donner.r: Explanation of code

Goals of code:

- Fitting a logistic regression model
  - calculating odds ratios from model parameters
  - confidence intervals for model parameters and odds ratios
- Plotting the data and fitted line

Fitting a logistic regression model: `glm( survival ~ age, family=binomial, data=donner)`

glm() is the basic function for generalized linear models, including logistic regression. glm() fits models for many different distributions, including normal (= gaussian), counts (e.g., Poisson), and 0/1 responses. The distribution is specified by the `family=` argument. For 0/1 responses, the family is binomial. *If you omit the family = argument, glm() assumes a normal distribution and the results are identical to using lm().* Not what you probably wanted!

The formula specification, `survival ~ age`, and `data=` arguments are identical to `lm()`. Remember that the **response variable is on the left** of the `~` and **the model terms are on the right**. If you specify `age ~ survival` you are requesting a t-test comparing the average age in the two survival groups.

The usual set of helper functions are available to work with glm() results. These include:

- summary(): to print out information about estimates and the likelihood ratio test
- coef(): extract the estimated coefficients
- confint(): to calculate profile likelihood confidence intervals for estimates
- anova(): likelihood ratio test of slope = 0
- predict(): to predict either the log odds or the probability for specified X values
- resid(): not especially useful for 0/1 data. Returns the deviance residuals as described in the text.

Obtaining useful results from the fitted model:

Odds ratio as X increases by 1: The slope, $\hat{\beta}_1$, is the quantity of interest. `coef()` gives you both the estimated intercept and estimated slope. You only care about the slope. You get the odds ratio by exponentiating the slope coefficient. A simple way to do this is to exponentiate both parameters and ignore the intercept.

Model comparison test of slope = 0: `anova()` will calculate the change in deviance (= -2 log likelihood) for the models with and without the slope. The simple use of `anova()` does not calculate a p-value. To the p-value, you need to specify the reference distribution, using `test = ` . For logistic regression, the appropriate test uses a Chi-square distribution, i.e. `test='Chisq'`.

Plotting the data:

The issue with just plotting the response (0/1) vs the X value is that many data points may overlap. The problem this causes is that you don't know whether a dot on the plot is one observation or many. There are two common ways to separate multiple observations: jittering and transparent colors.

Jittering is adding a small amount of random noise to each response. The code copies the response into a working variable (`jitterY <- donner$survival`) then adds random noise between -0.05 and 0.05 (`runif(length(jitterY), -0.05, 0.05)`).

Transparent colors are "less than 100% intensity". Overplotting these will increase the intensity. Instead of a color number or color name, e.g. `col=4` or `col='blue'`, you specify colors using 3 or 4 values from 0 - 255 specified using hexadecimal digits. 0 is "#00". 255 if "#FF". The # specifies this character string contains hexadecimal digits. 3 values gives you 100% intensity colors using Red / Green / Blue amounts. `col='#FF0000'` is red, `col='#00FF00'` is green, `col='#0000FF'` is blue, `col='#FFFF00'` is yellow, and there are a huge number of possible variations. When you add a fourth value, you specify the relative intensity, `col='#0000FF00'` will not appear (because the intensity is 0), `col='#0000FF10'` will be a very very light blue, and `col='#0000FFE0'` will be a nearly full intensity (E0) blue My code specifies an intermediate intensity (60). Some graphical devices don't support transparent colors. PDF files always do.

The plot commands in donner.r has a couple of bells-and-whistles to improve the appearance:

`ylim=c(-0.2, 1.2)`: increases the range of the Y axis

`yaxt='n'` followed by `axis()`: The default drawing of the Y axis has more tick marks than needed. This can be changed by telling R to not use the default labeling of the Y axis (`yaxt='n'`) then specifying how you want to label the axis using `axis()`.

`axis(2, at=, labels=, las=)`:

- The first argument specifies which axis this refers to (1 = bottom, 2 = left side, 3 = top, 4 = right side).
- at= specifies the locations to add information
- labels = specifies the text to write at those locations
- las=2 specifies that the text be written perpendicular to the axis (so horizontally for the left axis)

`axis(2, at=c(0.1), labels=c('No','Yes'), las=2)` will label the responses as Yes or No.

Plotting predicted probabilities:

`predict()` returns various predicted values. The default is to return the linear predictor, $\hat{\beta}_0 + \hat{\beta}_1 X_i$. These are the logit transformed probabilities. To get the predicted probabilities, add `type='response'`. By default, you get predicted probabilities for each observation in the data set used to fit the model. I find it more appealing to calculate probabilities for X values that I provide. This is done by creating a new data with the X values I want then getting predictions for those new values.

`newX <- 15:65`: creates a vector of values 15, 16, 17, ... up to 65. You can also use seq(from, to, by) to generate a sequence of values. from is the starting value, to is the ending value, and by is the increment.
`newdata=data.frame(age=newX)` in the predict() call: tells predict to use the values in a new data frame. That data frame has one column with the name given on the left-hand side of the = (here that's age) with values obtained from the expression on the right hand side of the = (here that's the contents of newX). The result is a vector of predictions.
`plot( , type='l', )`: draws a line between successive rows of the X and Y variables. To plot a line of predicted probabilities, you want the X values to be sorted, usually by increasing value.
`plot( , lwd=2, )`: optional argument to increase the line width. The default is 1.
`plot( , xlab=' text ', )`: labels the X axis with the specified text
`plot( , ylab=' text ', )`: labels the Y axis with the specified text

Overlaying data and lines:

Each call to plot() starts a new plot. Two helper functions, `points()` and `lines()` add points or lines to existing plots. So to plot the data and add the line, use `plot()` to draw the points then `lines()` to add the line. The axis limits are determined by the initial plot() command, so plot the component with the largest ranges of X and Y first.