Stat 5421 Lecture Notes: To Accompany Agresti Ch 4

Charles J. Geyer

January 23, 2024

License

Section 4.2.2 in Agresti

Data

```
heart.disease <- rbind(c(24, 1355),
                       c(35,  603),
                       c(21,  192),
                       c(30,  224))
snoring <- c("never", "occasionally", "nearly.every.night", "every.night")
x <- c(0, 2, 4, 5)
```

Logistic Regression

Then we fit the logistic GLM (usually called *logistic regression*) by

```
gout <- glm(heart.disease ~ x, family = binomial)
summary(gout)
```

```
##
## Call:
## glm(formula = heart.disease ~ x, family = binomial)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.86625    0.16621 -23.261  < 2e-16 ***
## x            0.39734    0.05001   7.945 1.94e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 65.9045  on 3  degrees of freedom
## Residual deviance:  2.8089  on 2  degrees of freedom
## AIC: 27.061
##
## Number of Fisher Scoring iterations: 4
```

We show this model first because it is the one we and Agresti recommend and is by far the most widely used.

Bernoulli Regression With Identity Link

Now we do the nonsense model, that no one ever uses, except teachers sometimes show it to show why all your intuitions developed from learning about linear models do not transfer to GLM.

```
gout.goofy <- glm(heart.disease ~ x, family = binomial(link="identity"))
summary(gout.goofy)
```

```
##
## Call:
## glm(formula = heart.disease ~ x, family = binomial(link = "identity"))
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.017247   0.003451   4.998 5.80e-07 ***
## x           0.019778   0.002805   7.051 1.77e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 65.904481  on 3  degrees of freedom
## Residual deviance:  0.069191  on 2  degrees of freedom
## AIC: 24.322
##
## Number of Fisher Scoring iterations: 3
```

We make a plot like Figure 4.1 in Agresti.

```
mu <- predict(gout, type = "response")
mu.goofy <- predict(gout.goofy, type = "response")
mu
```

```
##         1         2         3         4
## 0.02050742 0.04429511 0.09305411 0.13243885
```

```
mu.goofy
```

```
##         1         2         3         4
## 0.01724668 0.05680231 0.09635793 0.11613574
```

```
plot(x, mu, ylab = "probability", pch = 19, ylim = range(mu, mu.goofy))
curve(predict(gout, newdata = data.frame(x = x), type = "response"),
    add = TRUE)
grep("red", colors(), value = TRUE)
```

```
##  [1] "darkred"         "indianred"       "indianred1"      "indianred2"
##  [5] "indianred3"      "indianred4"      "mediumvioletred" "orangered"
##  [9] "orangered1"      "orangered2"      "orangered3"      "orangered4"
## [13] "palevioletred"   "palevioletred1"  "palevioletred2"  "palevioletred3"
## [17] "palevioletred4"  "red"             "red1"            "red2"
## [21] "red3"            "red4"            "violetred"       "violetred1"
## [25] "violetred2"      "violetred3"      "violetred4"
```

```
points(x, mu.goofy, col = "red", pch = 19)
curve(predict(gout.goofy, newdata = data.frame(x = x), type = "response"),
```

```
    add = TRUE, col = "red")
```

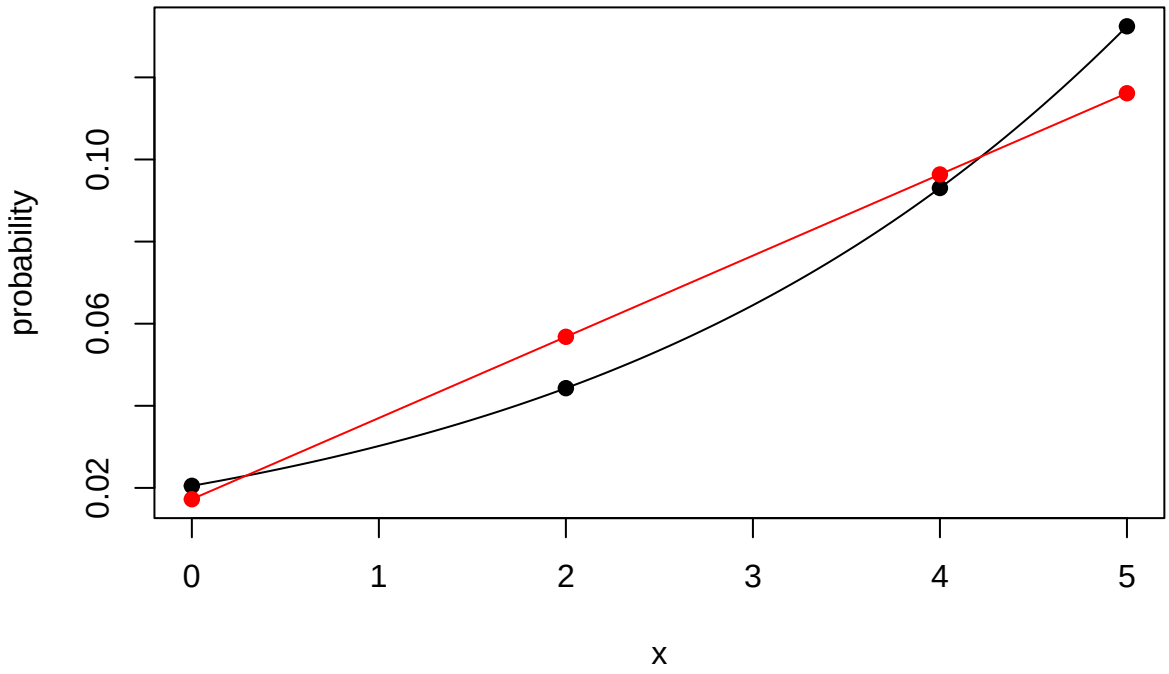


Figure 1: logit link (black) versus identity link (red)

Comments

Linearity versus Constraints

Linearity on the mean scale (identity link) fights the constraints that probabilities have to be between zero and one. Let us look at a few examples where we have probabilities near zero and one for our regression function.

Make up some data where the logistic regression model is true and probabilities go from near zero to near one.

```
z <- 1:100
theta <- z - mean(z)
theta <- 3 * theta / max(theta)
p <- 1 / (1 + exp(- theta))
y <- rbinom(length(p), 1, p)
rm(theta, p)
```

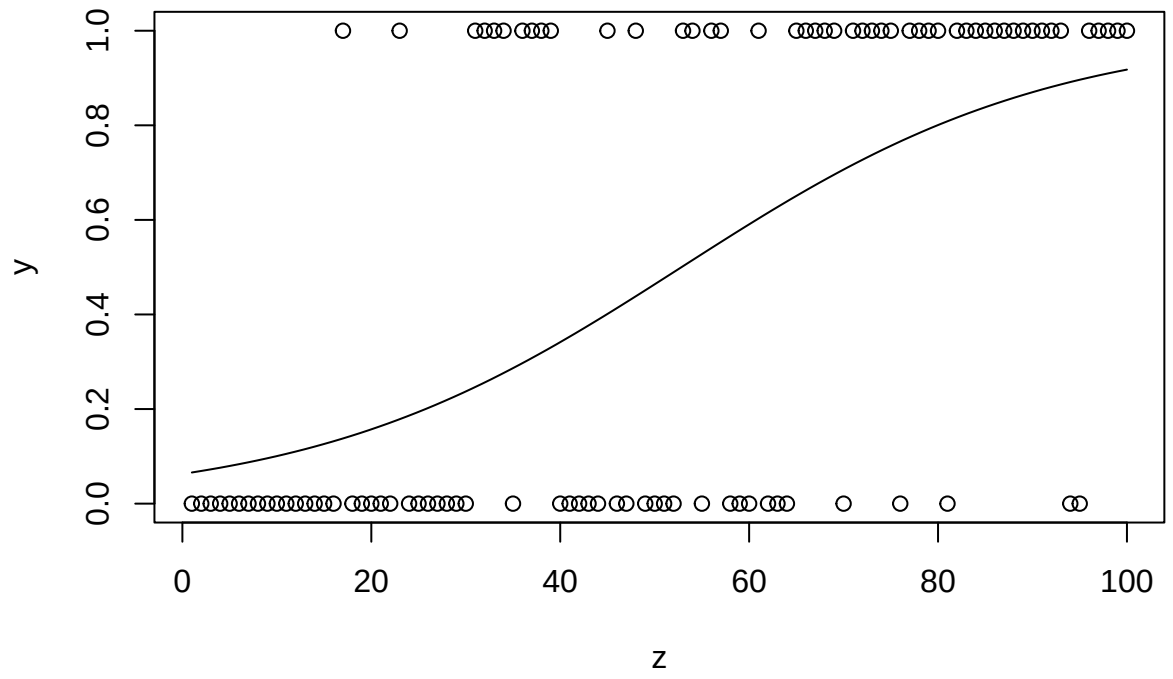
Now fit the logistic regression model and plot it.

```
gout.logit <- glm(y ~ z, family = binomial)
summary(gout.logit)
```

```
##
```

```
## Call:
## glm(formula = y ~ z, family = binomial)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.70322    0.58465  -4.624 3.77e-06 ***
## z            0.05116    0.01023   5.001 5.72e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 138.47  on 99  degrees of freedom
## Residual deviance: 101.71  on 98  degrees of freedom
## AIC: 105.71
##
## Number of Fisher Scoring iterations: 4
```

```
plot(z, y)
curve(predict(gout.logit, newdata = data.frame(z = x),
    type = "response"), add = TRUE)
```



Now fit the same data with identity link and add it to the plot.

```
gout.identity <- glm(y ~ z, family = binomial(link = "identity"))
```

```
## Error: no valid set of coefficients has been found: please supply starting values
```

This model is so bad that R function `glm` has trouble fitting it. Try two.

```
gout.identity <- glm(y ~ z, family = binomial(link = "identity"),
    start = c(0.5, 0))
```

```
## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds

## Warning: step size truncated: out of bounds
```

```
## Warning: glm.fit: algorithm did not converge
```

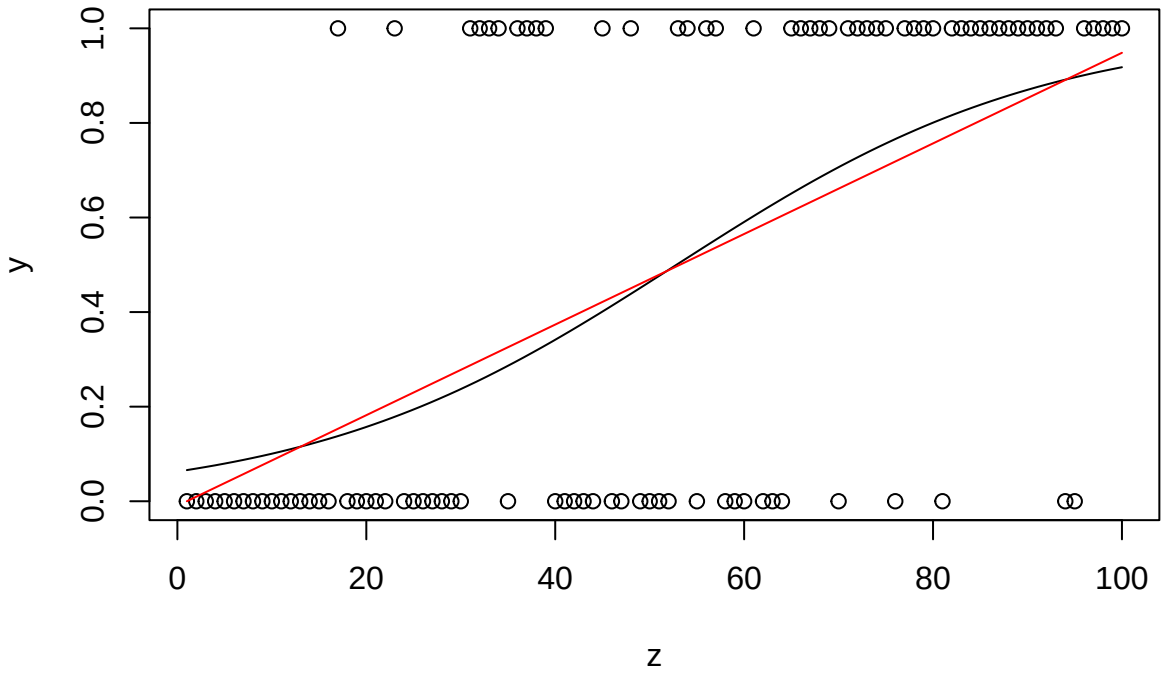
```
## Warning: glm.fit: algorithm stopped at boundary value
```

```
summary(gout.identity)
```

```
##
## Call:
## glm(formula = y ~ z, family = binomial(link = "identity"), start = c(0.5,
##     0))
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.0095787  0.0006464  -14.82   <2e-16 ***
## z            0.0095787  0.0006463   14.82   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 138.47  on 99  degrees of freedom
## Residual deviance: 100.57  on 98  degrees of freedom
## AIC: 104.57
##
## Number of Fisher Scoring iterations: 25
```

Now plot.

```
plot(z, y)
curve(predict(gout.logit, newdata = data.frame(z = x),
    type = "response"), add = TRUE)
curve(predict(gout.identity, newdata = data.frame(z = x),
    type = "response"), add = TRUE, col = "red")
```



- The identity link model is so bad that R function `glm` has a lot of problems with it and
- we can make the identity link look as bad as we want: it clearly is only paying attention to the endpoints of the data and ignoring everything in between.

Hypothesis Tests of Non-Nested Models

Although theory exists for comparing non-nested models (Cox, 1961, 1962, 2013), it is not widely known or used. The simplest procedure that makes sense is to compare both to the smallest model containing both. This is the model with four parameters that fits the data perfectly.

```
gout.super <- glm(heart.disease ~ snoring, family = binomial)
summary(gout.super)
```

```
##
## Call:
## glm(formula = heart.disease ~ snoring, family = binomial)
##
## Coefficients:
##                           Estimate Std. Error z value Pr(>|z|)
## (Intercept)                -2.0104     0.1944 -10.341  < 2e-16 ***
## snoringnearly.every.night  -0.2025     0.3010  -0.673  0.50111
## snoringnever               -2.0231     0.2832  -7.144  9.1e-13 ***
## snoringoccasionally        -0.8361     0.2608  -3.206  0.00135 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
```

```
##
##     Null deviance:  6.5904e+01  on 3  degrees of freedom
## Residual deviance: -1.2457e-13  on 0  degrees of freedom
## AIC: 28.253
##
## Number of Fisher Scoring iterations: 3
```

```
gout.goofy.super <- glm(heart.disease ~ snoring,
    family = binomial(link="identity"))
summary(gout.goofy.super)
```

```
##
## Call:
## glm(formula = heart.disease ~ snoring, family = binomial(link = "identity"))
##
## Coefficients:
##                            Estimate Std. Error z value Pr(>|z|)
## (Intercept)                 0.11811    0.02025   5.832 5.46e-09 ***
## snoringnearly.every.night -0.01952    0.02876  -0.679  0.49739
## snoringnever              -0.10071    0.02055  -4.900 9.61e-07 ***
## snoringoccasionally       -0.06325    0.02217  -2.853  0.00432 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance:  6.5904e+01  on 3  degrees of freedom
## Residual deviance: -5.7732e-14  on 0  degrees of freedom
## AIC: 28.253
##
## Number of Fisher Scoring iterations: 2
```

```
mu.super <- predict(gout.super, type = "response")
mu.goofy.super <- predict(gout.goofy.super, type = "response")
all.equal(mu.super, mu.goofy.super)
```

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

We fit the model with two different link functions, even though they are the same model, as proved by the `all.equal` statement. The reason for this is we are not sure how R function `anova` deals with different link functions. We compare the two models of interest to their least common supermodel as follows.

```
anova(gout, gout.super, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model 1: heart.disease ~ x
## Model 2: heart.disease ~ snoring
##   Resid. Df Resid. Dev Df Deviance Pr(>Chi)
## 1         2     2.8089
## 2         0     0.0000  2   2.8089   0.2455
```

```
anova(gout.goofy, gout.goofy.super, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model 1: heart.disease ~ x
```

```
## Model 2: heart.disease ~ snoring
##   Resid. Df Resid. Dev Df Deviance Pr(>Chi)
## 1         2   0.069191
## 2         0   0.000000  2 0.069191    0.966
```

We see that both models apparently fit these data, but what we are calling the "goofy" model actually fits better.

Section 4.2.5 in Agresti

There are other link functions that can be used.

```
gout.probit <- glm(heart.disease ~ x, family = binomial(link="probit"))
gout.cauchit <- glm(heart.disease ~ x, family = binomial(link="cauchit"))
mu.probit <- predict(gout.probit, type = "response")
mu.cauchit <- predict(gout.cauchit, type = "response")

plot(x, mu, ylab = "probability", pch = 19,
    ylim = range(mu, mu.goofy, mu.probit, mu.cauchit))
curve(predict(gout, newdata = data.frame(x = x), type = "response"),
    add = TRUE)
points(x, mu.goofy, col = "red", pch = 19)
curve(predict(gout.goofy, newdata = data.frame(x = x), type = "response"),
    add = TRUE, col = "red")
points(x, mu.probit, col = "yellow3", pch = 19)
curve(predict(gout.probit, newdata = data.frame(x = x), type = "response"),
    add = TRUE, col = "yellow3")
grep("yellow", colors(), value = TRUE)
```

```
##  [1] "greenyellow"          "lightgoldenrodyellow" "lightyellow"
##  [4] "lightyellow1"         "lightyellow2"         "lightyellow3"
##  [7] "lightyellow4"         "yellow"               "yellow1"
## [10] "yellow2"              "yellow3"              "yellow4"
## [13] "yellowgreen"
```

```
points(x, mu.cauchit, col = "blue", pch = 19)
curve(predict(gout.cauchit, newdata = data.frame(x = x), type = "response"),
    add = TRUE, col = "blue")
```

Your humble instructor recommends logit link for reasons that will become apparent when we discuss exponential families. One really has to like what Agresti calls the "latent tolerance" motivation to use probit or cauchit. And even then, logit link is also a "latent tolerance" model based on the logistic distribution. So it is unclear that there is any reason for avoiding the logit link even if one likes the "latent tolerance" story.

The underlying "latent" distributions (meaning completely unobservable, completely hypothetical, and completely unverifiable) for the available links for R function `binomial`.

```
curve(dcauchy(x, scale = 1 / qcauchy(0.75)), from = -4, to = 4, col = "blue",
    xlab = "latent variable", ylab = "probability density")
curve(dnorm(x, sd = 1 / qnorm(0.75)), add = TRUE, col = "yellow3")
curve(dlogis(x, scale = 1 / qlogis(0.75)), add = TRUE)
```

But there is absolutely no reason to stop at these. There are an infinite variety of such distributions. For example, the t distributions with ν degrees of freedom for $1 < \nu < \infty$ (although only integer degrees of freedom are used in intro stats, the formula for the t PDF makes sense for non-integer degrees of freedom, as explained in theory courses) interpolate between the Cauchy distribution, which is the t distribution with one degree of freedom, and the normal distribution, which is the limit of $t(\nu)$ distributions as $\nu \to \infty$. So

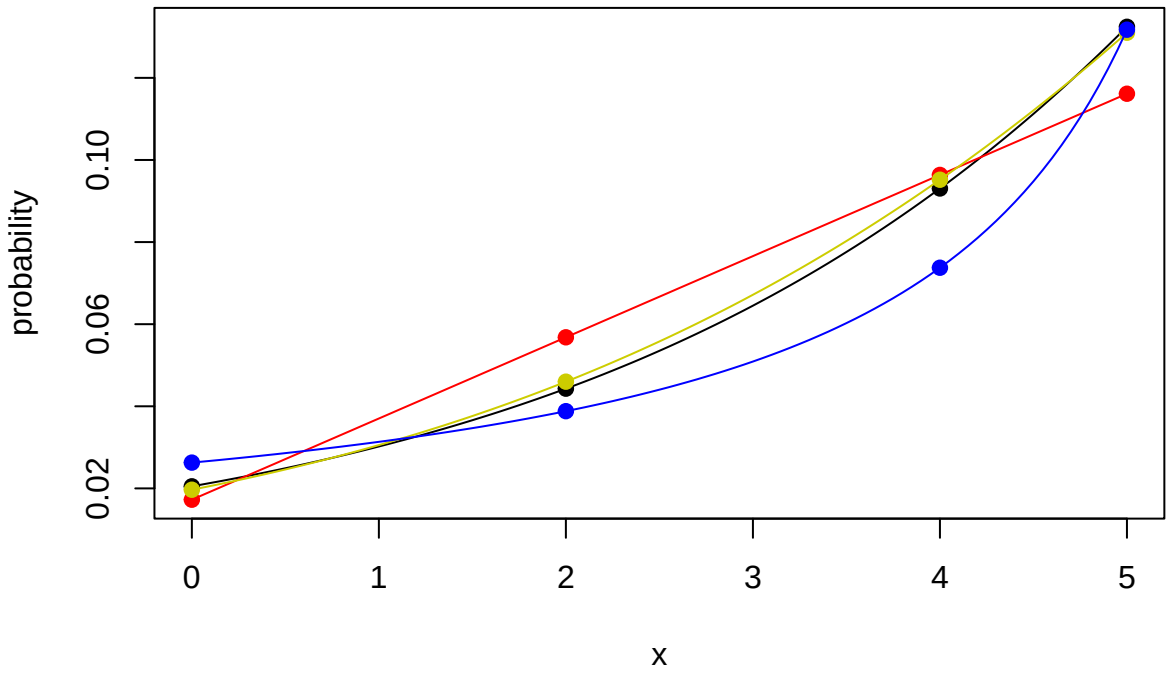


Figure 2: logit link (black) versus identity link (red) versus probit link (yellow) versus cauchit link (blue)

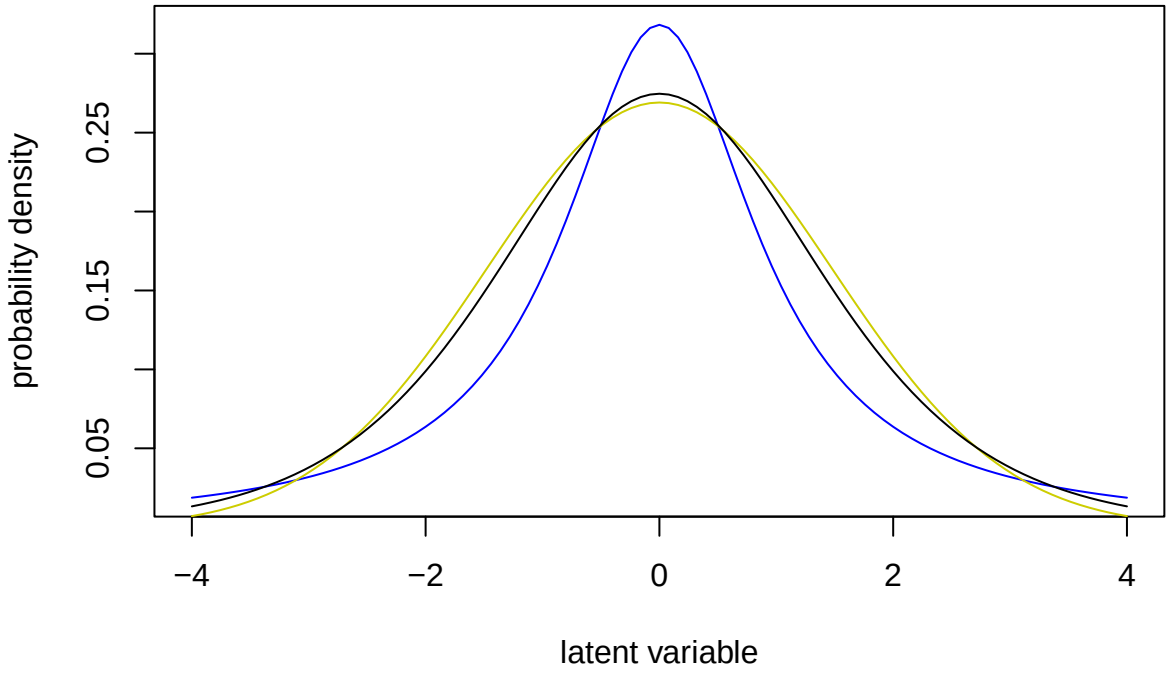


Figure 3: Distributions. Logistic (black), normal (yellow) and Cauchy (blue). Same median and quartiles.

that is already an infinity of possibilities, but there are even more than that. And there is no way to choose a "right" one. So R function `binomial` stops at these three latent distributions.

Confidence Intervals

Wald Intervals

These are produced by R function `predict`. From now on we will only use the fit in R object `gout`, which uses the default logit link and has 2 parameters.

First, the output of summary gives standard errors, so if one wanted confidence intervals for the betas, one would do them as follows.

```
sout <- summary(gout)
class(sout)
```

```
## [1] "summary.glm"
```

```
class(unclass(sout))
```

```
## [1] "list"
```

```
names(unclass(sout))
```

```
##  [1] "call"           "terms"          "family"         "deviance"
##  [5] "aic"            "contrasts"      "df.residual"    "null.deviance"
##  [9] "df.null"        "iter"           "deviance.resid" "coefficients"
## [13] "aliased"        "dispersion"     "df"             "cov.unscaled"
## [17] "cov.scaled"
```

```
cout <- sout$coefficients
class(cout)
```

```
## [1] "matrix" "array"
```

```
cout
```

```
##               Estimate Std. Error    z value      Pr(>|z|)
## (Intercept) -3.8662481 0.16621436 -23.260614 1.110885e-119
## x            0.3973366 0.05001066   7.945039  1.941304e-15
```

```
conf.level <- 0.95
crit <- qnorm((1 + conf.level) / 2)
crit
```

```
## [1] 1.959964
```

```
lower <- cout[ , "Estimate"] - crit * cout[ , "Std. Error"]
upper <- cout[ , "Estimate"] + crit * cout[ , "Std. Error"]
foo <- data.frame(lower, upper)
foo
```

```
##                  lower      upper
## (Intercept) -4.1920223 -3.5404739
## x            0.2993175  0.4953557
```

However, the regression coefficients (betas) are meaningless quantities. They are very far from the data, and the model can be reparameterized without changing the model. Thus we look at confidence intervals for probabilities.

```
pout <- predict(gout, type = "response", se.fit = TRUE)
lower <- pout$fit - crit * pout$se.fit
upper <- pout$fit + crit * pout$se.fit
data.frame(lower, upper)
```

```
##        lower      upper
## 1 0.01396364 0.02705120
## 2 0.03561897 0.05297125
## 3 0.07330823 0.11279999
## 4 0.09798190 0.16689580
```

```
library(sfsmisc)
errbar(x, mu, upper, lower, ylab = "probability")
```

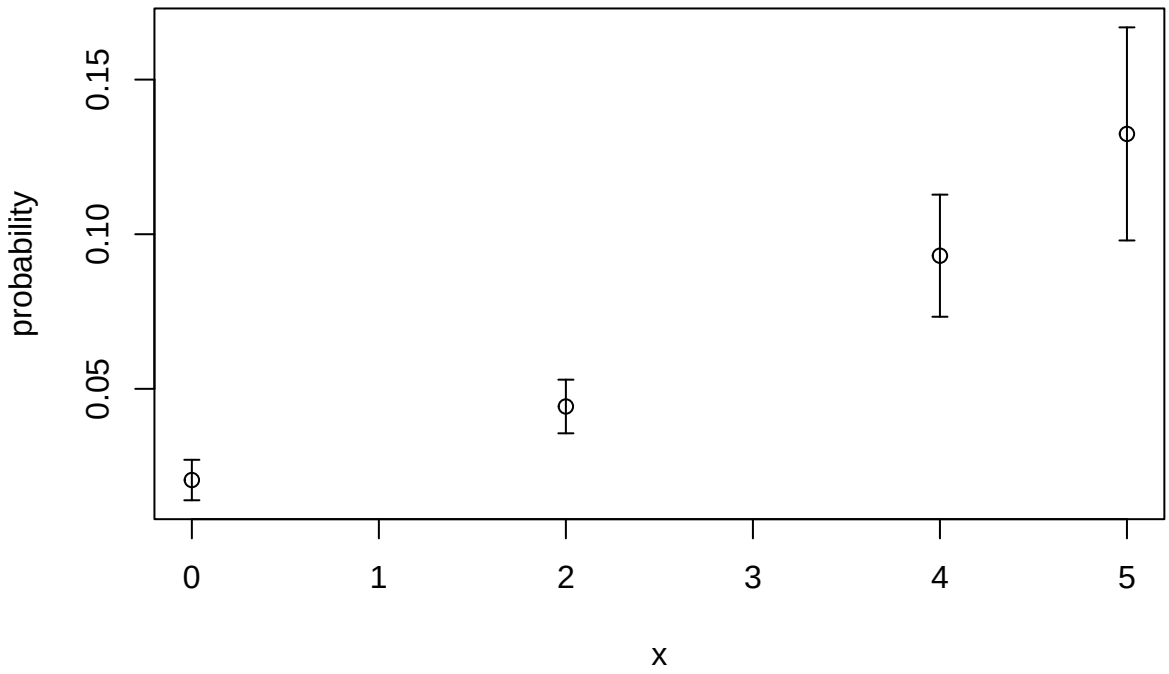


Figure 4: 95% Wald confidence intervals, not simultaneous

These are not, of course, corrected for multiple comparisons.

We can also get Wald intervals at x values other than those in the observed data. It is not clear that these make any sense in these data, but we will illustrate them anyway.

```
plot(x, mu, ylim = range(mu, upper, lower), ylab = "probability")
curve(predict(gout, newdata = data.frame(x = x), type = "response"),
    add = TRUE)
sally <- function(x) {
    pout <- predict(gout, type = "response", se.fit = TRUE,
        newdata = data.frame(x = x))
```

```
    pout$fit - crit * pout$se.fit
}
curve(sally, add = TRUE, lty = "dashed")
sally <- function(x) {
    pout <- predict(gout, type = "response", se.fit = TRUE,
        newdata = data.frame(x = x))
    pout$fit + crit * pout$se.fit
}
curve(sally, add = TRUE, lty = "dashed")
```

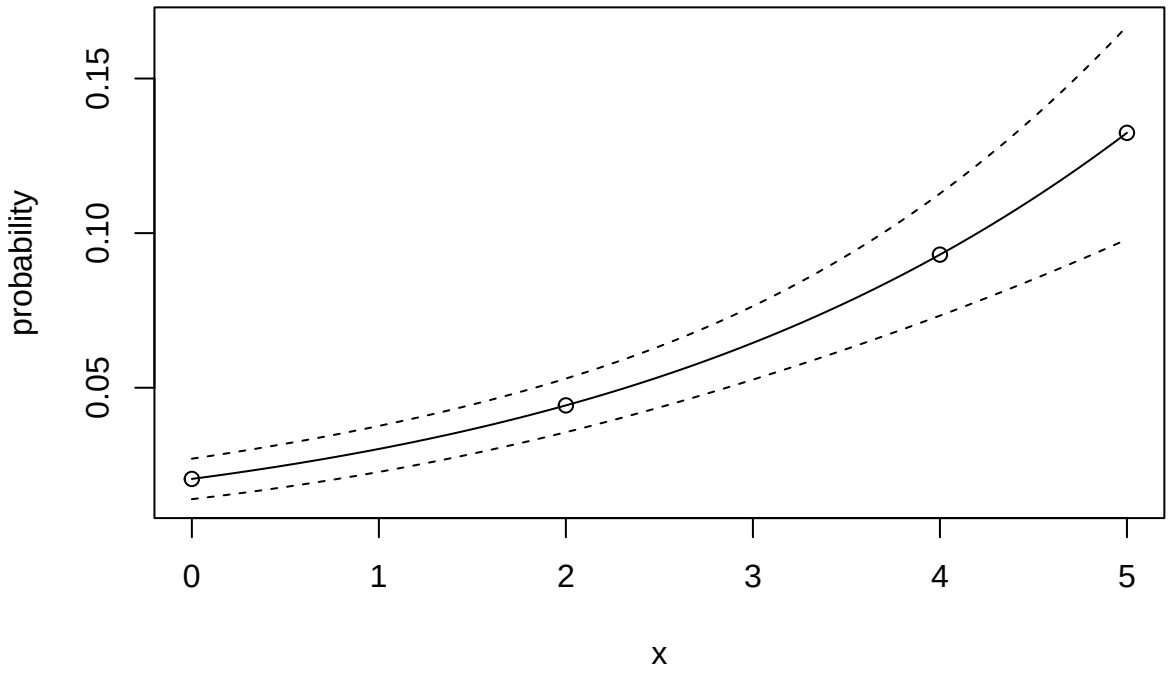


Figure 5: 95% Wald confidence bands, not simultaneous

Likelihood Intervals

R Function Confint

R has a built-in function for doing likelihood intervals

```
library(MASS)
confint(gout)
```

```
## Waiting for profiling to be done...
```

```
##                  2.5 %     97.5 %
## (Intercept) -4.2072190 -3.5544117
## x            0.2999362  0.4963887
```

Unfortunately these intervals are for the meaningless parameters (regression coefficients) rather than the meaningful parameters (success probability as a function of x). To calculate those we need to use the methods of the handout on likelihood computation.

Log Likelihood

But even before we try that, we need the log likelihood function, which we have not had explicitly, because we were using R function `glm`, which knows the log likelihood function but doesn't return it to the user.

For this problem, the log likelihood is

$$\sum_i [x_i \log(p_i) - (n_i - x_i) \log(1 - p_i)]$$

where each row of the data matrix `heart.disease` is $(x_i, n_i - x_i)$ and the p_i are components of a vector $\mathbf{p}$ that satisfies

$$\theta = \mathbf{M}\beta$$

where $\mathbf{M}$ is the model matrix, β is the vector of regression coefficients, and θ is the so-called "linear predictor" which is related to $\mathbf{p}$ by the inverse link function.

$$p_i = \frac{e^{\theta_i}}{1 + e^{\theta_i}} = \frac{1}{1 + e^{-\theta_i}}$$

so

$$1 - p_i = \frac{1}{1 + e^{\theta_i}} = \frac{e^{-\theta_i}}{1 + e^{-\theta_i}}$$

and

$$\log(p_i) = \theta_i - \log(1 + e^{\theta_i}) = -\log(1 + e^{-\theta_i})$$
$$\log(1 - p_i) = -\log(1 + e^{\theta_i}) = -\theta_i - \log(1 + e^{-\theta_i})$$

The reason why we have two expressions for each of these is that e^θ may overflow (be a number too big for the computer to represent) when θ is moderately large

```
exp(800)
```

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

and, of course, $e^{-\theta}$ can overflow when θ is moderately large negative. Thus we want to use whichever formula cannot overflow, the one containing $e^{-\theta}$ when $\theta > 0$ and the one containing e^θ when θ is negative.

Also we never want to calculate $1 - p_i$ by subtraction because that can cause "catastrophic cancellation", loss of all significant figures, when p_i is near zero.

```
1 + 1e-20 - 1
```

```
## [1] 0
```

Hence we need all 4 formulas above.

Finally, to avoid more catastrophic cancellation, we use R function `log1p` which calculates the function $x \mapsto \log(1 + x)$ more accurately when x is small than just using the log function and addition. From calculus $\log(1 + x) \approx x$ when x is near zero.

```
xx <- c(1e-20, 1e-10, 1, 10)
log1p(xx)
```

```
## [1] 1.000000e-20 1.000000e-10 6.931472e-01 2.397895e+00
```

```
log(1 + xx)
```

```
## [1] 0.0000000000 0.0000000001 0.6931471806 2.3978952728
```

Such finicky calculation is also something we do not expect for homework but which is necessary to write good code usable by others. This subject of computer arithmetic is really the subject of another course, so we won't say much about it in this course. Those that are interested can look at my Stat 3701 notes on this subject.

We also need the model matrix M. We could figure it out for ourselves, but we can also get it from the output of R function `glm` if we ask for it.

```
gout <- glm(heart.disease ~ x, family = binomial, x = TRUE)
gout$x
```

```
##   (Intercept) x
## 1           1 0
## 2           1 2
## 3           1 4
## 4           1 5
## attr(,"assign")
## [1] 0 1
```

Now we are ready to code the log likelihood function. We use a factory function, but, of course, the global variables approach can also be used.

```
mlogl.factory <- function(modmat, response) {
    stopifnot(is.numeric(modmat))
    stopifnot(is.finite(modmat))
    stopifnot(is.matrix(modmat))
    stopifnot(is.numeric(response))
    stopifnot(is.finite(response))
    stopifnot(is.matrix(response))
    stopifnot(round(response) == response) # check for integer-valued
    stopifnot(response >= 0)
    stopifnot(ncol(response) == 2)
    stopifnot(nrow(response) == nrow(modmat))
    success <- response[ , 1]
    failure <- response[ , 2]
    function(beta) {
        stopifnot(is.numeric(beta))
        stopifnot(is.finite(beta))
        stopifnot(length(beta) == ncol(modmat))
        theta <- as.vector(modmat %*% beta)
        logp <- ifelse(theta > 0, - log1p(exp(- theta)),
            theta - log1p(exp(theta)))
        logq <- ifelse(theta > 0, - theta - log1p(exp(- theta)),
            - log1p(exp(theta) ))
        - sum(success * logp + failure * logq)
    }
}
```

We should have said when we introduced the function factory approach the reasons why it is better than the global variables approach are

- global variables are evil — they are easily clobbered accidentally before they are used, or they can clobber other variables of the same name being used by the user for other purposes and
- the factory function can check the validity of what would be the global variables in the other approach — and this check is done once at the time the `mlogl` function is created rather than every time the `mlogl` function is invoked (as would have to be done in the global variables approach where there is

no factory function to do this checking)

```
mlogl <- mlogl.factory(gout$x, heart.disease)
```

Let us check that our function works and gives the same MLE as R function `glm`

```
nout <- nlm(mlogl, c(0, 0))
nout$code <= 2
```

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

```
all.equal(nout$estimate, gout$coefficients, check.attributes = FALSE)
```

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

Good! The reason why we did not bother to find a "good" starting point for the optimization is that we know this log likelihood function has a unique local maximum, which is also the global maximum, that is found from any starting point. But the explanation for this will have to wait until we study exponential families of distributions.

Now we also need the function that maps β to $\mathbf{p}$.

```
pooh.factory <- function(x) {
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(length(x) == 1)
    function(beta) {
        stopifnot(is.numeric(beta))
        stopifnot(is.finite(beta))
        stopifnot(length(beta) == 2)
        theta <- sum(c(1, x) * beta)
        1 / (1 + exp(- theta))
    }
}
```

Try it out.

```
pooh <- pooh.factory(0)
pooh(nout$estimate)
```

```
## [1] 0.02050742
```

```
mu[1]
```

```
##          1
## 0.02050742
```

Now we are finally ready to calculate the likelihood intervals.

```
library(alabama)
```

```
## Loading required package: numDeriv
```

```
lower <- double(length(x))
for (i in seq(along = x)) {
    aout <- suppressWarnings(auglag(nout$estimate,
        fn = pooh.factory(x[i]),
        hin = function(theta) nout$minimum + crit - mlogl(theta),
        control.outer = list(trace = FALSE)))
    stopifnot(aout$convergence == 0)
    lower[i] <- aout$value
```

```
}
upper <- double(length(x))
for (i in seq(along = x)) {
    aout <- suppressWarnings(auglag(nout$estimate,
        fn = pooh.factory(x[i]),
        hin = function(theta) nout$minimum + crit - mlogl(theta),
        control.outer = list(trace = FALSE),
        control.optim = list(fnscale = -1)))
    stopifnot(aout$convergence == 0)
    upper[i] <- aout$value
}
data.frame(lower, upper)
```

```
##        lower      upper
## 1 0.01461808 0.02784713
## 2 0.03783432 0.05225022
## 3 0.07436824 0.11426074
## 4 0.10038659 0.16994416
```

```
errbar(x, mu, upper, lower, ylab = "probability")
```

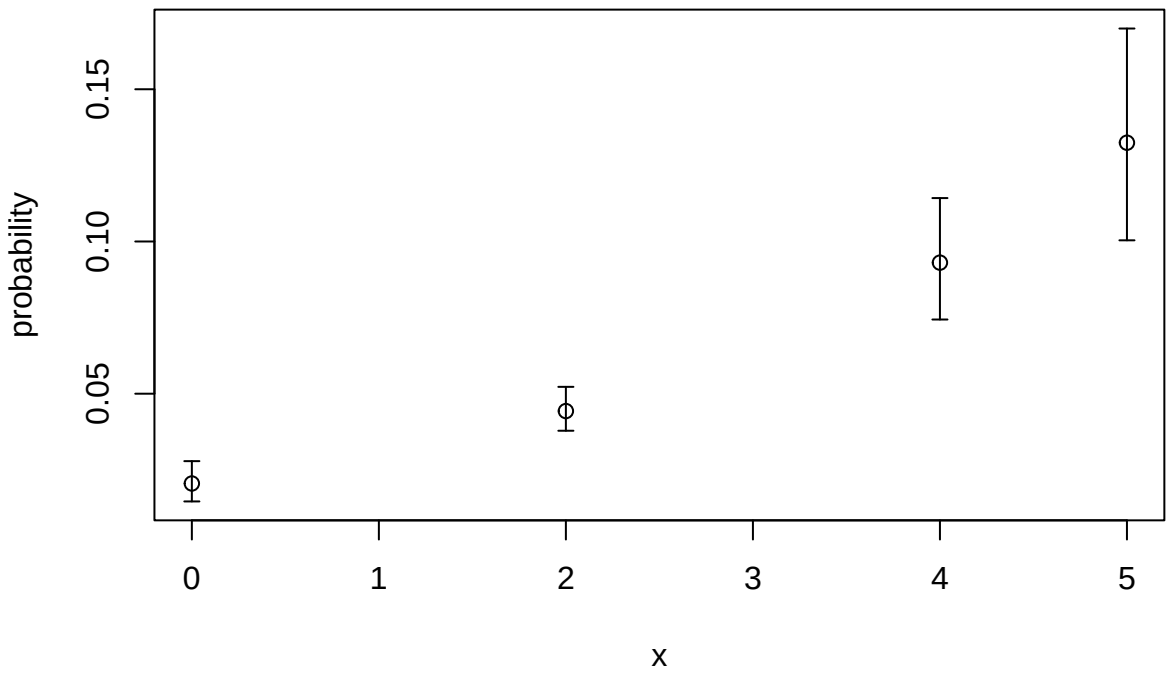


Figure 6: 95% likelihood confidence intervals, not simultaneous

Of course, if we want simultaneous coverage, we can just use the appropriate critical value for that, degrees of freedom equal to the length of `beta`, here 2, as discussed in the section on simultaneous coverage of the lecture notes on likelihood computation

Section 4.3 in Agresti

A Poisson GLM Example

```
# clean out R global environment and start fresh
rm(list = ls())
```

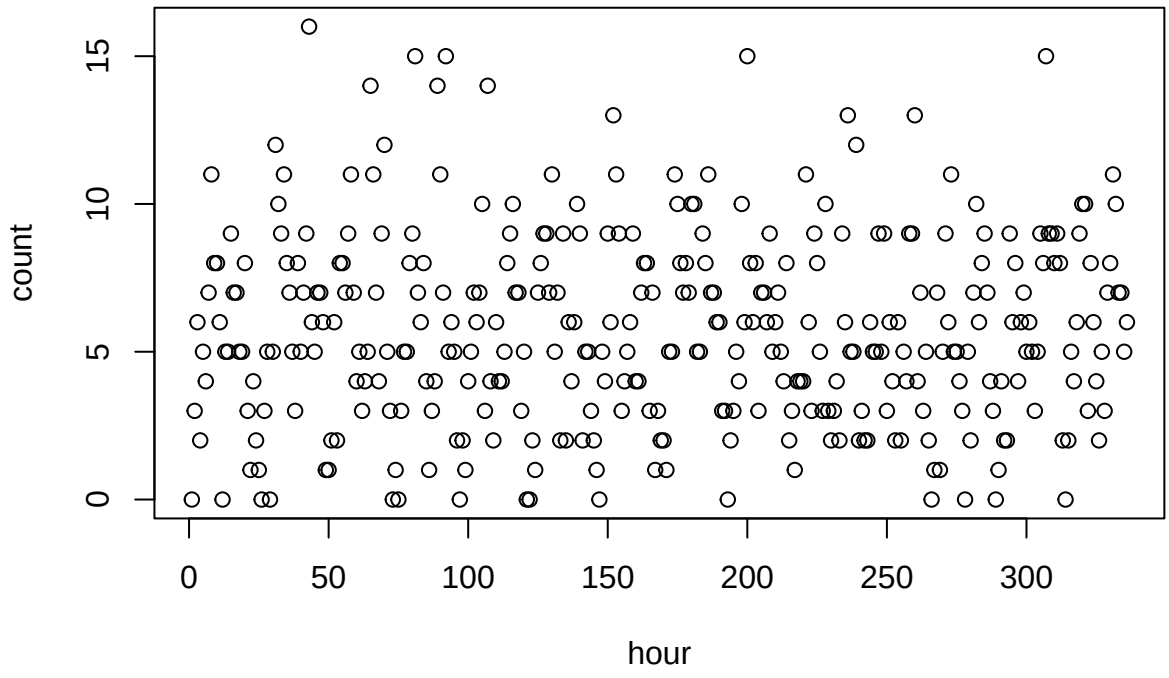
Data

There are also GLM for Poisson response. For an example, we will use some data analyzed in my 5102 lecture notes

```
foo <- read.table("http://www.stat.umn.edu/geyer/5102/data/ex6-4.txt",
    header = TRUE)
names(foo)
```

```
## [1] "hour"  "count"
```

```
hour <- foo$hour
count <- foo$count
plot(hour, count)
```

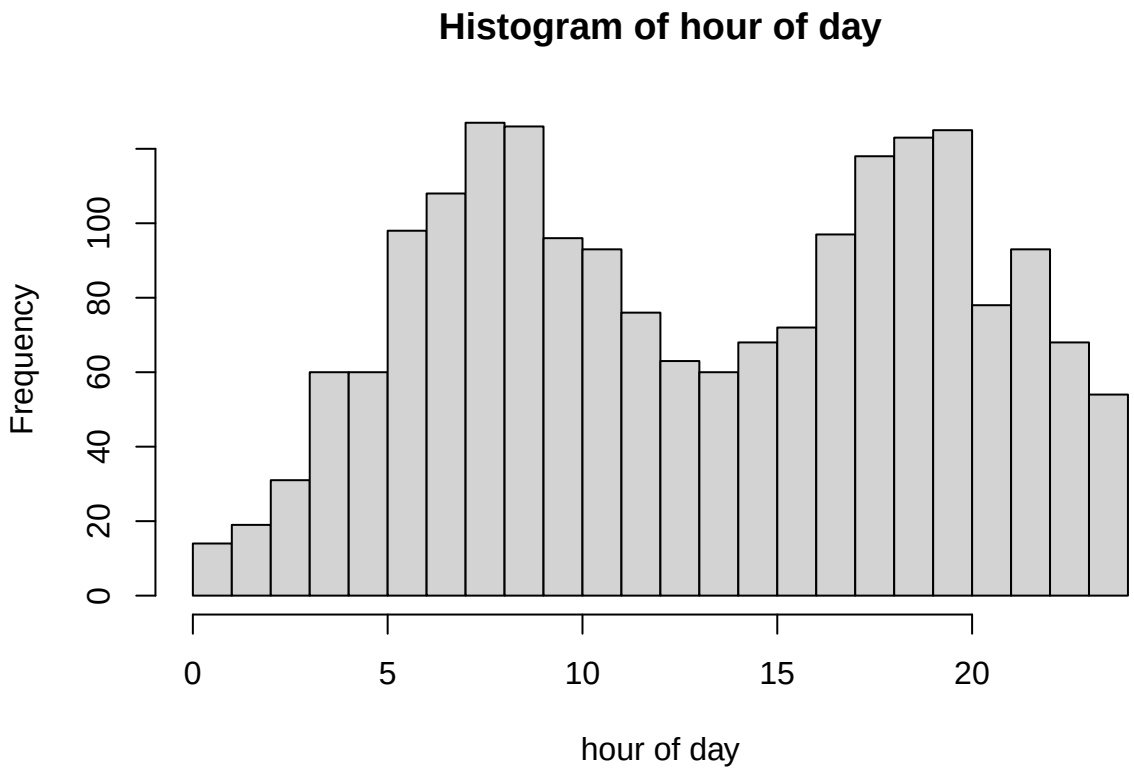


These (simulated) data are hourly counts from a not necessarily homogeneous Poisson process. The variables are

- `hour` which counts hours sequentially throughout a 14-day period (running from 1 to $14 \times 24 = 336$) and
- `count` giving the count for each of these hours.

The idea of the regression is to estimate the mean as a function of time. Many time series have a daily cycle. If we pool the counts for the same hour of the day over the 14 days of the series, we see a clear pattern in the histogram.

```
hourofday <- (hour - 1) %% 24 + 1
fred <- structure(list(breaks = 0:24,
    counts = sapply(split(count, hourofday), sum),
    mids = 1:24 - 1 / 2, xname = "hour of day",
    equidist = TRUE), class = "histogram")
plot(fred)
```



The somewhat mysterious code above makes an R object of class `"histogram"` and hands it off to R generic function `plot` which has a method for objects of this class. The requirements for such objects are documented in the help page for R function `histogram`.

First GLM

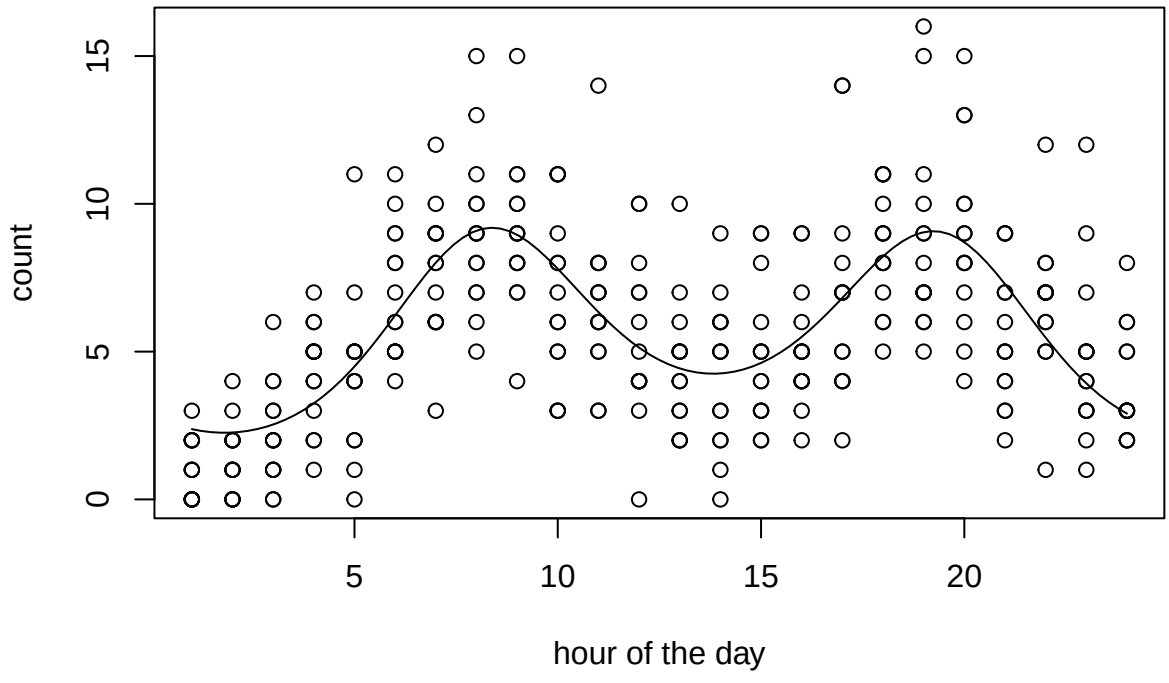
Since there seems to be a daily cycle with two peaks we make the "linear predictor" for our GLM a Fourier series with frequencies one per day and two per day.

```
w <- hour / 24 * 2 * pi
out <- glm(count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) +
    I(cos(2 * w)), family = poisson)
summary(out)
```

```
##
## Call:
## glm(formula = count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) +
```

```
##     I(cos(2 * w)), family = poisson)
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)    1.65917    0.02494  66.516  < 2e-16 ***
## I(sin(w))     -0.13916    0.03128  -4.448 8.66e-06 ***
## I(cos(w))     -0.28510    0.03661  -7.787 6.86e-15 ***
## I(sin(2 * w)) -0.42974    0.03385 -12.696  < 2e-16 ***
## I(cos(2 * w)) -0.30846    0.03346  -9.219  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 704.27  on 335  degrees of freedom
## Residual deviance: 399.58  on 331  degrees of freedom
## AIC: 1535.7
##
## Number of Fisher Scoring iterations: 5
```

```r
plot(hourofday, count, xlab = "hour of the day")
curve(predict(out, data.frame(w = x / 24 * 2 * pi),
    type="response"), add=TRUE)
```



Not bad. We seem to have a reasonable regression function for these data.

Hypothesis Tests

But maybe more terms in our Fourier series would be better?

```
form <- "count ~ I(sin(w)) + I(cos(w))"
for (i in 2:10) {
    form <- c(form, paste0(form[length(form)],
        " + I(sin(", i, " * w)) + I(cos(", i, " * w))"))
}
outs <- list()
for (i in seq(along = form))
    outs[[i]] <- glm(as.formula(form[i]), family = poisson)
class(outs) <- "glmlist"
anova(outs, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model  1: count ~ I(sin(w)) + I(cos(w))
## Model  2: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w))
## Model  3: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w))
## Model  4: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w))
## Model  5: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w)) + I(sin(5 * w)) + I(cos(5 * w))
## Model  6: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w)) + I(sin(5 * w)) + I(cos(5 * w)) + I(sin(6 * w)) + I(cos(6 *
##     w))
## Model  7: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w)) + I(sin(5 * w)) + I(cos(5 * w)) + I(sin(6 * w)) + I(cos(6 *
##     w)) + I(sin(7 * w)) + I(cos(7 * w))
## Model  8: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w)) + I(sin(5 * w)) + I(cos(5 * w)) + I(sin(6 * w)) + I(cos(6 *
##     w)) + I(sin(7 * w)) + I(cos(7 * w)) + I(sin(8 * w)) + I(cos(8 *
##     w))
## Model  9: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w)) + I(sin(5 * w)) + I(cos(5 * w)) + I(sin(6 * w)) + I(cos(6 *
##     w)) + I(sin(7 * w)) + I(cos(7 * w)) + I(sin(8 * w)) + I(cos(8 *
##     w)) + I(sin(9 * w)) + I(cos(9 * w))
## Model 10: count ~ I(sin(w)) + I(cos(w)) + I(sin(2 * w)) + I(cos(2 * w)) +
##     I(sin(3 * w)) + I(cos(3 * w)) + I(sin(4 * w)) + I(cos(4 *
##     w)) + I(sin(5 * w)) + I(cos(5 * w)) + I(sin(6 * w)) + I(cos(6 *
##     w)) + I(sin(7 * w)) + I(cos(7 * w)) + I(sin(8 * w)) + I(cos(8 *
##     w)) + I(sin(9 * w)) + I(cos(9 * w)) + I(sin(10 * w)) + I(cos(10 *
##     w))
##    Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
## 1        333     651.10
## 2        331     399.58  2  251.523 < 2.2e-16 ***
## 3        329     396.03  2    3.546  0.169802
```

```
## 4        327     388.16  2    7.870  0.019547 *
## 5        325     377.96  2   10.208  0.006072 **
## 6        323     369.95  2    8.005  0.018269 *
## 7        321     368.57  2    1.383  0.500901
## 8        319     366.65  2    1.917  0.383492
## 9        317     363.37  2    3.279  0.194049
## 10       315     361.98  2    1.393  0.498338
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The big improvement is going from frequency 1 per day to 2 per day, but smaller improvements occur up to frequency 6 per day.

Alternative methods of model selection that avoid formal hypothesis tests use the information criteria AIC and BIC.

```
out.aic <- sapply(outs, AIC)
out.aic
```

```
##  [1] 1783.204 1535.680 1536.134 1532.264 1526.056 1522.051 1524.668 1526.751
##  [9] 1527.472 1530.079
```

```
which(out.aic == min(out.aic))
```

```
## [1] 6
```

```
out.bic <- sapply(outs, BIC)
out.bic
```

```
##  [1] 1794.655 1554.766 1562.854 1566.618 1568.044 1571.673 1581.925 1591.642
##  [9] 1599.997 1610.238
```

```
which(out.bic == min(out.bic))
```

```
## [1] 2
```

- BIC (Bayesian Information Criterion) is for when one believes that the true unknown probability model is in one of the statistical models considered — in this example that the regression function mapped to the linear predictor scale (by taking logs) has the form of a trigonometric series.
- AIC (Akaike Information Criterion) is for when one does not believe the true unknown probability model is in one of the statistical models considered.

Confidence Intervals

Confidence intervals are very like those done above for binomial response. The only difference is that one uses the log likelihood for Poisson response rather than binomial response.

We will just show some Wald confidence bands.

```
conf.level <- 0.95
crit <- qnorm((1 + conf.level) / 2)
plot(hourofday, count, xlab = "hour of the day")
curve(predict(out, newdata = data.frame(w = x / 24 * 2 * pi),
    type="response"), add=TRUE)
sally <- function(x) {
    pout <- predict(out, type = "response", se.fit = TRUE,
        newdata = data.frame(w = x / 24 * 2 * pi))
    pout$fit - crit * pout$se.fit
}
curve(sally, add = TRUE, lty = "dashed")
```

```
sally <- function(x) {
    pout <- predict(out, type = "response", se.fit = TRUE,
        newdata = data.frame(w = x / 24 * 2 * pi))
    pout$fit + crit * pout$se.fit
}
curve(sally, add = TRUE, lty = "dashed")
```

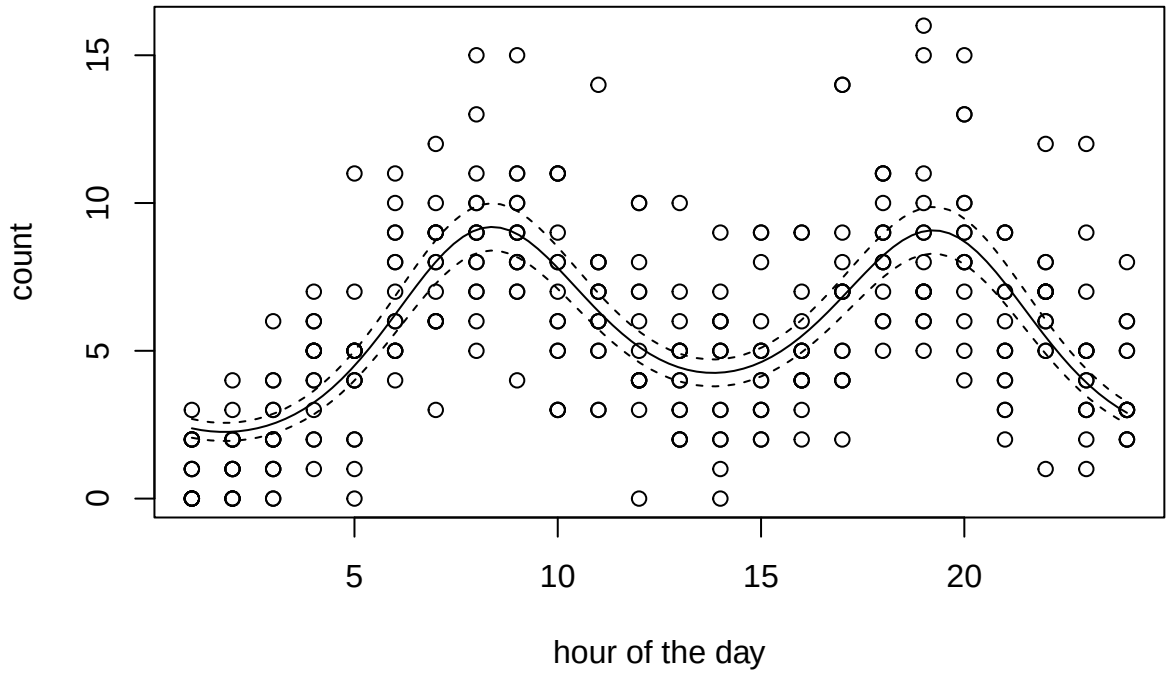


Figure 7: 95% Wald confidence bands, not simultaneous

Intercept Makes No Difference When Any Predictor Is Categorical

This is just a fact about how R formulas work. It is another aspect of canonical parameters are meaningless.

We use for an example data from the examples for R function `glm`

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
data.frame(treatment, outcome, counts) # showing data
```

```
##   treatment outcome counts
## 1         1       1     18
## 2         1       2     17
## 3         1       3     15
```

```
## 4         2       1     20
## 5         2       2     10
## 6         2       3     20
## 7         3       1     25
## 8         3       2     13
## 9         3       3     12
```

```
glm.D93 <- glm(counts ~ outcome + treatment, family = poisson())
summary(glm.D93)
```

```
##
## Call:
## glm(formula = counts ~ outcome + treatment, family = poisson())
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  3.045e+00  1.709e-01  17.815   <2e-16 ***
## outcome2    -4.543e-01  2.022e-01  -2.247   0.0246 *
## outcome3    -2.930e-01  1.927e-01  -1.520   0.1285
## treatment2   1.217e-15  2.000e-01   0.000   1.0000
## treatment3   8.438e-16  2.000e-01   0.000   1.0000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 10.5814  on 8  degrees of freedom
## Residual deviance:  5.1291  on 4  degrees of freedom
## AIC: 56.761
##
## Number of Fisher Scoring iterations: 4
```

Note that `outcome` and `treatment` are factors (the terminology R uses for categorical variables)

```
class(outcome)
```

```
## [1] "factor"
```

```
class(treatment)
```

```
## [1] "factor"
```

and they each have three levels

```
nlevels(outcome)
```

```
## [1] 3
```

```
nlevels(treatment)
```

```
## [1] 3
```

So how does a categorical variable correspond to a regression model in which it is used? For each categorical variable, the regression model needs *dummy variables*, one for each category (R calls them *levels* of the *factor* variable). We can see these dummy variables as follows

```
model.matrix(~ 0 + outcome)
```

```
##   outcome1 outcome2 outcome3
## 1        1        0        0
```

```
## 2         0         1         0
## 3         0         0         1
## 4         1         0         0
## 5         0         1         0
## 6         0         0         1
## 7         1         0         0
## 8         0         1         0
## 9         0         0         1
## attr(,"assign")
## [1] 1 1 1
## attr(,"contrasts")
## attr(,"contrasts")$outcome
## [1] "contr.treatment"
```

```
model.matrix(~ 0 + treatment)
```

```
##   treatment1 treatment2 treatment3
## 1          1          0          0
## 2          1          0          0
## 3          1          0          0
## 4          0          1          0
## 5          0          1          0
## 6          0          1          0
## 7          0          0          1
## 8          0          0          1
## 9          0          0          1
## attr(,"assign")
## [1] 1 1 1
## attr(,"contrasts")
## attr(,"contrasts")$treatment
## [1] "contr.treatment"
```

Each dummy variable indicates the cases (components of the response vector, rows of the model matrix) which are in that category (level).

Note that the dummy variables corresponding to a categorical variable add up to the vector all of whose components are equal to one, which is what R calls the *intercept* dummy variable.

```
rowSums(model.matrix(~ 0 + outcome))
```

```
## 1 2 3 4 5 6 7 8 9
## 1 1 1 1 1 1 1 1 1
```

```
rowSums(model.matrix(~ 0 + treatment))
```

```
## 1 2 3 4 5 6 7 8 9
## 1 1 1 1 1 1 1 1 1
```

```
model.matrix(counts ~ 1)
```

```
##   (Intercept)
## 1           1
## 2           1
## 3           1
## 4           1
## 5           1
## 6           1
## 7           1
```

```
## 8           1
## 9           1
## attr(,"assign")
## [1] 0
```

Thus we would not have an identifiable model if we kept all of the dummy variables corresponding to a categorical variable and kept an "intercept" dummy variable. The default behavior in R is to

- keep an "intercept" dummy variable and
- drop one of the dummy variables for each categorical variable (factor).

This gives an identifiable model.

```
model.matrix(counts ~ outcome + treatment)
```

```
##   (Intercept) outcome2 outcome3 treatment2 treatment3
## 1           1        0        0          0          0
## 2           1        1        0          0          0
## 3           1        0        1          0          0
## 4           1        0        0          1          0
## 5           1        1        0          1          0
## 6           1        0        1          1          0
## 7           1        0        0          0          1
## 8           1        1        0          0          1
## 9           1        0        1          0          1
## attr(,"assign")
## [1] 0 1 1 2 2
## attr(,"contrasts")
## attr(,"contrasts")$outcome
## [1] "contr.treatment"
##
## attr(,"contrasts")$treatment
## [1] "contr.treatment"
```

The dummy variables `outcome1` and `treatment1` are omitted.

But we can tell R to omit the intercept.

```
model.matrix(counts ~ 0 + outcome + treatment)
```

```
##   outcome1 outcome2 outcome3 treatment2 treatment3
## 1        1        0        0          0          0
## 2        0        1        0          0          0
## 3        0        0        1          0          0
## 4        1        0        0          1          0
## 5        0        1        0          1          0
## 6        0        0        1          1          0
## 7        1        0        0          0          1
## 8        0        1        0          0          1
## 9        0        0        1          0          1
## attr(,"assign")
## [1] 1 1 1 2 2
## attr(,"contrasts")
## attr(,"contrasts")$outcome
## [1] "contr.treatment"
##
## attr(,"contrasts")$treatment
## [1] "contr.treatment"
```

Now the behavior is

- omit an "intercept" dummy variable and
- drop one of the dummy variables for each categorical variable (factor) **except for one** categorical variable (for which we keep all of its dummy variables)

Note — this is very important — that these two recipes give the *same* model.

- Leaving out the intercept produces a *different* model *when all of the predictor variables are quantitative.*
- Leaving out the intercept produces the *same* model *when at least one of the predictor variables is categorical (qualitative, factor).*

```
glm.D93.no.intercept <- glm(counts ~ 0 + outcome + treatment,
    family = poisson())
summary(glm.D93.no.intercept)
```

```
##
## Call:
## glm(formula = counts ~ 0 + outcome + treatment, family = poisson())
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## outcome1   3.045e+00  1.709e-01   17.82   <2e-16 ***
## outcome2   2.590e+00  1.958e-01   13.23   <2e-16 ***
## outcome3   2.752e+00  1.860e-01   14.79   <2e-16 ***
## treatment2 6.476e-16  2.000e-01    0.00        1
## treatment3 2.665e-16  2.000e-01    0.00        1
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 572.6047  on 9  degrees of freedom
## Residual deviance:   5.1291  on 4  degrees of freedom
## AIC: 56.761
##
## Number of Fisher Scoring iterations: 4
```

Note that the regression coefficients (estimates) are different from the ones with intercept shown above. But the estimated cell means are the same, hence the *both specify the same statistical model.* These are different parameterizations of the same model.

```
mu <- predict(glm.D93, type = "response")
mu.no.intercept <- predict(glm.D93.no.intercept, type = "response")
all.equal(mu, mu.no.intercept)
```

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

R Function `drop1`

R function `drop1` is useful for finding out the effect of dropping terms from a model, especially when there are categorical variables, in which case one wants to drop all of the dummy variables for a categorical variable or drop none of them.

```
drop1(glm.D93, test = "LRT")
```

```
## Single term deletions
```

```
##
## Model:
## counts ~ outcome + treatment
##           Df Deviance    AIC    LRT Pr(>Chi)
## <none>         5.1291 56.761
## outcome    2  10.5814 58.214 5.4523  0.06547 .
## treatment  2   5.1291 52.761 0.0000  1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

R Function add1

There is also an R function `add1`

```
glm.D93.intercept.only <- glm(counts ~ 1, family = poisson())
add1(glm.D93.intercept.only, scope = ~ outcome + treatment, test = "LRT")
```

```
## Single term additions
##
## Model:
## counts ~ 1
##           Df Deviance    AIC    LRT Pr(>Chi)
## <none>        10.5814 54.214
## outcome    2   5.1291 52.761 5.4523  0.06547 .
## treatment  2  10.5814 58.214 0.0000  1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

R Function anova

But one can also do these tests using R function `anova`

```
glm.D93.outcome.only <- glm(counts ~ outcome, family = poisson())
glm.D93.treatment.only <- glm(counts ~ treatment, family = poisson())

# compare with results of drop1
anova(glm.D93.outcome.only, glm.D93, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model 1: counts ~ outcome
## Model 2: counts ~ outcome + treatment
##   Resid. Df Resid. Dev Df   Deviance Pr(>Chi)
## 1         6     5.1291
## 2         4     5.1291  2 6.2172e-15        1
```

```
anova(glm.D93.treatment.only, glm.D93, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model 1: counts ~ treatment
## Model 2: counts ~ outcome + treatment
##   Resid. Df Resid. Dev Df Deviance Pr(>Chi)
## 1         6    10.5814
## 2         4     5.1291  2   5.4523  0.06547 .
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# compare with results of add1
anova(glm.D93.intercept.only, glm.D93.outcome.only, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model 1: counts ~ 1
## Model 2: counts ~ outcome
##   Resid. Df Resid. Dev Df Deviance Pr(>Chi)
## 1         8    10.5814
## 2         6     5.1291  2   5.4523  0.06547 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
anova(glm.D93.intercept.only, glm.D93.treatment.only, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model 1: counts ~ 1
## Model 2: counts ~ treatment
##   Resid. Df Resid. Dev Df   Deviance Pr(>Chi)
## 1         8     10.581
## 2         6     10.581  2 3.5527e-15        1
```

The Bozo Method of Model Selection

This is not the time to discuss model selection. We have notes on that but are not ready for that yet.

This section is just a warning about a particularly awful method of model selection that is invented over and over again by naive users. I call it the Bozo method of model selection after a particularly losing clown. The word bozotic is a technical term of geek speak.

The Bozo method is to look at some regression output, for example

```
summary(outs[[10]])
```

```
##
## Call:
## glm(formula = as.formula(form[i]), family = poisson)
##
## Coefficients:
##                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)     1.630552   0.026616  61.262  < 2e-16 ***
## I(sin(w))      -0.168785   0.033345  -5.062 4.15e-07 ***
## I(cos(w))      -0.344647   0.041442  -8.316  < 2e-16 ***
## I(sin(2 * w))  -0.491215   0.038528 -12.750  < 2e-16 ***
## I(cos(2 * w))  -0.349382   0.036558  -9.557  < 2e-16 ***
## I(sin(3 * w))  -0.133101   0.039556  -3.365 0.000766 ***
## I(cos(3 * w))  -0.015095   0.035291  -0.428 0.668851
## I(sin(4 * w))  -0.136768   0.038876  -3.518 0.000435 ***
## I(cos(4 * w))   0.061855   0.035869   1.724 0.084626 .
## I(sin(5 * w))  -0.132380   0.037341  -3.545 0.000392 ***
## I(cos(5 * w))   0.073011   0.037152   1.965 0.049391 *
## I(sin(6 * w))  -0.086515   0.036597  -2.364 0.018080 *
## I(cos(6 * w))   0.089696   0.037461   2.394 0.016650 *
```

```
## I(sin(7 * w))  -0.012420   0.036204  -0.343 0.731548
## I(cos(7 * w))   0.055176   0.037257   1.481 0.138614
## I(sin(8 * w))   0.006671   0.037001   0.180 0.856931
## I(cos(8 * w))   0.052532   0.036047   1.457 0.145030
## I(sin(9 * w))   0.069486   0.035535   1.955 0.050534 .
## I(cos(9 * w))  -0.018666   0.034253  -0.545 0.585794
## I(sin(10 * w))  0.038381   0.034465   1.114 0.265438
## I(cos(10 * w)) -0.014599   0.034080  -0.428 0.668374
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 704.27  on 335  degrees of freedom
## Residual deviance: 361.98  on 315  degrees of freedom
## AIC: 1530.1
##
## Number of Fisher Scoring iterations: 5
```

and keep the regressors that have stars and drop the regressors that don't.

There are many things wrong with this method.

- It ignores the structure of the problem. In this problem it makes no sense to keep $\sin(kw)$ and drop $\cos(kw)$ for the same k or vice versa. That's not how Fourier series work. In other problems, it makes no sense to drop some of the regressors that come from one categorical variable (`factor` in R speak) and keep others. The whole point of R functions `drop1` and `add1` is to not do that.
- It is, in effect, multiple testing without correction. Hypothesis tests only make sense when you either do only one test or make some sort of correction for multiple testing.
- There are many methods of model selection that have been put in the statistics literature. This is not one of them. No authority has ever recommended the Bozo method. We will cover some valid methods but not yet.

What R calls "signif stars" are so misleading that I sometimes recommend putting the command

```
options(show.signif.stars = FALSE)
```

In your Rprofile so you never see them again.

```
summary(outs[[10]])
```

```
##
## Call:
## glm(formula = as.formula(form[i]), family = poisson)
##
## Coefficients:
##                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)     1.630552   0.026616  61.262  < 2e-16
## I(sin(w))      -0.168785   0.033345  -5.062 4.15e-07
## I(cos(w))      -0.344647   0.041442  -8.316  < 2e-16
## I(sin(2 * w))  -0.491215   0.038528 -12.750  < 2e-16
## I(cos(2 * w))  -0.349382   0.036558  -9.557  < 2e-16
## I(sin(3 * w))  -0.133101   0.039556  -3.365 0.000766
## I(cos(3 * w))  -0.015095   0.035291  -0.428 0.668851
## I(sin(4 * w))  -0.136768   0.038876  -3.518 0.000435
## I(cos(4 * w))   0.061855   0.035869   1.724 0.084626
```

```
## I(sin(5 * w))  -0.132380   0.037341  -3.545 0.000392
## I(cos(5 * w))   0.073011   0.037152   1.965 0.049391
## I(sin(6 * w))  -0.086515   0.036597  -2.364 0.018080
## I(cos(6 * w))   0.089696   0.037461   2.394 0.016650
## I(sin(7 * w))  -0.012420   0.036204  -0.343 0.731548
## I(cos(7 * w))   0.055176   0.037257   1.481 0.138614
## I(sin(8 * w))   0.006671   0.037001   0.180 0.856931
## I(cos(8 * w))   0.052532   0.036047   1.457 0.145030
## I(sin(9 * w))   0.069486   0.035535   1.955 0.050534
## I(cos(9 * w))  -0.018666   0.034253  -0.545 0.585794
## I(sin(10 * w))  0.038381   0.034465   1.114 0.265438
## I(cos(10 * w)) -0.014599   0.034080  -0.428 0.668374
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 704.27  on 335  degrees of freedom
## Residual deviance: 361.98  on 315  degrees of freedom
## AIC: 1530.1
##
## Number of Fisher Scoring iterations: 5
```

The P-values output by R function `summary` are only valid if corrected for multiple testing (by Bonferroni correction, for example). Or if you are only interested in one of them and are ignoring the rest.

The Log Likelihood for Poisson Regression

For homework problem 3-2 we need the log likelihood for Poisson regression (with default link). This does not seem to be in the notes we have seen so far. It is in the notes on exponential families, sections on the Poisson distribution and canonical affine submodels.

The Poisson distribution has log likelihood for the usual parameter μ

$$l(\mu) = y \log(\mu) - \mu$$

but R function `glm` and other GLM software use the parameter

$$\theta = \log(\mu)$$

(or some other parameter if one does not use the default link function).

Solving for μ gives

$$\mu = e^\theta$$

So the log likelihood for θ is

$$l(\theta) = y\theta - e^\theta$$

Now we want to have a contingency table with more than one cell, so y, μ, and θ all become vectors, and the log likelihood for θ becomes

$$l(\theta) = \sum_{i \in I} \left[y_i \theta_i - e^{\theta_i} \right]$$

The reason for the sum here is that the joint distribution is the product of the marginal distributions and the log of a product is the sum of the logs.

Now we come to the linear model part of GLM (generalized linear model). The parameter vector θ is defined in terms of the parameter vector β (what R calls the "coefficients") by

$$\theta = M\beta$$

where M is the model matrix.