Stat-340/341/342 – Assignment 6 – 2015 Spring Term

Part 0 - Welcome to *R* - Getting started - nothing to hand-in

Welcome to the *R* section of Stat-340/341/342. In this assignment, you will see how to use *R* to perform a simple analysis in Part I, and more complex analysis in Part II. Don't worry too much if you don't understand all of the *R* code as we will go over much of it during the upcoming term. You should however, make some effort to understand what the code is doing, even though you don't fully understand the internals. This is especially true when we want repeated analyses using the *split-apply-combine* paradigm and the *plyr* package.

Similarly, creating plots in *R* is flexible and frustrating because what takes a single line in *SAS* often takes several lines in *R*. Be patient and persevere. There are two types of plotting in *R*. Base *R* Graphics which follow the *pen-on-paper* paradigm and the *ggplot2* package (`http://ggplot2.org`) that follow a more formal grammar of graphics. The *ggplot2* package has become the de facto standard for graphing in *R* and the use of Base *R* routines has fallen off.

This assignment also shows you how to use *RStudio* and *R Markdown* to create an HTML file that contains both code and output in the same file. *R Markdown* is a very powerful feature, and in later assignments, we will show you how you can embed parts of your written report around chunks of *R* code without having a complete computer dump. As you will see, one of the very frustrating parts of *R* is simple getting decent output – what was trivial in *SAS* using PROC PRINT, labels, format, etc is excruciating difficut in *R*.

1. Install *R* and *RStudio* on your machine. It is not necessary to install *R Commander*. Visit

http://www.stat.sfu.ca/~cschwarz/CourseNotes/HowGetSoftware.html
for details.

2. Create a separate a separate personal library for packages. See the link above for details. Install the packages given at the above URL.

3. Read Slides 1–12, and Slides 53–68 at http://www.stat.sfu.ca/~cschwarz/Stat-R/CourseNotes/R-slides.pdf as a quick intro to *R* and *RStudio*. There are also many YouTube videos on using *RStudio* to help you get started.

Part 1 - Breakfast cereals - Easy

In this part of the assignment you will learn

- How to read in data from a *.csv file.
- How to check that you have read the data properly using the *str()*, *dim()*, *head()* and *tail()* functions.
- How to construct a basic scatterplot with jittering using *ggplot()*.
- How to fit a simple linear regression using *lm()*
- How to extract information from the fit using the standard method functions.
- How to get predictions from this fit.
- How to plot the fitted line on an existing plot.
- How to save TEXTUAL and GRAPHICAL information to files.
- How to create an HTML notebook that integrates source and output into one file.

Many people in North America start their morning with breakfast cereals. Every two years the Section on Statistical Graphics of the American Statistical Association sponsors a special exposition where one or more data sets are made available, analyzed by anyone interested and presented in a special poster session at the Annual Meeting. One of the datasets in past years was nutritional information about breakfast cereals.

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. Launch *RStudio*, navigate to the directory with the *.csv file, and and set the working directory (look the *More* drop down menu in the *File* tab in the lower right pane of *RStudio*)) to point to the directory where the cereal dataset is located. You can check that *R* is pointed to the correct directory by issuing the

```
getwd()
```

command in the *console* window. You can also see the current location of the working directory in the title bar of the Console pane in the *RStudio* windows.

4. Create a new *R* script, and add the comments to the start of your file to both identify your name, student number, purpose of the script, revision dates, etc. See the Assignment web page for details of sample code.

5. It is common practice to specify any additional packages that need to be loaded at the start of your script. Type the following commands into your script and run the commands:

```
# load required libraries
library(ggplot2)
library(GGally)
```

This will load the *ggplot2* package which is the preferred method of making plots. The *GGally* package provides some additional functionality to the *ggplot2* package. If you get an error message in your Console pane of *RStudio*, you likely forgot to install the two packages. Refer back to the instructions on installing *R* provided earlier.

6. Unlike *SAS*, *R* does not make a clear distinction between data processing function (the DATA step) and analysis function (PROCs) and uses functions to do both. The *read.table()*, *read.csv()* and other related functions are used to read data from external locations into the *R* workspace. Use the *help()* function to read the options available for each of the these functions – there are many!

   Notice that many of the arguments to *R* functions can be specified with default values that are shown in the help results for that function.

   Read the cereal data into *R* and create a suitable data frame. Then check the structure of the object. Your code will look similar to:

```
cereals <- read.csv('cereal.csv', as.is=TRUE,
    strip.white=TRUE, header=TRUE, fill=TRUE)
str(cereals)
```

   Be careful to match the parentheses and the quotes around the file name. You can use either single or double quotes (but they have to match).

   How does a *data.frame* differ from a *matrix*?

   What does the *as.is = TRUE* argument do? Hint: try running the above code with and without that argument to see the different in the *cereals* data frame. What does the *strip.white* argument do? What does the *header* argument do?

What does the *fill=* argument do? Hint: Create a copy of the *cereal.csv* file, remove some of the values in the first row, and try and read in that modified copy with and without the *fill=* options specified. What option in *SAS* does this correspond to?

Unlike *SAS*, *R* does NOT have a facility to add metadata (such as the output format) to individual variables (groan), so there is NO equivalent of the *label*, or *format* statement from *SAS*.

The **data.frame** is the standard object to store data sets. Do you understand the underlying properties of a data.frame, i.e. what is the data structure, what are the data types of each variable in a data.frame, how does it differ from a matrix?

7. It is always a good idea to verify that you've read in the data properly.

   Unlike *SAS* there is no *log* file so you have to explicitly verify the data. You did part of this verification with the *str()* function as seen above.

   You should also verify the dimensions of the data.frame using the *dim()* function. This information was also given by the *str()* function – do you see it?

   Enter and run the following code:

   ```
   names(cereals)
   dim(cereals)
   cereals[1:10,]
   head(cereals)
   tail(cereals)
   ```

   What is the purpose of the above functions?

8. *R* has very powerful index indexing features. For example, type

   ```
   1:10
   ```

   on the Console pane of the *RStudio*. You notice that you will get a vector of the integers 1 through 10. So to print the first 10 records of the cereals data frame, use

   ```
   cereals[1:10,]
   ```

   Don't forget the comma (why do you think you need this?) and match the parentheses and square brackets. The square braces are actually an *indexing* function and so cannot be interchanged with the parentheses.

9. Do you notice some unusual values for *Potassium* or *Serving Size*? What do these represent? The same convention might be used for the other variables as well.

10. Recode some data values as appropriate.

    Indices to data frame, vectors, matrices and arrays do not have to be integers (as you used 1:10 above). You can also use logical elements (TRUE and FALSE) to index these data structures.

    Type the following code

    ```
    cereals$potass
    cereals$potass ==  -1
    ```

    on the console. What is the purpose of the *$potass* string appended to the data.frame name?

    Notice the use of the == to represent testing for equality. CAUTION: When testing for equality you need to use a TWO equal signs. If you type a single equal sign, you likely will get an error.

    The second statement above creates a vector of TRUE and FALSE values where the potassium levels are or are not equal to -1. You can then use this vector of logical values to select elements of the vector using

    ```
    cereals$potass[ cereals$potass ==-1] <- NA
    ```

    to assign the *R* missing value (NA) to those elements of the vector that are equal to -1. Think carefully about what this statement means.

    It would be good practice to do the recoding for ALL variables, and not just the ones in this particular dataset that appear to have problematic values. Repeat for all of the other columns as needed. (Do you need to check the name of the cereal for missing values?)

    You can check if there are any -1 values anywhere in the numeric variables of the data frame by using the

    ```
    any(cereals == -1, na.rm=TRUE)
    ```

    What does the *any()* function do? What do you thing happens with variables that have the *character* data type?

    Unlike *SAS*, *R* propagates missing values (NA) throughout any operation. A common argument to many functions is the *na.rm=* argument – what does this do? Try this function with and without that argument.

    In this example, the SAME missing value was used throughout the dataset. What do you get when you type

    ```
    cereals == -1
    ```

    on the console? Can you think of a "simpler" way to set all -1 to NA in the entire data frame in one fell swoop?

11. Print out the first 10 rows of the data frame after you have fixed up all of the columns for potential missing values.

12. Use the *ggpairs()* function to create a scatter plot matrix (also known as a casement plot) involving calories, protein, fat, complex carbohydrates, sugars, sodium and vitamins.

```
scatmat <- ggpairs(cereals[,c("protein","fat","carbo",
                 "sugars","sodium","vitamins")],
  title="Casement plot of variables from cereals dataset")
scatmat
```

    Interpret the plot. What do you conclude from these plots. Why do plots involving vitamins look "odd"? What is "odd" about plots involving protein and fat (and other variables)?

13. We now want do a regression of calories against grams of fat. Start my making a simple scatter plot of the two variables:

```
plotcalfat <- ggplot(data=cereals, aes(x=fat, y=calories))+
   ggtitle("Calories vs fat content")+
   geom_point()
plotcalfat
```

    Every *ggplot()* plot has several standard parts. The *ggplot()* specifies the data.frame where ALL the variables for a plot are located, and the aesthetics (which is the X and which is the Y variable). Then there are additional layers added - such as titles, and then the actual things that are plotted (called *geometries*). You may find the *ggplot2* reference card helpful which is available at:
    `http://people.stat.sfu.ca/~cschwarz/Stat-R/CourseNotes/R-manuals/R-ggplot2_ref_card.pdf`

    This plot is not completely satisfactory - why? As before, we can jitter the points for plotting purposes. Compare the previous plot to

```
plotcalfat2 <- ggplot(data=cereals, aes(x=fat, y=calories))+
   ggtitle("Calories vs fat content - point jittered")+
   geom_jitter()
 plotcalfat2
```

    It is possible to adjust the amount of jittering as you will see later in the course.

14. *R* uses the *lm()* (linear model) function for simple regression, ANOVAs, and ANCOVAs. Here *simple* refers to models with only a single random effect (the error term) in the model.

In most modeling function of R, a formula is used to declare the model using variables found in a data frame and a special object is returned. Fit the model using

```
my.fit <- lm( calories ~ fat, data=cereals)
```

The variable to the left of the $\sim$ represents the response variable, and the terms to the right represent the predictor variables.

When you do a *str(my.fit)* you see that has a complex(!) structure. In particular, look at the final line from the *str()* output and you see that *my.fit* has a CLASS of 'lm'. A class is like an extension of a data structure – it tells any function that tries to access parts of this object how it is structured.

15. For most model functions, there are standard *accessor functions* (also called *methods* in the object orientated programming vernacular).

    For example, the ANOVA table and hypothesis testing about effects occurs via the *anova()* method. Construct the ANOVA table using the code:

    ```
    anova(my.fit)
    ```

    What is the hypothesis being tested, and what do you conclude?

    CAUTION: The default hypothesis tests from the *anova()* function in *R* are incremental or Type I tests. In many cases you will want marginal or Type III tests. Only in the very simplest modes with the *anova()* function give you the correct output.[1]

16. Similarly, you can get the table of coefficients using

    ```
    summary(my.fit)
    ```

    This has many of the standard features you would expected in the output from a linear model. What is the equation of the fitted line. Interpret the coefficients.

    Because this is an object in its own right, if you type *str(summary(my.fit))* you see that this too has a complex structure!

17. We can also get the individual coefficients, the standard errors of the coefficients, and the confidence intervals for the slope and intercept using:

    ```
    # Extract other bits
    my.fit.coef <- coef(my.fit)
    my.fit.coef
    ```

[1] An this is what distinguishes you from a trained monkey. Many users of R are unaware that the *anova()* functions often gives WRONG results!

```
vcov(my.fit)
my.fit.se <- sqrt(diag(vcov(my.fit)))
my.fit.se

my.fit.ci <- confint(my.fit)
my.fit.ci
```

Notice that the *summary()* method didn't provide confidence limits for the coefficients. The *confint()* method computes confidence limits for the extracted coefficients (you can select which coefficients are extracted and the confidence level – read the help on this function) As you say from a previous assignment, the standard value for calories/gram of fat is 9. Based on the confidence interval what can you say about the standard value based on the results of this dataset?

R is quite demanding about your statistical knowledge. *R* expects that you understand that the standard error is found from the square root of the diagonal of the sampling distribution variance-covariance matrix (! Try saying that quickly 10 times). As I will often say, *R* is free, but not cheap. The SAMPLING covariance matrix for the estimates (do you understand what is meant by a SAMPLING covariance) is extracted using the *vcov()* method and the SE of the coefficients is the square root of the diagonal elements. Use the *vcov()* method to extract the SAMPLING covariance, and then *sqrt()* and *diag()* functions to get the SE. Do these match those in the table of coefficients presented in Assignment 1? Interpret the standard error of the estimates coefficient for grams of fat.

We often want to put various things together into a little table. This is often done by creating a data.frame with the relevant columns. Enter:

```
my.table <- data.frame(Coef=round(my.fit.coef,2),
                       SE=  round(my.fit.se,2),
                       CI=round(my.fit.confint,2))
my.table
```

Notice that unlike *SAS*, *R* does NOT have the equivalent of the *label*, *format* and other meta data features- GROAN. That means we need to construct our tables in excruciating details and still don't get very nice looking tables.

What is the purpose of the *round()* function?

18. We often want to save TEXTUAL output to a file for inclusion in a word process etc. *R* has a clumsy syntax:

```
sink('assign06-part01-cereal-table1.txt', split=TRUE)
my.table
sink()
```

The first *sink()* function start sending any subsequent output to the file listed and also the console. This continues until the closing *sink()*.[2]

DON'T FORGET THE CLOSING *sink()*; otherwise your output will continue for ever. After you run the code, you should see the new text file in your directory.

19. We want to add the fitted line to the previous plot. Enter and run

```
plotcalfat3 <- plotcalfat2 +
   geom_abline( intercept=my.fit.coef[1], slope=my.fit.coef[2])
plotcalfat3
ggsave(plotcalfat3, file='assign06-part01-calfat.png',
       h=4, w=6, units="in", dpi=200)
```

Notice the *R* allows you to add to previously created plots. The *geom_ abline()* adds a line to a plot with the given intercept and slope. Do you understand why we needed to index the *my.fit.coef* vector to get the two values?

To save *ggplot2* graphs, the *ggsave()* function copies the plot the specified file (the type of plot, i.e. a *.png graphic, is specified using the file suffix). The rest of the function specifies the height and width (in inches, but you can could change the units to metric units), and the resolution (dots per inch).

After running the *ggsave()*, check your directory to see the saved file.

20. Unlike *SAS*, *R* does NOT automatically generate residual plots and the other traditional diagnostic plots. The *ggplot2* package does not have a built-in diagnostic plot for linear models, but I have included one such function in my "helper" functions that are accessed using *source()* function. The *sf.autoplot.lm()* then creates the standard diagnostic plots for a linear model - much like *SAS*. The code is:

```
# Create the model diagnostic plot
source("http://www.stat.sfu.ca/~cschwarz/Stat-650/Notes/MyPrograms/schwarz.functions.r"
diagplot <- sf.autoplot.lm(my.fit)
diagplot
```

What do you conclude from this diagnostic plots? What should a "good" plots look like? What do "bad" plots look like and what do "bad" plots tell you about an assumption that is likely violated in the model?

21. How about predictions for a new value of X? To make new predictions, we first construct a data.frame with the same predictor variable (*fat* – note that the name must also match the case (upper or lower) as used in the fit).

[2] This is similar to the *ODS ... ODS CLOSE* statements in *SAS*.

```
newfat <- data.frame(fat=c(4))
newfat
```

The *predict()* method can be used to get predictions:

```
my.fit.at4.mean <- predict(my.fit, newdata=newfat, se.fit=TRUE,
        interval="confidence")
my.fit.at4.mean
my.fit.at4.indiv <- predict(my.fit, newdata=newfat, se.fit=TRUE,
        interval="prediction")
my.fit.at4.indiv
```

Are these values consistent with your plot?

Do these match the values you obtained from *SAS*?

22. Finally, it is a real pain to try and get a nice summary report out of *R*. There is no nice equivalent of the ODS PDF FILE=xxx and ODS PDF CLOSE feature of *SAS*.

    One of the easier ways to create an HTML file with all of the output and code lumped together is using *RStudio*.

    Be sure that you have saved all of your code in a suitable file.

    Click on the "Compile an HTML notebook..." button to generate an HTML file that contains all of the code and output.

    The first time you try this, *RStudio* may complain that it is missing the package *knitr*. Go to the files pane, click on *Install Packages*, and install the *knitr* package. You will need to be connected to the internet for this to work.

    If there is an error in your code, the compilation process will stop with surprising UNINFORMATIVE error messages. You have to look carefully at the error message (Hint: press the *Output* button on the Console pane for more details).

    Keep trying to compile and creating a notebook until it runs without errors. You will then have a *.html file in your directory that you can hand in. This is nice document that integrates both code, textual output, and graphical output into one nice document.

    You can also compile your output into a MSWord or PDF file if that is easier to use?

    CAUTION. When you compile an HTML notebook, NO output will be written to any *.txt file from any of the *sink()* pairs – Groan - *R* is free, but not cheap. So following the successful compilation, you should select ALL of your code and run it in directly from the *RStudio* script window to update any text files created.

Hand in the following using the electronic assignment submission system:

- Your *R* code that did the above analysis.
- The HTML notebook 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.
  - The fitted regression line. Interpret the slope, se, and confidence interval for the slope.
  - Is there evidence of a relationship between the grams of fat and calories? Is the relationship consistent with the standard value of 9 calories/gram of fat? How do you tell?
  - Give a possible reason why the confidence interval for the slope is so wide – hint – what happens if you don't include other explanatory variables in a model?
  - What is the estimated calories for cereals with 4 grams of fat? Give both the confidence interval for the mean and the prediction interval for a single cereal and explain when each might be used.

  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.

Don't forget you can look at the solutions from Assignment 1 to 5 to see samples of write-ups and to compare your output from that from *SAS* – they should be identical!

Part 2 - Titanic Data - Intermediate

In this part of the assignment, you will learn

- How to read data from the web
- How to use the *table()* and *xtabs()* functions
- How to compute odds, log-odds and the inverse from base probabilities.
- How to fit a two-factor logistic ANOVA
- How to extract the marginal estimates from the model using the *lsmeans* package.
- How to plot a profile plot with confidence limits at each point.

Many people are familiar with the *Titanic* disaster. Data on the passenger aboard the liner are available at `http://www.statsci.org/data/general/titanic.html`.

The data have already been cleaned up, so you don't have to do any data screening.

1. Create an *R* script, and add the commands to the start of your file to both identify your *R* code and to identify your *R* output. See the Assignment web page for details of sample code.

   Launch *RStudio*, and change the working directory (look under the menu items) to point to the directory where you want your script and output stored. You can check that *R* is pointed to the correct directory by issuing the

   ```
   getwd()
   ```

   command. This is a good idea in general at the start of your program script so that when you have the output, you can identify where the data was located etc.

   Similarly, it is a usually good idea to clear your workspace of detritus before starting a new project by using the

   ```
   rm(list=ls())
   ```

Create a new script with the proper header lines at the start of the script. All of these previous commands commands should appear near the start of your script.

2. Read in the dataset directly from the web using something along the lines of

```
titanic <- read.table(url("http://www.statsci.org/data/general/titanic.txt"),
           sep='\t', header=TRUE, fill=TRUE, as.is=TRUE, strip.white=TRUE)
```

   Because the dataset is tab delimited, you need to specify the *sep='\t'* (backslash t) argument on the function call. Notice that the dataset contains the variable names on the first lines, so specify the appropriate *header=* option. Don't forget to specify the appropriate *as.is=*, *fill=* and *strip.white* arguments.

   R automatically adjusts the lengths of the character strings, so you don't have specify a maximum length. Notice that some ages are given as NA to indicate a missing value.

3. Look at the structure and dimensions of the data frame, the first few and last few records to ensure that you've read the data properly. Refer to Part 01 of this assignment on how to do this.

4. We want to create a summary table for the number of passengers that lived and died as a function of passenger class and sex.

   R does NOT have a similar function as *Proc TABULATE* in *SAS* – a pity, as *Proc TABULATE* is very versatile and one of the most common procedures used!

   The *tables()* function can be used to get simple contingency tables:

```
with(titanic, table(PClass, Sex, Survived, useNA='always'))
```

   Unfortunately, *R* is NOT consistent in how you specify the data.frame to use for a function. For example, in Part 01 of this assignment, you specify *data=cereal* for the *lm()* and *ggplot()* functions. But the *table()* function requires a different method.[3]

   The *with()* function provides a data frame (in this case *titanic*) in which to look up the variables *PClass*, *Sex*, and *Survived*. You saw the use of the *data=* feature in the *lm()* function in question 1, but not all functions in *R* use this paradigm (Argh – this lack of consistency is a major stumbling block to learning *R*).

   What does the *useNA=* argument do?

   What happens if you change the order of the variables in the previous statement.

[3] *R* is free, but not cheap.

Do you understand the information presented?

The *xtabs()* function is similar, but uses a formula to specify the dimensions of the table:

```
xtabs(~Survived+Sex+PClass, data=titanic)
```

Notice how the *xtabs()* function uses the *data=* argument to specify the data.frame[4]

5. It is quite common to use odds and log-odds in the analysis of categorical data (e.g. in logistic regression). Comparison of proportions is then equivalent to comparisons of odds or log-odds through the odds ratio or log(odds ratio). Be sure you understand what these all refer to – read the chapter on Logisitic Regresion in `http://www.stat.sfu.ca/~cschwarz/CourseNotes`.

   Let's first ignore the passenger class and sex, and compute the probability of survival, the odds of survival, and the log-odds of survival.

   Get the total number of passenger that survived or died over all sexes and passenger classes using the *table()* or *xtabs()* function. We find:

```
Survived
  0   1
863 450
```

   The overall probability of death is $p_{death} = 863/(863 + 450) = 0.657$. The overall odds of death is found as $ODDS_{death} = p_{death}/(1 - p_{death}) = 863/450 = 1.91$ and the log-odds of death is $LOGODD_{death} = \log(1.91) = 0.65$. Why? Of course you should do these computations in R(!) using something like:

```
overall.surviv <- with(titanic, table(Survived))
overall.surviv

overall.prob.death <- overall.surviv[1]/sum(overall.surviv)
cat("Overall prob of death ", overall.prob.death, "\n")

overall.odds.death<- overall.surviv[1]/overall.surviv[2]
cat('Overall Odds of death: ',overall.odds.death,'\n')
```

   Also compute and print the overall log-odds of death.

   The *cat()* function prints its arguments to the console; the terminating '$\backslash n$' (backslash n) starts a new line when the next printing takes place.

   Now compute and print the corresponding odds of survival. What do you notice about the relationship between

[4] Sigh, R is free, but not cheap.

- p_{death} and $p_{survive}$?
- $ODDS_{death}$ and $ODDS_{survive}$?
- $LOGODDS_{death}$ and $LOGODDS_{survive}$?

6. The *glm()* (generalized linear model) function is the equivalent to *PROC GENMOD* in *SAS*, and can be used to do this directly:

```
overall.glm <- glm(Survived ~1,
     data=titanic, family=binomial(link=logit))
```

   To the left of the tilde in the formal is the categorical response variable (the survival status). The *glm()* function allows for several forms of the input data – the most common is a numerical variable with two values (typically 0 and 1) and it treats the '1' as the category of interest (in this case survival).

   To the right are the classification variables (in this case, none as we want to model survival over all passenger classes and sexes). The *family=* argument indicates that we are doing logistic regression.

   The ~ 1 formula is analogous to an INTERCEPT ONLY model in a simple linear regression. What is the estimate of β_0 in regular regression under these cases? The analogous estimate occurs in *glm*.

   Use the *summary()* method to look at the estimated intercept in the *overall.glm* object. By looking at our 'hand' computations earlier, does it estimate the log-odds of survival or death?

   Use the *coef()* and *confint()* methods to extract the estimate and the 95% confidence interval for the log-odds of survival from the *overall.glm* object. Put them together into one vector using the *c()* function and print them out.

```
overall.logodds.glm <- c(coef(overall.glm),confint(overall.glm))
cat("Estimate of overall log-odds of survival ",
       "and 95 ci from glm",overall.logodds.glm,"\n")
```

   We now want to estimate the odds (i.e. on the anti-log) scale. Simply take the *exp()* of the estimate and the 95% confidence interval. Again, print out your answer.

   Finally, convert the estimated odds to the original probability. How do you convert odds to probability – think of how to invert $ODDS = p/(1-p)$. This inverse transformation can be applied to the estimate and the confidence intervals, but not to the standard errors.

   Compare your final answers from *glm()* with your hand computations.

7. We now want a separate estimate for each combination of sex and passenger class and want to see if there are effect of sex and/or passenger class

on the odds of survival and if there is an interaction. This should now start to look familiar – see the previous assignments.

We want to fit a model using *glm()* that has the two factors and their interaction. In *SAS*, we used the *Class* statement to specify a categorical factor variable. In *R*, we need to create a new factor variable using

```
titanic$PClassF <- as.factor(titanic$PClass)
titanic$SexF    <- as.factor(titanic$Sex)
str(titanic)
```

Notice how the *str()* output shows that the *PClassF* and *SexF* are categorical (Factor) variables.

8. Now we can fit the two-factor *glm()* model:

```
sex.pass.glm <- glm(Survived ~ SexF +PClassF + SexF:PClassF,
                    data=titanic, family=binomial(link=logit))
```

Notice that the interaction term has a colon (:) rather than an asterisk as used by *SAS*, and that each term is separated by a plus sign rather than a space in *SAS*.

9. It is surprising difficult to get the Type 3 test from *R*.[5] The default action from

```
# Analysis of deviance table - CAUTION these are Type I (incremental)
# rather than Type III (marginal) tests, but only the interaction
# result is of interest
anova(sex.pass.glm, test='Chi')
```

are the incremental (Type 1) tests which are rarely useful. Fortunately, we are only interested in the results for the interaction term. The Type 1 and 3 tests are the same if the interaction term appears last in the model.

Notice that despite its name, this is NOT an ANOVA – this is actually an analysis of deviance.[6] This will be explained in more details in later courses in Statistics.

What do you conclude based on the ANODEV table?

10. We now want to get the marginal estimates of the log-odds (and probability). In *SAS*, we used the *lsmeans* statement. *R* has an similar functionality through the *lsmeans()* function in the *lsmeans* package.

Be sure that the *lsmeans* package has been installed. Then run

[5] *R* is free, but not cheap.
[6] *R* is free, but not cheap.

```
library(lsmeans)
sex.pass.glm.lsmo <- lsmeans::lsmeans(sex.pass.glm, ~SexF:PClassF)

sex.pass.glm.est.logodds <- summary(sex.pass.glm.lsmo, type="link")
sex.pass.glm.est.logodds

sex.pass.glm.est.p  <- summary(sex.pass.glm.lsmo, type="response")
sex.pass.glm.est.p
```

There are several packages that have a *lsmeans()* function. So to force *R* to use the *lsmeans()* function from the *lsmeans* package, you need to use the *lsmeans::lsmeans(...)* syntax. The name before the double colon (::) is the package name; the name after the double colon is the function name.[7]

This is a two-step process. First we create a *least-squares mean object*. And then we get a *summary* of the object on the *link* scale (i.e. in terms of log-odds), or on the *response* scale, i.e. in terms of probabilities.[8]

Do you understand the output created? Again, despite the term *lsmeans* being used everywhere, inference is about log-odds or probabilities and NOT MEANS!!![9]

11. Now we are set to draw the graph. Use the following code:

```
# Make the same plot using ggplot
library(ggplot2)
logodds.plot <-
  ggplot(data=sex.pass.glm.est.logodds,
     aes(x=PClassF, y=lsmean, group=SexF, shape=SexF, color=SexF))+
  ggtitle("Comparison of logodds of survival by sex and passenger")+
  xlab("Passenger Class")+ylab("log(Odds) survival and 95% confidence interval")+
  geom_point()+
  geom_errorbar(aes(ymin=asymp.LCL, ymax=asymp.UCL),width=0.2)+
  geom_line()
logodds.plot
```

The *ggplot()* function starts off by specifying the data frame that contains all of the plotting information. The *aes()* function specifies various aesthetics of the plot, i.e. what is the X and Y axes variables, is there any grouping of the data when lines are drawn, should points be distinguished by shape and/or color[10]. You can guess the purpose of the *ggtitle()*, *xlab()* and *ylab()* functions. The *geom_point()* then plots the

[7] *R* is free, but not cheap.
[8] *R* is free, but not cheap.
[9] *R* is free, but not cheap.
[10] It generally NOT a good idea to use color exclusively to distinguish among groups because color does not render well in black and white and many people are color blind.

points; the *geom_errorbar()* adds the 95% confidence intervals, and the *geom_line()* add the lines.

Notice we didn't have to worry about the range of tick marks for either axis and that saving the final plot is relatively straightforward.

There are many more options for the *ggplot2* package – we will run into some of these in the next few assignments.

12. Create a similar plot for the probability of survival using the *ggplot()* function..

13. Finally, create the HTML notebook file as you did in part 1.

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 who has had one course in statistics. You should include:
  - A (very) brief description of the dataset.
  - The plot you created of the log-odds of survival above.
  - An explanation of what is measured by the odds and log-odds and what the plot shows you about the log-odds across the three passenger classes and sexes.
  - There is an old saying "Women and children first" (see `http://en.wikipedia.org/wiki/Women_and_children_first`). Does this seem to apply in the case of males and females? How do you tell?

Part 03 - writing functions

In this part of the assignment you will learn

- How to write simple functions.

Writing functions is an important part of using *R* properly. A function is simply a set of command that take inputs and product outputs. The input are specified by the arguments to the function; the outputs are an object produced by the function. Unlike standard functions such as log, inputs to *R* functions and outputs from *R* functions can be any type of object.

1. Let's start with a simple function that computes the coefficient of variation for a set of numbers. The coefficient of variation (the c.v.) is defined as

$$cv = \frac{s}{\overline{Y}}$$

   R has functions that will compute the mean and standard deviation of a vector of numbers, so we have the building blocks for our function.

   Define a vector of numbers as

```
myvec <- c(1, 3, 5, 7, 9, 11, 13)
```

   Use the *mean()* and *sd()* function in *R* to compute the mean and standard deviation respectively.

   Now lets create simple function

```
cv <- function(y){
  # computes the cv of the vector y
   mean <- mean(y)
   sd      <- sd(y)
   cv      <- sd / mean
   return(cv)
} # end of cv function
```

   To compile the function, highlight the function in *RStudio* and run it. If there are no syntax errors, you should get a silent return, but now the object *cv* should appear in your workspace.

   You can use your *cv* function in the usual ways

```
cv(myvec)

mycv <- cv(myvec)
cat("The cv is ", mycv, "\n")
```

This should give

```
> mycv <- cv(myvec)
> cat("The cv is ", mycv, "\n")
The cv is  0.6172134
```

2. Make a function that compute the standard error from a simple random sample. Recall that the the *se* for a SRS/CRD is found as

$$se = \frac{s}{\sqrt{n}}$$

The *length()* function returns the number of elements in the vector.

This should give you:

```
myse <- se(myvec)
cat("The se is ", myse, "\n")
The se is  1.632993
```

3. Let's expand the function slightly, to return the mean, sd, cv, se, and rse (relative standard error) of a vector of values. The values should be returned in a vector with the appropriate names for the elements.

```
summary <- function(y){
  # computes the se from y
   n        <- length(y)
   mean <- mean(y)
   sd       <- sd(y)
   cv       <- sd / mean
   se       <- sd / sqrt(n)
   rse    <- se/mean
   myres   <- data.frame(mean, sd, cv, se, rse)
   return(myres)
} # end of summary function
```

Use the summary function on the *myvec* object. This should give you

```
> summary(myvec)
  mean       sd       cv       se       rse
1    7 4.320494 0.6172134 1.632993 0.2332847
```

4. Write a function that takes a vector of numbers and then compute the values needed for the central part of box-plot (i.e. the median, 25th, and 75th percentiles). Your function should return the three values.

Hint: *R* has a function *median()* and *quantile()* that will be helpful.

This should give you:

```
> boxplotvals(myvec)
    median q1 q3
25%      7  4 10
```

5. You saw in Part 1, how to fit a regression line between calories per serving and the grams of fat. Write a *R* function that takes the cereal data frame, does the regression, and returns the estimated intercept and slope, the se of each estimate, and the 95% confidence interval for each population parameter. Your function will look something like:

```
fat.reg <- function(cereal){
# create little report from the regression
    my.fit <- lm( calories ~ fat, data=cereals)

  # Extract the coefficient, confidence limits
    my.fit.coef    <- coefficients(my.fit)
...
   my.table <- data.frame(....
   return(my.table)
}

fat.reg(cereal)
 > fat.reg(cereal)
                 Coef       SE      LCL      UCL
(Intercept) 95.131579 3.141224 88.87394 5.409642
fat          9.806005 2.206897 88.87394 5.409642
```

Nothing to hand in, but these types of functions are the sort you will be expected to write on a term test.

Comments from the marker

General comments:

- There were lots of submissions without names in the header!
- I used the code TMDP for too many decimal places.

Part 01 Cereal

General comments from the marker:

- Not including/discussing prediction intervals (code: no PIs).
- Not discussing whether $\beta_{fat} = 9$.
- A few people said that the confidence interval was wide because the sample size is too small. This is only part of the answer.

There were no problems with getting the regression equation or the CI, or an interpretation of the slope right. I was pleased until I realized that a very large fraction of students simply copied your solutions from HW #1 verbatim, at least the paragraph relating to the regression line.

Nearly everyone claimed that the wide CI was due to the fact that there was only one explanatory variable, but this is also not entirely surprising, cause it was heavily hinted on in the assignment wording. I did not deem that explanation to be sufficient – I needed some evidence of understanding of the mechanism by which missing covariates affect residuals.

There were plenty of good explanations, but by far the best one I've seen was given by **********:

> "Consider two cereals, one with high sugar content and one with low sugar content. The calories of these two cereals will likely differ considerably because sugar adds calories. However, since sugar isn't used as a predictor in our model, this variation is considered random error and not accounted for by the model. "

A couple of students mentioned something about confounding variables, which was just weird – it would imply that there are some variables in the "true" model that affect both fat and calories. Confounding variables seldom affect residual variation. Read the definition at `http://en.wikipedia.org/wiki/Confounding`. Consider, for example, a variable completely confounded with fat. This would imply a perfect association between fat and this hidden variable, so once you add fat to the model, there is nothing more to explain in the response by the confounder and adding this confounded variable would NOT reduce SSE. As noted above, the problem is not including other explanatory variables (unrelated to fat) whose (hidden) effect are treated as noise.

Part 02 Titanic

The most common problems are:

- Omitting a legend for the plot.
- Not giving an explicit explanation of odds ratios and log-odds ratios.

By far the most common problem was with people posting the plot of the odds rather than the log(odds). I don't mind that at all – it's entirely possible to draw the same conclusions from either plot – but I insist that the description/title of the plot matches the content. I penalized plots of odds that claimed they were plots of log(odds). There is nothing more confusing that have labels on plots not matching what is actually plotted. Plot once, check twice is a good motto.

As with every write-up that includes plots, I was expecting a short blurb that explained what log(odds) measured and how to glean information from the plot. When I didn't see this, I took marks off.

A disturbingly large fraction of students saw it fit to copy your solution to HW #1 *verbatim*, at least in part and in some cases in its entirety. I'm sorry to say that your solution did not score very well. It involved completely irrelevant passages about how odds ratios are often used in epidemiological studies, and a rather confusing bit about "classes", which in the context of this problem (where passenger class has a different meaning), was very confusing.

I've included some stern warnings about copying other people's work verbatim without proper citation in University. In the cases where the copy-pasting

was most egregious I took off a point. As noted earlier, your professional status and integrity is heavily related to your personal integrity. I don't want to be a "copy police" and rely on your personal integrity and personal code of honor when you hand in your assignments.

Yes, the assignments are somewhat difficult, and yes, you likely have never had to do much writing in your past life, but University is the place to learn and stretch your personal mind space in a relatively low risk way. Ask alumni working in the real work and they will tell you that people with technical skills are "a dime a dozen", but people with technical skills AND communication skills (especially written communication skills) command top dollar in the workforce.

There is plenty of help available on campus to help with writing skills, e.g the Learning Commons `http://www.learningcommons.sfu.ca`; our Stat-300 course on writing, the Tutorials (which are seldom well attended!), etc.. These resources are often underutilized by students, so if you feel that these services are not meeting your needs, you need to let "them" (Administrators) know.

That said, there was still a large number of very good (and original!) write-ups, with clear and concise explanations. I gave out a lot of perfect marks.