORDER of level labels when you create them using something like:

```
factor(shelfC, levels=c("low","medium","high"), ordered=TRUE)
```

Be sure that your ordering uses the values actually in the variable *shelfC*. Create a new variable in your data frame called *shelfCFO* that is a factor with the shelf height ordered as above. (I usually add the characters *FO* to the end of a variable to remind myself that the variable is an ordered factor.) Print the first 10 records; do a *str()* on your data frame, and examine CAREFULLY the structure of the *shelf*, *shelfC*, *shelfCF*, and *shelfCFO* variables in your data frame.

BE SURE YOU UNDERSTAND WHAT IS GOING ON and how the variables *shelf*, *shelfC*, *shelfCF* and *shelfCFO* differ while all representing the same thing, i.e. which shelf the cereal was located. You need to be careful in how you use these four variables in subsequent plotting and modeling steps – this is a good exam question.

10. Lets make some side-by-side dot and box plots using the *ggplot()* function.. Load the *ggplot2* library using the *library()* function.

    Note it is ALWAYS a good idea to use the Package Updates feature of *Rstudio* before using a new library to see there have been updates. You can click on the *News* button of the Package Updates window to see what has been changed in your packages when you update them.

    Also, it is traditional to put all of the *library()* calls at the start of your script so new users of the script can see which packages are being used.

    It is also helpful to print off the *ggplot2* reference card from `http://people.stat.sfu.ca/~cschwarz/Stat-R/CourseNotes/R-manuals/R-ggplot2_ref_card.pdf`.

11. The *ggplot* methods requires several steps in creating a plot. First, you must define the data frame where ALL variables used in a plot are defined. Second, the *aes()* function (the *aes* stands for *aesthetics*) tells *ggplot* which are the variables that serve as the $X$ and the $Y$ variables and if the plots will be "coded" by groups (similar to the *group=xxx* option in many of the commands in *Proc SGplot* of *SAS*). Third, the *geom_xxxx()* functions tell how to plot the $Y$ and $X$ variables. Fourth, there are various functions to modify the titles, axes labels, axes limits, etc. Look at `http://ggplot2.org` for the home page of this package and at `http://docs.ggplot2.org/current/` for the latest documentation.

    The various layers of the plot (set by the various commands) can be specified in any order (but it is tradition to specify the *ggplot* layer first), and the plot is not created until all layers are specified and a plot is requested. So unlike Base *R* graphics, the limits on the axes are not fixed in stone after the first layer is drawn.

And, unlike Base *R* graphics, *ggplot* allows you to save a plot as an *R* object – Base *R* graphics writes everything out to the plot window and does NOT create an *R* object in the workspace. The ability to create *ggplot* objects is very useful as you can then have several plots in a row that embellish earlier plots. For example, you could draw a scatter plot and create an object in the workspace. Then after fitting the line, you can add to the object to create a new object that includes, for example, the fitted line.

Because you create an object in the workspace, don't forget to actually display the plot – many people wonder why no plot is produced after they run the ggplot commands.

So let us begin by specifying the object to be created and the basic features of the plot.

```
plot.dot.box <- ggplot(data=cereals, aes(x=shelfCFO, y=sugars))+
    ggtitle("Sugars by shelf - using an ordered factor")+
    xlab("Shelf")+ylab("Sugar per serving (g)")
plot.dot.box
str(plot.dot.box)
```

Notice how you create an object *plot.dot.box* in the workspace and how you specify the title etc. If you run this code, you will get an "error" because while you have specified what variables to plot, you haven't specified how to plot them (via the *geom_ xxxxx()* functions. Also notice how the various parts of the graphs are added to the plot using the '+' operator - you are not doing arithmetic, but adding things to the plot.

Finally, notice that the *str()* function show that a complex list object now exists.

12. We now start to add the plotting layers. The *geom_ boxplot()* function adds a box plot. So add this to the the plot to get:

```
plot.dot.box <- ggplot(data=cereals, aes(x=shelfCFO, y=sugars))+
    ggtitle("Sugars by shelf - using an ordered factor")+
    xlab("Shelf")+ylab("Sugar per serving (g)")+
    geom_boxplot()
plot.dot.box
```

This now give the box-plots. Read the help on *geom_ boxplot* to see how to add notches to the box plot and add the notches to the above plot. What do you conclude based on the above plot. Some of the notches look odd because the notch extends past the 25th or 75th percentiles. Do you understand why the plot looks odd and how to interpret the plot? What can you conclude based on the notched side-by-side box plot? Hint: read `http://www.cms.murdoch.edu.au/areas/maths/statsnotes/samplestats/boxplotmore.html`. Again a good exam question.

13. Add the individual points to the plot using the *geom_point()* function. There are over 70 breakfast cereals in the dataset, but only about 30 dots are displayed. Why? Hmm... we need some jittering. Try the *geom_jitter()* rather than *geom_point()*. The plot now has too much jittering. Look at the help file for *geom_jitter* to see how you can modify the amount of horizontal and vertical jittering. Modify the *geom_jitter()* function call so that the jittering is not too extreme.

14. Finally, notice that the box plot is opaque (i.e. the back ground plotting color (the grey screen) does not show through. The *alpha=* argument in the *geom_ boxplot()* function controls the opaqueness with *alpha=1* completely opaque, and *alpha=0* completely transparent. Try various values for *alpha*.

15. When you are happy with the plot, you can save the plot to your directory using the *ggsave()* function where you specify the plot object to save and the file to hold the objects.

    ```
    ggsave(plot=plot.dot.box, file='cereal-dot-plot.png',
       h=4, w=6, units="in", dpi=300)
    ```

    The *ggsave()* function examines the extension to the file name to determine the type of file to create (in this case a *png* graphics file). The other options control the height, width, and resolution of the plot.

16. Repeat the plot but use the *shelf*, *shelfC*, *shelfCF* and *shelfCFO* variables. Do you see how the plots change? Pay particular attention to the plot when the ordinary numerical variable *shelf* is used – BE AFRAID, BE VERY AFRAID of categorical variables coded using numeric values!

17. Install the *gridExtra* package and use the *grobArrange()* to put these 4 plots into one big plot using

    ```
    plot.dot.box.all <- arrangeGrob(plot.dot.box.CFO,
                                 plot.dot.box.CF,
                                 plot.dot.box.C,
                                 plot.dot.box, ncol=2)
    plot.dot.box.all
    ```

18. Let's compute some summary statistics about the amount of calories and amount of sugar/serving. Try

```
mean(calories)
```

This will fail. Do you understand why? This is an important concept in *R* and you need to understand how data.frames encapsulate the interior variables.

Instead try

```
mean(cereals$calories)
```

Do you understand why this now work? These are good exam questions.

The $ syntax to access variables within a data.frame is sometimes clumsy, particularly if the same variable is used repeatedly. The *with()* can provide some relief. Try

```
with(cereals, mean(calories))
with(cereals, mean(sugars))
```

What does the *with()* do and why is it necessary?

You will notice that the result for the mean of the sugar/serving is NA? Print all of the value of the the *sugars* variable and you will see that there are some missing values. Unlike *SAS*, *R* does not automatically drop missing value and so if a computation includes a missing value, the result is also missing.

Look at the *help(mean)*. You will see an argument *na.rm=* which indicates what is supposed to happen if missing values are present in your data. Now try

```
with(cereals, mean(sugars, na.rm=TRUE))
```

*R* is not alway consistent in the treatment of missing values[1]. For example, I want a count of the number of non-missing values in the *sugars* variable. There is no function in *R* that does this directly. For example, try

```
with(cereals, length(sugars))
```

This includes the missing values and there is no *na.rm=* argument in the *length* function. But there is a *na.omit()* function which drops missing values, so a combination of both should work:

```
with(cereals, length(na.omit(sugars)))
```

There is also the *is.na()* function which returns a TRUE/FALSE if the corresponding value of the argument is missing. So something like:

[1] *R* is free but not cheap

```
with(cereals, sum( !is.na(sugars)))
```

also works. What does the ! (exclamation mark) operator do? What does

```
with(cereals, sum( is.na(sugars)))
```

do? This is a common code fragment in *R* scripts.

19. We would like to compute summaries for subsets of the data.frame, e.g. for every shelf. This is the Split-Apply-Combine paradigm that we first used in *SAS* and in Assignment 6. The *plyr* package is the standard package now used rather than the more obscure Base *R* functions.

    There are two common ways to use the *plyr* package, and in particular the *ddply()* function (see several questions below).

    The first is to use the *summarize* function for simple summary statistics. For example, to compute some simple summary statistics by *shelf*, we use

    ```
    shelf.sum <- ddply(cereals, "shelf", summarize,
            mean.calories = mean(calories),
            std.calories   = sd(calories),
            mean.sugars    = mean(sugars, na.rm=TRUE))
    ```

    This is useful when only a simple summary is wanted.

    Use this form of the functions to find the following summary statistics for each shelf:

    - The number of cereals
    - The number of missing values for calories
    - The mean calories/serving
    - The standard deviation of calories/serving
    - The standard error of the mean for calories/serving (recall that for an SRS, $se(\overline{y}) = \frac{s}{\sqrt{n}}$.
    - The number of missing values for sugars
    - The mean sugars/serving ignoring missing values
    - The standard deviation of sugars/serving
    - The standard error of the mean for sugars/serving.

    As you can see, this can become tedious and difficult if the summary is anything but simple. For example, can you modify the above to also compute the upper and lower confidence intervals for the respective means!

20. The second common usage is via a function. For example, create a function that computes for its arguments (a data frame and a variable), the number of elements, the number of non-missing entries, the mean, the standard deviation, and the standard error (assuming a CRD/SRS design), and a 95% (by default) confidence interval for the mean and returns a data.frame for its result. I'll start you off:

```
mysummary <- function(mydf, var, alpha=0.05){
# Compute the number of element, number of non-missing elements
# mean, sd, and se of mydf$var
# mydf is assumed to be a Dataframe and values are collected using a CRD/SRS
  values  <- mydf[, myvar]
  ntotal  <- length(values)
  nonmiss <- length(na.omit(values))
  ...
  return(data.frame(ntotal,nonmiss,....))
}
```

Notice the use of a default value for the *alpha* level that will be use used to find the upper/lower bounds of the confidence interval (see previous assignments).

Why do I specify

```
  values  <- mydf[, myvar]
```

rather than

```
  values  <- mydf[, "myvar"]
```

to extract the values from the data frame?

Don't forget to remove missing values when computing the summary statistics.

Did you use the correct number of observations when computing the standard error and the $t$-multiplier for the 95% confidence interval? The *qt()* function gives the quantiles of a t-distribution that you will need to compute the 95% confidence interval.

Why do I specify *nonmiss=mynonmiss* in the final result (try changing this to see what happens)?

This is what I get when I try my function on *cereals$sugars* using

```
mysummary(cereals, "sugars")
    ntotal    nonmiss       mean         sd         se        lcl        ucl
77.0000000 76.0000000  7.0263158  4.3786564  0.5022663  6.0257499  8.0268816
```

21. We now want a summary table above for cereals on EACH shelf. We will use the *plyr* package.

    Again, start by loading the *plyr* library. Documentation on the *plyr* package is available at `http://plyr.had.co.nz`

22. The *plyr* package has series of functions of the form *xyply()* where *x* and *y* specify the input form and output form, e.g. *x=d* implies a data frame input, *y=l* (an el), specifies a list output etc.

    Read the help file on the *ddply()* function. The calling sequence is

    ```
    ddply(.data, .variables, .fun = NULL, ..., .progress = "none",
      .inform = FALSE, .drop = TRUE, .parallel = FALSE, .paropts = NULL)
    ```

    We will ignore for now all but the first 4 arguments.

    The first argument (*.data*) is data frame to be processed. Don't be taken aback by the odd name of the argument (starting with a period) – this is a valid variable name and the use of the dots in the first position of the variable name is often used so that most programmer don't accidentally use this argument name as a "real" variable.

    The second argument (*.variables*) is the VECTOR of variable NAMES that define the groups in the data frame.

    The third argument (*.fun*) is the function to apply to each each subset of the data frame. Often this is specified as part of the calling sequence (as we did in class), but it can also be a separate function as you will see later.

    The fourth argument (*...*) is a bit strange looking! The (*...*) argument often refers to additional argument that will be passed, not to the calling *ddply()* function, but rather to the called *.fun* function. We put this together to get:

    ```
    summary.shelf <- ddply(cereals, "shelf", mysummary, myvar="sugars")
    summary.shelf
    str(summary.shelf)
    ```

    You should get:

    ```
    > summary.shelf <- ddply(cereals, "shelf", mysummary, myvar="sugars")
    > summary.shelf
      shelf ntotal nonmiss     mean       sd        se      lcl       ucl
    1     1     20      19 5.105263 4.483237 1.0285251 2.944412  7.266114
    2     2     21      21 9.619048 4.128876 0.9009947 7.739606 11.498490
    3     3     36      36 6.527778 3.835817 0.6393028 5.229924  7.825632
    ```

    Do you see how the *...* argument in the *ddply()* function worked? This is important to understand as it is a VERY common way to pass additional information to functions that are called by functions. Do you standard what I mean by "functions called by functions" in the context of this example?

So how would you find a 99% confidence interval without rewriting the function, but rather passing a new value to the *alpha* argument?

What do you notice about the standard deviation of the amount of sugar per serving across the shelf heights? Most linear model procedures assume that the population variances are equal across all treatment groups. Based on these summary statistics, is this a reasonable assumption? Why? At which point would you be concerned? Hint: refer to Section 5.3 in `http://www.stat.sfu.ca/~cschwarz/CourseNotes`.

The sample sizes in the three shelves is not equal – we say that the design is unbalanced. Normally, most computer packages can deal with unbalance rather easily. However, this will have an influence on the estimated standard errors (and confidence intervals) for the mean of each group. What will be this effect?

23. Repeat the *ddply()* call using *shelfC*, *shelfCF*, and *shelfCFO* variables and compare the results table and the structure of the results. Do you see how the tables change depending on how the shelf is coded as a number, as a character value, as a factor, or as an ordered factor.

24. I want to create a plot the mean by shelf with the 95% confidence intervals for each mean.using the *ggplot()* function with the different types of shelf variable.

    The data frame containing the information to plot is the *summary.shelf* object created above. It contains a variable called *shelf* and *mean* that contain the mean sugar/serving for each shelf. Start by plotting these means using *ggplot()* in a similar way to see previously.

    The *geom_ errorbar()* function adds error bars to the plot. Find this on the *ggplot* reference card. When you read the help on this function, notice that it require TWO more arguments(*ymax* and *ymin*) in the aesthetics portion of the calls. You can add these to the *ggplot()* function call, or to the *geom_ errorbar()* call – I prefer the latter because it keeps the limits with the function call to make it clearer, i.e.

    ```
    .... +
    geom_errorbar( aes(ymin=lcl, .... ))
    ```

    The width of the top and bottom horizontal bars are too wide for my taste - there is a *width* argument that controls this. Be careful to put this OUTSIDE the *aes()* call - refer to the reference card.

    Finally, add a line joining the means using the *geom_ line()* layer. However, read `http://stackoverflow.com/questions/15978836/ggplot2-geom-line-for-single-obser` to see why you need to "outsmart" the *ggplot()* by using:

```
   geom_line(aes(group=1))
```

25. Repeat the above plots using the *shelf*, *shelfC*, *shelfCF* and *shelfCFO* variables.

26. We will now do a formal hypothesis test. We again use the *lm()* function to test the hypothesis that the mean amount of sugars/serving is the same across the three shelves. Now we MUST BE EXTREMELY careful that shelf is defined as a factor rather than a continuous variable.

    For example compare the outputs from

    ```
    my.fit.shelf   <- lm(sugars ~ shelf,    data=cereals)
    my.fit.shelfC  <- lm(sugars ~ shelfC,   data=cereals)
    my.fit.shelfCF <- lm(sugars ~ shelfCF,  data=cereals)
    my.fit.shelfCFO<- lm(sugars ~ shelfCFO, data=cereals)

    anova(my.fit.shelf)
    anova(my.fit.shelfC)
    anova(my.fit.shelfCF)
    anova(my.fit.shelfCFO)

    summary(my.fit.shelf)
    summary(my.fit.shelfC)
    summary(my.fit.shelfCF)
    summary(my.fit.shelfCFO)
    ```

    When shelf if treated as simple numeric variable (my.fit.shelf), you get regression (e.g. similar to the fit of calories vs. grams of fat) which is NOT what is wanted here? Do you understand why? THIS IS A COMMON ERROR made by novices in *R* (and *SAS*)!

    So we are now only interested in the last 3 fits. What is the null and alternate hypotheses, the F-value, and the p-value. What do you conclude? Notice that the results from the *anova()* function are identical, as they must be. Why?

    If we look at the results from the *summary()* function, they results are NOT all identical – sigh.[2] *R* does ANOVA by creating indicator variable that represent the change in MEANS from some reference level. In the my.fit.shelfC and my.fit.shelfCF, the reference level is the *high* shelf because the value of *high* is first alphabetically. In the the last fit, the reference level is the *low* shelf. Do you see how to see this from the output of the *summary()* function?

    So the test conducted in the output from the *summary()* function are difficult to interpret. Base *R* does NOT have a simple way to compute

[2] *R* is free but not cheap.

the marginal means that we saw how to do in *SAS*. Fortunately, people have written such supplemental routines in units called packages.

27. Load the *lsmeans()* package. The *lsmeans()* function has similar functionality to the LSMEANS statement in *SAS*. For example, the following code

```
library(lsmeans)
my.fit.shelfCF <- lsmeans::lsmeans(my.fit.shelfCF, ~shelfCF, adjust="tukey")
summary(my.fit.shelfCF.lsmeans)
cld    (my.fit.shelfCF.lsmeans)
pairs  (my.fit.shelfCF.lsmeans)
confint(pairs  (my.fit.shelfCF.lsmeans))
```

    produces a 'table" of the marginal means and 95% confidence intervals for the marginal means, a compact letter display comparing the population means (as you saw in *SAS*), and all of results form the pairwise comparisons along with confidence intervals for the difference in means for each pair.

    Unfortunately, there are several packages that have a function called *lsmeans* so the double colon (::) between the package name (*lsmeans*) and the function within the package *lsmeans()* makes sure that the correct function from the correct package is used.

    Run the above code on the *my.fit.shelfCFO* object and compare the results.

    What do you conclude from each multiple comparison. Notice that the results MUST be identical in the end regardless of which factor coding is used.

    You can't run this code on the my.fit.shelfC object as shelf was stored as a character rather than as a factor.

    Any it makes NO sense to run the *lsmeans()* function on the *my.fit.shelf* object - why?

28. We will now make a plot of the means and 95% confidence intervals based on the *lsmeans()* output in a similar fashion as before – it is only necessary to use *ggplot*. Don't forget to save the output from the lsmeans calls before plotting, i.e.

```
shelfCFO.lsmeans <- summary(my.fit.shelfCFO.lsmeans)
shelfCFO.lsmeans
str(shelfCFO.lsmeans)

plot.shelfCFO.means2 <- ggplot(data=shelfCFO.lsmeans, aes(x=shelfCFO, y=....))+ .....
```

There is slight difference between the two plots of the means and confidence intervals - why are the confidence intervals slightly different between these plots? Hint: compare the df used in finding the t-value for the two plots? Why are these different? What else is different?

29. Unfortunately *R* does NOT produce many of the diagnostic plots that we saw in *SAS*. Furthermore, the *ggplot()* package does not have many of the standard diagnostic plots that we want. Fortunately, there are many examples on the web on how to produce these, and I'm bundled some of these into a some helper functions.

    These functions and diagnostic plots for *lm()* objects can be obtained using

    ```
    source("http://www.stat.sfu.ca/~cschwarz/Stat-650/Notes/MyPrograms/schwarz.functions.r"
    sf.autoplot.lm(my.fit.shelfCFO)
    ```

    This automatically produces diagnostic plots of the fit of the ANOVA model. Examine the diagnostic plots. Is there evidence of a problem? Why does the residual plot have only 3 vertical bands of points?

30. Finally, always create an HTML notebook of your final code. This ensures that it runs without problems (but of course, does not make logic checks).

Hand in the following using the electronic assignment submission system:

- Your *R* code that did the above analysis.
- An HTML file containing all of your *R* output.
- A one page (maximum) double spaced PDF file containing a short write up on this analysis explaining the results of this analysis suitable for a manager who has had one course in statistics. You should include the following:
  - A (very) brief description of the dataset.
  - Why a simple regression of sugars vs. shelf is not an appropriate method.
  - The estimated mean amount of sugar/serving for each shelf along with a 95% confidence interval for the population mean.
  - The plot of the mean amount of sugar/serving by shelf with the 95% confidence intervals. Normally you would not give both the table and the plot, but I want you to get practice in generating both types of output. You can adjust the size of the plot in your word-processor.

  - A conclusion about the difference in mean amount of sugar/shelf across the 3 shelves, i.e. which shelf seems to have the highest mean amount of sugar per serving; which shelves seem to have the same mean amount of sugars/serving?

You will likely find it easiest to do the write up in a word processor and then print the result to a PDF file for submission. Pay careful attention to things like number of decimal places reported and don't just dump computer output into the report without thinking about what you want.

## Part 2 - Road Accidents with Injury - Intermediate

In this assignment, we will examine the circumstances of personal injury road accidents in Great Britain in 2010. The statistics relate only to personal injury accidents on public roads that are reported to the police, and subsequently recorded, Information on damage-only accidents, with no human casualties or accidents on private roads or car parks are not included in this data.

Very few, if any, fatal accidents do not become known to the police although it is known that a considerable proportion of non-fatal injury accidents are not reported to the police.

In this part of the assignment, you will learn how to:

- import dates and times;
- format dates and time for display purposes;
- extract features of dates and times, such as the year, month, day, hour, minute;
- create summaries at a grosser level.

1. Return to the Accidents data file from earlier in the course.
2. Open the *.csv file and have a look at the data values. Notice that the dates are in day/month/year format with varying number of digits for the year (but you can assume that they are all in 2010).

   There are over 150,000 records, so manually scanning the entire data is simply not feasible, nor desirable.
3. Read the data into *R* using the *read.csv()* function with appropriate options. Print out the first few records and look at the structure of the data frame.
4. Notice that the *date* and *time* have been read in as character data. We need to convert them to the internal *R* representation of dates and times – the number of second since 1970-01-01 and midnight respectively.
5. First start with Date variables. This is fairly straightforward – as in *SAS*, these are converted to the number of days since the origin (which is 1970-01-01 in *R* vs. 1960-01-01 in *SAS*).

   The *as.Date()* function does the conversion using code:

```
radf$myDate <- as.Date(radf$Date, "%d/%m/%Y")
sum(is.na(radf$myDate))
radf$myDate[1:10]
```

The % codes are found in the help for the *strptime()* and specify the format of the input date variables. Notice the difference between %y and %Y.

The default display format for Dates in *R* is the ISO YYYY-MM-DD format. There is no easy way to change this as can be done in *SAS*.

6. We will begin by looking at the number of (reported) daily accidents by the day of the year and see if the average number of daily accidents varies by month.

   Use the *plyr* package and the *ddply()* function to count the number of accidents in each date. Hint: use the *summarize* method (see Part I), and the *length()* function to count the number of accidents in each group of data.

   Your final object should have 365 rows and a column for the date and the number of accidents.

   You should print out the first few records of the the daily summary to see the names of the variables. Find the sum of the daily number of accidents and compare it to the total number of rows in the original data frame. What do you expect to see?

7. Plot the number of accidents by date using *ggplot*. What is the general impression given by the plot? When is the number of accidents generally lowest and when are they highest?

   Improve the scatter plot by adding a smoother to the curve using the *geom_ smooth()* function. What does the smoothed curve seem to show?

8. Extract the month from the date value from the daily summary. Your code fragment will look something like:

   ```
   naccidents$month <- as.numeric(format(naccidents$myDate, "%m"))
   ```

   Again, the % codes are found in the *help(strptime)* information. Note that the conversions from Date variable ALWAYS returns a character representation and so you may need to convert it to numeric values.

9. Create a summary of the number of days, the mean accidents/day, and the std deviation of the accidents per day for each month, the se for the mean, and a 95% confidence interval as you did in Part 1 of this assignment. You should be able to use your code from Part 1 with only a few changes.

   Create a plot of the mean number of accidents/day for each month along with 95% confidence limits as you did in Part 1. You can force the $X$ axis to be discrete integers using the *ggplot* layer

```
... +
scale_x_discrete(lim=1:12)
```

Also save the plot of the mean number accidents to an png file for the report. What do you conclude from this plot?

10. Use the *lm()* function to test the hypothesis that the number of accidents/day is the same across all months. This will be done in a similar fashion as in Part 1. Don't forget to make a new variable which is the the month variable as a factor.

    Again check the std. deviations to see if the assumption of equal population standard deviations across the months is tenable.

    Again, use the *lsmeans* pcakge to again extract the estimated mean number of accidents/day from the fitted object and examine the output from the *cld()* method. What do you conclude from this?

11. Dealing time-stamp date (Dates + Times) in *R* is very clumsy. I find the following to be most useful.

    We will deal with dates and times using the POSIXct system of representation. There are a few records with no time (the time field is null), and so we will (arbitrarily) assign these a value of "12:00" before the conversion (notice that the == is against a null string, i.e. no space between the quote marks.):

```
radf$Time[ radf$Time==''] <- "12:00"
radf$DateTime <- as.POSIXct(paste(radf$Date," ",radf$Time),
          format="%d/%m/%Y %H:%M", tz="UTC")
radf$DateTime[1:10]
str(radf$DateTime)
as.numeric(radf$DateTime[1:10])
```

    The codes used in the *format=* options are described in the *strptime()* function help. Again, notice that the internal representation of the the date-time (as shown by the output from the *as.numeric()* function is displayed in a standard format when displayed in *R* output.

    The *tz="UTC"* indicates that the date-time is stored in Universal Coordinated Time, i.e. ignoring daylight savings time, and other time zone problems.

    Once date-times are converted to POSIXct format, then differences in date-times can be found using the *difftime()* function. You can also extract the months, or day of the week etc by using the *ormat()* function with the same codes as used previously. For example

```
radf$month <- as.numeric(format(radf$DateTime, "%m"))
radf[1:10,c("DateTime","month")]
```

extracts the numeric months for each date-time. The *as.numeric()* function is needed because the *format()* function ALWAYS returns a character string.

12. Now we will examine the number of accidents by time of day. Go back to the original data file, and create a derived variable for the hour and for the minute of the accidents. Again, we always want to first create a POSIXct variable and then extract as needed.

    Your code fragment will look something like:

    ```
    radf$hour <- as.numeric(format(radf$DateTime, "%H"))
    radf$min  <- as.numeric(format(radf$DateTime, "%M"))
    ```

13. Print the first few records to verity that you have extracted the hour and minute correctly.

14. Use *ggplot* to make a histogram of times of accidents by hour of the day. You can adjust the bin width of the histogram to 1 hour using the *bindwidth=1* argument in the geometry call. You can adjust the opacity of the histogram using the *alpha* parameter as previously done.

    What does the distribution of accidents by the hour of the day indicate.

15. Repeat the above by the minute of the hour. Again use the *break=* argument to get a separate count for each minute in the hour. When is the most dangerous minute of an hour to be on the roads, as indicated by the histogram. Do you believe this? Why has this happened?

    Save the plot of the distribution of accidents by the minute of the hour to an png file for the report.

Hand in the following using the online submission system:

- Your *R* code.
- An HTML file containing the the output from your *R* program.
- A one page (maximum) double spaced PDF file containing a short write up on this analysis suitable for a manager of traffic operations who has had one course in statistics. You should include:
  - A (very) brief description of the dataset.
  - A plot of the number of accidents as it varies by minute of the hour.
  - Explain the odd shape of the histogram and implications for policing. What is the most surprising aspect of the histogram?

## Part 3 - Practice using plyr

The *plyr* package is one of the MOST commonly used packages in *R* because the SAC paradigm is so common.

In this part of the assignment, you will get some practice in using *plyr()* functions. There is nothing to hand in, just compare your solutions to the ones given here. These type of questions would make good exam questions.

All examples use the Accident database from Part 02. Read in the accident data. Create the *DateTime*, month and hour variable as in Part 02. Also create a day of the week variable in a similar fashion as you did for month or hour.

The *Accident_Severity* variable has the value 1 if the accident was fatal. Create a new variable, *Fatal* that is 0 if the accident did not result in a fatality and 1 if the accident resulted in a fatality.

### Simple ddply

Create simply summary of the number of fatal accidents by day of the week. Hint: simply add the *fatal* variable over the day of the week. Use the *summarize* method and then use the function() method.

You should get

```
  dow nfatal
1   0    305
2   1    221
3   2    226
4   3    210
5   4    208
6   5    261
7   6    300
```

### More complex ddply

The above only reports the number of fatalities, but not the fraction of accidents with injury that resulted in fatalities. Modify your function to compute the total number of accident, the total number of fatalities, and the fatality proportion by day of the week.

Again use the summarize and function methods.

You should get:

```
  dow naccidents nfatal      pfatal
1   0      16794    305 0.018161248
2   1      22452    221 0.009843221
3   2      23045    226 0.009806900
4   3      23018    210 0.009123295
5   4      22810    208 0.009118808
6   5      25475    261 0.010245339
7   6      20820    300 0.014409222
```

## More complex ddply

You are not restricted by a single *by* variable. Create a new variable for weekend vs. weekday depending of the day of the week.

Now compute the proportion of fatalities for each combination of month and weekend/weekday.

You should get:

```
   month DayType naccidents nfatal      pfatal
1      1 Weekday       7643     68 0.008897030
2      1 Weekend       2994     52 0.017368069
3      2 Weekday       9065     92 0.010148924
4      2 Weekend       2659     34 0.012786762
5      3 Weekday      10280     89 0.008657588
6      3 Weekend       2885     47 0.016291161
....
```

## Comments from the marker.

Some general comments from the marker:

- Students need to familiarize themselves with using *RStudio* or *R* commands to save plots, instead of taking screen captures. The resolution on screen captures is really bad and looks unprofessional! There are better ways!

  If you are using ggplot() then all you need do is

  ```
  plotname <- ggplot(....
  plotname # to display the plot on your Rstudio window
  ggsave(plot=plotname, file='plotfile.png', h= , w=, units=, dpi=)
  ```

  If you are using Base *R* graphics (AVOID), then you need to wrap your code before/after as shown below

  ```
  png("plotfile.png")
  plot(x,y....)
  dev.off()  # Close the output plot file.
  ```

  Don't forget the dev.off() command to save the information to the file.

- In the same vein, please nicely reformat tables from *R* output. Would you put a screen capture of *R* output in a report for an employer? Textual information can be captured from *R* using the *sink()* function as explained in earlier assignments

  ```
  sink('file.name.txt', split=true)
  R commands that generate textual output
  sink() # to close the file
  ```

  Don't forget the closing *sink()*.

### Part 01 *Cereal*

Some general comments from the marker:

- The main deduction was for not including a hypothesis test of whether there is a difference in cereal sugar content for different shelves (coded no HYP).

- The other main deduction was for not including a proper legend for the plot and the table
- Also, many students interpreted the confidence interval plot, saying such which shelves' means were higher than others, but did not give formal evidence for differences among shelves (i.e. confidence intervals for the mean do or do not overlap)

The analysis for this dataset was virtually the same as for HW#2, so I didn't assign too many marks to it. And indeed, everybody without exception did that part right: got the CI's for the means for each shelf, noted that some intersected and some didn't, and made appropriate conclusions – well done!

The part that required people to explain why regression was not appropriate did have quite a few wrong answers, most of them different. I explained as best I could in the comments why they were insufficient.

For correct answers I accepted the minimum of 'shelf is a categorical variable", but here are some particularly well-written explanations from two students:

> "Fitting a simple regression of sugars vs. shelf is not an appropriate method because the simple regression treats the different levels of shelf as quantitative variables (numeric value 1 to 3) instead of qualitative variables (factors: low, medium, high)."

or

> "... it would be impossible to make predictions for a shelf whose value is 1.5, because the shelf variable can only fall into one of the three groups."

I had to take off a lot of marks for TOO MANY DECIMAL PLACES this time, which is weird considering how late into the semester this is. When you feel you must include a table with numbers into your writeup, make sure that the reader can get information within a few seconds of looking at the table. Part of it is informative row and column names, part of it is appropriate reference in the writeup, pointing out the significance of certain features, and part of it is not including a deluge of numbers in the table. And while *SAS* outputs can be "groomed" to look decent, *R* outputs generally are pretty ugly – so I don't recommend copy & pasting *R* outputs into the writeups at all! Rather, extract whatever information you consider important, ROUND IT to the digit that you consider important and then make your own table in Word.

As you saw in the previous assignment, the *round()* can be used to round results to a more suitable number of decimal places.

## Part 02 *Accidents*

- Not including a legend on the plot.
- The main deduction was for not catching that there is "heaping" - there is no relevance to police because the data is inaccurate. Most people are reporting accidents to the nearest multiple of 5 minutes or nearest half hour. As an example of heaping, what time did you arrive at SFU today? If you said 8:30, was it really 8:30 and not 8:29 or 8:31 etc.
- Some students also lost marks if they mentioned something along the lines of heaping (i.e. continuous variables being ordinal in practice) if they also said something contradictory elsewhere (i.e. saying that there were more accidents at 5 minute intervals).

Once again, the actual coding and analysis portion of this exercise was trivial, so I assigned half the points to the conclusion students drew from the histogram. Anyone who claimed that the features in the histogram represented any useful information that had to be acted on got marks off.

There was a popular theory amongst the papers that the spikes at 00 and 30 minute points were caused by the fact that appointments generally took place at those time, and people were rushing more during those minutes than during others. First of all, if I'm rushing to make an appointment at 11:30, I'm going to be running a red light not at that precise moment, but probably something like 11:12 – cause I'd still have to leave time for parking and waiting for the elevator. And even if that theory is true, how could those "rushing to make the appointment" accidents possibly account for a whopping 20% of all accidents??

However, this exercise has an important point to make: when you go to a fine enough scale, all data ends up being ordinal, even though time is a continuous variable.

Another very common mistake that I took points off was the title of the histogram claiming that it represented "Numbers of Accidents by minute of the hour", while the histogram itself plotted relative frequencies of accidents (relative to the total number of accidents).

Perhaps I'm being too anal about this, but the truth of the matter is – a plot is the first thing that draws the eye of the reader in any report; in fact, with most reports you can easily expect the reader not to do any more than look at the pictures. And so, a confusing title, or improperly named axes can really give the wrong impression about the data to someone who doesn't know the proper context.

Finally, I took marks off for not using 60 bins for the histogram. Not only does it just make sense, but failure to do that would cause you to miss the spikes at 5 minute intervals.