GLM notes

Exponential family & GLMs

Linear regression can be summarised in one equation: $EY|\,X \sim N(X\beta, \sigma^2 I)$. It assumes that the conditional mean $EY|\,X$ is IID normal, can be estimated using a linear prediction $X\beta$, and has constant variance σ^2. This course relaxes the normality assumption, allowing us to model a wider variety of processes. The analysis of hierarchical & other dependent data course relaxes the IID assumption.

The normal distribution is a member of the *exponential family*. The p parameter exponential family has a density $f(x)$ of the form

$$\ln f_\theta(x) = \sum_{i=1}^{p} \eta_i(\theta) T_i(x) - A(\theta) + c(x)$$

GLMs replace the assumption that the conditional mean is normally distributed with the assumption that the conditional mean is an exponential family distribution. For most applications it's better to work with the *exponential dispersion model* instead, which has a simpler form

$$\ln f(x) = \frac{x\theta - A(\theta)}{\phi} + c(x, \phi)$$

where θ, ϕ are scalar parameters and $\phi > 0$. Most of the common distributions are in the exponential family, here's some examples (ignore the g column for now, that's explained in the next few paragraphs:

distribution	density	θ	$A(\theta)$	ϕ	g
Normal	$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$	μ	$\mu^2/2$	σ^2	identity
Binomial	$\binom{n}{x} p^x (1-p)^{n-x}$	$\ln\left(\frac{p}{1-p}\right)$	$n \ln(1+e^\theta)$	1	logit
Poisson	$e^{-\lambda}\lambda^x / x!$	$\ln \lambda$	e^θ	1	log
Negative binomial	$\binom{x+r-1}{x} p^x (1-p)^r$	$\ln p$	$-x \ln(1-e^\theta)$	1	log

ϕ is the dispersion parameter, and $A(\theta)$ is called the log partition function. These are related to the mean and variance through

$$EX = A'(\theta), \qquad VX = \phi A''(\theta)$$

The binomial and poisson distributions have relationships between their means & variances, which may not hold in the data. For example using a poisson distribution assumes that $VX = EX$, if the variance in a particular

group is very different to the mean then the data is overdispersed or underdispersed. This can be accounted for by allowing ϕ to vary instead of taking the values in the table - these are called quasipoisson & quasibinomial models. ϕ is usually estimated through the method of moments in these 'quasi' models, because it saves on needing to explicitly define the $c(x, \phi)$ term so is more computationally efficient.

The generalised linear model says that the *natural parameter* θ can be modelled by a linear predictor, $\theta = X\beta$. In the table above, the natural parameter is related to the mean of the distribution in some way (this is always true for a member of the exponential family), so GLMs have the form

$$g(EY|X) = X\beta$$

The function g is called the *canonical link function*. The canonical link function should be used when modelling unless you have a good reason to use something else. This is because canonical links are guaranteed to be 'nice' functions, in the sense that their log-likelihoods will always be quadratic and so easy to maximise through the usual techniques. Generally you would want a link function to have a range of $(-\infty, \infty)$, because there is no restriction on the range of values for the linear predictor. Canonical links always have this property. Using a different link - a log link for a binomial response for example - may cause convergence issues due to the range mismatch.

The interpretation of regression coefficients depends on the response distribution used in GLMs. To figure out the interpretation, start from the fact that the model is linear on the link scale and consider a one-unit change in one of the predictors. Imagine you have two people who are identical in every way, except one person has $x_i = \tau + 1$ and the other person has $x_i = \tau$. Then the linear predictors η for the two people are

$$\eta_1 = \beta_0 + \beta_1 x_1 + \cdots + \beta_i \tau + \cdots + \beta_p x_p$$
$$\eta_2 = \beta_0 + \beta_1 x_1 + \cdots + \beta_i (\tau + 1) + \cdots + \beta_p x_p$$

The difference between the two people is

$$\eta_2 - \eta_1 = \beta_i$$

So a one unit change in variable i causes a β_i unit change *on the link scale*. To interpret this properly you need to convert back to the response scale, which depends on the specific link function used. A similar thing holds for calculating combinations of the model parameters - first calculate everything on the link scale where things are linear, then transform back to the response scale for interpretation.

Binary data - logistic regression part 1

Suppose you have binary data (X, y), where $y_i \sim Bern(p_i)$. The canonical link function for binomial data is the logit $\ln[p/(1-p)]$. Models which use a logit as the response variable are called *logistic regression* models. So for binary data, the canonical GLM model is

$$\ln\left(\frac{p}{1-p}\right) = X\beta$$

Following on from the end of the previous section, a one unit change in the x_i variable is associated with a β_i unit change on the link scale, that is

$$\begin{aligned}\beta_i &= \ln\left(\frac{p_2}{1-p_2}\right) - \ln\left(\frac{p_1}{1-p_1}\right) \\ &= \ln\left(\frac{p_2}{1-p_2} \Big/ \frac{p_1}{1-p_1}\right)\end{aligned}$$

The right hand side is the log odds ratio. So e^{β_i} represents the odds ratio for the outcome in someone with xi one unit higher than someone else, with all other variables held constant.

GLMs are fit in R using the `glm` command. As an example, here's how to fit a logistic model to the `insect.dta` data. The insect data contains data on 8 *groups* of insects. The number of insects `n` in each group is recorded, along with the `dose` of insecticide given to the group and the number `r` of insects who died after receiving the dose. There's two ways to model this data, the first is to work with the grouped data directly, and the other is to convert the data into individual format. Here's both of them:

```
insect = haven::read_dta('practicals/data/insect.dta')

# Model on grouped data
m1 = glm(cbind(r, n - r) ~ dose, data = insect, family = binomial(link = 'logit'))

# Convert to individual format and fit second model
died = c()
for (i in 1:nrow(insect)) died = c(died,
                                   rep(1, insect$r[i]),
                                   rep(0, insect$n[i] - insect$r[i]))
insect_indiv = data.frame(dose = rep(insect$dose, insect$n),
                          died = died)

# Fit model
m2 = glm(died ~ dose, data = insect_indiv, family = binomial(link = 'logit'))
```

The syntax for `m1` looks a bit weird - if the data is grouped then `glm` function needs the left hand side of the formula to be a 2-column table, the first column is the number of successes and the second is the number of failures. This is needed because `glm` assumes you're working with individual data. Stata makes you do something similar, the syntax for the same model in stata is `glm r dose, family(binomial n) link(logit)`. Stata needs you to tell it what the group size variable is, and R needs you to make a table. No big deal.
The syntax for `m2` is much more similar to the usual `lm` type models. The only new bit with `glm` is the `family` argument which lets you specify the distribution of the response variable and, inside the `family` function, you specify the link function to use. All the family functions will use the canonical link by default, so the link argument only needs to be specified if you're working with non-canonical links. As was mentioned in the previous section, this can be a bad idea - changing the link to log instead of logit will cause the model to not converge.

Both of the models have the exact same parameter estimates, makes sense seeing as they're the same models fit to the same data:

```
# Use the broom package to collect the output into the usual format
broom::tidy(m1)
```

```
# A tibble: 2 x 5
```

```
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)  -14.1      1.23       -11.5 1.92e-30
2 dose           0.237    0.0203      11.7 2.21e-31
```

```
broom::tidy(m2)
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)  -14.1      1.23       -11.5 1.92e-30
2 dose           0.237    0.0203      11.7 2.21e-31
```

Accounting for dispersion is easy, just change the family argument. This code fits a quasibinomial model:

```
glm(died ~ dose, data = insect_indiv, family = quasibinomial())
```

Quasibinomials allow for under or over dispersion compared to a binomial model, so the standard error and confidence intervals from this model will be slightly different to the previous two models. The parameter estimates will be the same. It's not possible to do inference about the dispersion parameter ϕ since it requires the $c(x, \phi)$ term in the exponential family equation to be known which isn't possible in general. This means that there isn't a formal test available to determine if dispersion is present in the data, it's more of a vibe.

How are these models fit?

When you run `glm` R starts doing a bunch of matrix algebra. It's nice to walk through the steps R does to fit GLMs, just so it isn't a black box and you can understand what is happening. All the steps are just the usual maximum likelihood estimation process, though the maths is a little bit more involved.

Suppose you've got individual data (X, y) and you want to fit a logistic model to y. The log likelihood is

$$\begin{aligned}\ln L(y) &= \sum_{i=1}^{n} y_i \ln p_i + (1 - y_i) \ln(1 - p_i) \\ &= \ln(1 - p_i) + y_i \ln\left(\frac{p_i}{1 - p_i}\right)\end{aligned}$$

Where p_i is the probability of success for person i ($y_i = 1$). Using the canonical link you would fit a linear model to the log odds, since it is the natural parameter for the problem:

$$\ln\left(\frac{p}{1-p}\right) = X\beta, \qquad \ln\left(\frac{p_i}{1-p_i}\right) = x_i^T\beta \text{ for individual } i$$

This lets us write the log likelihood in terms of the design matrix X and the parameter vector β, a bit of rearranging gives

$$p_i = \frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}} = \frac{1}{1 + e^{-x_i^T \beta}}$$
$$1 - p_i = \frac{1}{1 + e^{x_i^T \beta}}$$

Giving a log likelihood of

$$\ln L(\beta) = \sum_{i=1}^{n} -\ln(1 + e^{x_i^T \beta}) + y_i x_i^T \beta$$

Differentiating with respect to the j^{th} component of β, β_j gives

$$\partial_{\beta_j} \ln L(\beta) = \sum_{i=1}^{n} y_i x_{i,j}^T - p_i x_{i,j}^T$$

Where $x_{i,j}^T$ is the j^{th} element of x_i^T. This can be written in matrix form as

$$\nabla_\beta \ln L(\beta) = X^T (y - p)$$

This is the *score equation*, and setting it equal to zero and solving for β gives the MLE. This is difficult / impossible to do in general, so instead R uses numerical methods to get the solution. R uses Newton Raphson to find β, which finds the roots of a function $f(x)$ using the update rule of

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

(To derive this imagine x_{n+1} is the root, calculate the slope at x_n and solve for x_{n+1}). Applying this to the score equation gives an update rule

$$\beta_{(k+1)} = \beta_{(k)} - H^{-1}\left(\beta_{(k)}\right) \nabla_\beta \ln L\left(\beta_{(k)}\right)$$

Where H is the matrix of derivative of the score function (or the second derivatives of the log likelihood), called the *Hessian*. Differentiating the equation for the j^{th} component of the derivative of the log likelihood with respect to β_k gives the k, j element of H as

$$\begin{aligned}
\partial_{\beta_k} (\nabla_\beta \ln L)_{(j)} &= \partial_{\beta_k} \sum_{i=1}^{n} y_i x_{i,j}^T - \frac{x_{i,j}^T}{1 + e^{-x_i^T \beta}} \\
&= -\sum_{i=1}^{n} x_{i,j}^T x_{i,k}^T \frac{e^{-x_i^T \beta}}{(1 + e^{-x_i^T \beta})^2} \\
&= -\sum_{i=1}^{n} x_{i,j}^T \, x_{i,k}^T \, p_i \, (1 - p_i)
\end{aligned}$$

Or, in matrix form

$$H = -X^T D X, \qquad D = diag(p_i \, (1 - p_i))$$

All what R does is use the update rule (along with the matrices we just derived) to keep updating the estimate of β. Eventually the estimate doesn't change much between steps - the estimate converges - and the algorithm stops.

When a model is fit, R calculates the design matrix X and it can be accessed using `model.matrix(my_model)`. Here's the algorithm done 'by hand' on the insect data:

```
logistic_byhand = function(X, y, max_iter = 25, tol = 1e-10){
  # X = design matrix, y = response vector

  beta = rep(0, ncol(X)) # Initial guess for beta, to be updated using NR
  for (i in 1:max_iter){
    beta_old = beta
    p = 1 / (1 + exp(-X %*% beta_old))
    D = diag(as.numeric(p * (1 - p)))
    H = -crossprod(X, D %*% X)
    score = t(X) %*% (y - p)

    beta = beta_old - solve(H, score) # Update step
    if (sqrt(crossprod(beta - beta_old)) < tol) break
  }
  return(beta)
}

X = model.matrix(m2)
y = insect_indiv$died

logistic_byhand(X, y)
```

```
                  [,1]
(Intercept) -14.0864027
dose          0.2365929
```

Which is exactly the same coefficients in `m2`.

Count data

Count data is usually modelled using either the Poisson or negative binomial distributions. Poisson regression is usually taken as an initial model, and negative binomial is used if there's evidence of dispersion in the data. Similar to the binomial family we saw in the last section, there's also a quasipoisson family which allows the dispersion parameter to vary. If there is dispersion a Poisson model will not be able to model the data as accurately as a negative binomial or quasipoisson model. As these models better capture the variation in the data, negative binomial and quasipoisson models will differ to a poisson model in the standard errors and confidence intervals - parameter estimates will be similar or identical, regardless of which family is chosen.

Suppose you have count data of the form (X, y) where $y_i \sim Po(\lambda_i)$. λ_i is the expected count in a given period. The canonical link function for Poisson data is the log, and the natural parameter is $\ln(\lambda)$. The canonical GLM for count data is

$$\ln(\lambda) = X\beta$$

Going through the same steps as we did in logistic regression, a one unit change in the x_i variable is associated with a β_i unit change on the link scale, that is

$$\beta_i = \ln(\lambda_2) - \ln(\lambda_1) = \ln\left(\frac{\lambda_2}{\lambda_1}\right)$$

So e^{β_i} is a *rate ratio*. λ is the expected number of events (adjusting for the variables X) - so e^{β_i} is the multiplicative increase in mean number of events in person 2 relative to person 1. The same interpretation holds for negative binomial models.

To fit count models, just change the `family` argument in `glm`. Here's a model which predicts length of stay in hospital (the *count* of the number of days someone is in hospital for) given their age group and the type of admission:

```
medpar = haven::read_dta('practicals/data/medpar.dta')

m1 = glm(los ~ as.factor(age) + type2 + type3, data = medpar, family = poisson())
broom::tidy(m1)
```

```
# A tibble: 11 x 5
   term            estimate std.error statistic   p.value
   <chr>              <dbl>     <dbl>     <dbl>     <dbl>
 1 (Intercept)        2.43     0.117      20.8  2.30e- 96
 2 as.factor(age)2   -0.254    0.123      -2.06 3.94e-  2
 3 as.factor(age)3   -0.201    0.119      -1.69 9.05e-  2
 4 as.factor(age)4   -0.234    0.118      -1.99 4.65e-  2
 5 as.factor(age)5   -0.267    0.118      -2.26 2.35e-  2
 6 as.factor(age)6   -0.229    0.118      -1.95 5.14e-  2
 7 as.factor(age)7   -0.275    0.119      -2.32 2.04e-  2
 8 as.factor(age)8   -0.323    0.121      -2.66 7.78e-  3
 9 as.factor(age)9   -0.312    0.126      -2.48 1.33e-  2
10 type2              0.236    0.0210     11.2  2.40e- 29
11 type3              0.725    0.0260     27.9  1.41e-171
```

Compared to people of the same age group who are admitted to hospital by type 1, people admitted by type 2 have a length of stay which is $e^{0.236} = 1.27$ times longer (that is, 27% longer), and people admitted by type 3 have a length of stay more than twice as long as similar people admitted through type 1.

Negative binomial models aren't included in the `glm` function by default, but the `MASS` package has an implementation if you want to fit a negative binomial model:

```
m2 = MASS::glm.nb(los ~ as.factor(age) + type2 + type3, data = medpar)
broom::tidy(m2)
```

```
# A tibble: 11 x 5
   term            estimate std.error statistic  p.value
   <chr>              <dbl>     <dbl>     <dbl>    <dbl>
 1 (Intercept)        2.45     0.297      8.27  1.37e-16
 2 as.factor(age)2   -0.306    0.311     -0.982 3.26e- 1
 3 as.factor(age)3   -0.210    0.302     -0.696 4.86e- 1
 4 as.factor(age)4   -0.259    0.299     -0.864 3.88e- 1
 5 as.factor(age)5   -0.309    0.299     -1.03  3.02e- 1
 6 as.factor(age)6   -0.249    0.299     -0.834 4.04e- 1
 7 as.factor(age)7   -0.292    0.301     -0.970 3.32e- 1
 8 as.factor(age)8   -0.354    0.306     -1.16  2.48e- 1
 9 as.factor(age)9   -0.331    0.316     -1.05  2.95e- 1
10 type2              0.239    0.0503     4.75  2.06e- 6
11 type3              0.731    0.0757     9.66  4.50e-22
```

The coefficient estimates are the slightly different but the standard errors are much larger. This is because the negative binomial model is able to more properly model the variation in the data, leading to more accurate standard error estimates. Fitting a quasipoisson model to this data results in a fitted dispersion parameter of 6.3, a value of 1 would mean no dispersion is present so there is significant over-dispersion in this data. This explains why the standard errors in the two models are so different.

A third option to account for dispersion is to use robust standard errors. This can be done through the `sandwich` and `lmtest` packages:

```
vcov = sandwich::vcovHC(m1, type = 'HC1') # Same variance-covariance matrix as stata
lmtest::coeftest(m1, vcov = vcov)
```

```
z test of coefficients:

                 Estimate Std. Error z value  Pr(>|z|)
(Intercept)      2.427377   0.201187 12.0653 < 2.2e-16 ***
as.factor(age)2 -0.254162   0.231924 -1.0959    0.2731
as.factor(age)3 -0.201331   0.212464 -0.9476    0.3433
as.factor(age)4 -0.234396   0.205987 -1.1379    0.2552
as.factor(age)5 -0.266634   0.206301 -1.2925    0.1962
as.factor(age)6 -0.229200   0.207770 -1.1031    0.2700
as.factor(age)7 -0.275280   0.210022 -1.3107    0.1900
as.factor(age)8 -0.323079   0.212843 -1.5179    0.1290
as.factor(age)9 -0.312446   0.226582 -1.3790    0.1679
type2            0.235798   0.053212  4.4313 9.366e-06 ***
type3            0.725071   0.117429  6.1746 6.635e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Again these standard errors are much larger than the initial Poisson model, and are fairly similar to the standard errors from the negative binomial model. Setting `type = 'HC1'` tells R to use the same algorithm stata uses when doing robust regression, which makes these estimates agree with stata (up to rounding).

Models for rates (+ calculating DSRs)

The expected number of events λ can be expressed in terms of a rate. If μ is the rate of events per unit time, then $\lambda = \mu t$. We can use this to write the count data GLM in terms of the rate:

$$\begin{aligned} \ln(\lambda) &= X\beta \\ \ln(\mu t) &= X\beta \\ \implies \ln(\mu) &= \ln(t) + X\beta \end{aligned}$$

So to model a rate, we just need to add the $\ln(t)$ term to the linear predictor. This is called an *offset*, because the coefficient is forced to be 1 for the $\ln(t)$ term.

As an example, lets have a go at calculating mortality rates in Kazakhstan and Sweden following Germán Rodríguez's notes:

```
mortality = read.table("https://grodri.github.io/datasets/preston21long.dat",
                       header = F, col.names = c("country", "ageg", "pop", "deaths"))

mortality |>
  dplyr::group_by(country) |>
  dplyr::summarise(death_rate = 1e3 * sum(deaths) / sum(pop))
```

```
# A tibble: 2 x 2
  country    death_rate
  <chr>           <dbl>
1 Kazakhstan       7.42
2 Sweden          10.5
```

Sweden has a higher death rate than Kazakhstan, which feels weird. The issue is that Sweden has a lot more old people compared to Kazakhstan, and old people tend to die more than young people. It's not correct to compare the death rates because the age structure of the two countries are very different, age needs to be *adjusted for* in the analysis.

Adjusting for things is exactly what regression is good for! So fit a model for the rates, adjusting for age:

```
m3 = glm(deaths ~ country + ageg + offset(log(pop)), data = mortality, family = poisson)
```

The coefficient on the Sweden term is the mortality rate ratio for Sweden vs Kazakhstan, exponentiating gives `exp(coef(m3[2]))` $= 0.619$ - after accounting for the different population structure in the two countries, mortality rates in Sweden is 38.1% lower than Kazakhstan (or equivalently, mortality rates in Kazakhstan are $1/0.619 - 1 = 61.6\%$ higher than Sweden).
This is very similar to how direct standardisation is done. The idea of direct standardisation uses the fact that rates are weighted sums and the mortality rate depends (1) the age specific mortality rate, and (2) the age structure of each country. Direct standardisation applies the age specific mortality rates of the two countries to a 'reference population'. This removes any possible age structure effect as the age specific rates are now being applied to the same 'standard' population. Below we calculate a reference population which is just the average of the two countries age structures, then use our modelled rates to estimate how many deaths we would see when applying each countries age specific rates to this reference population:

```r
av_pop = mortality |>
  dplyr::group_by(ageg) |>
  dplyr::summarise(pop = mean(pop))

new_df = dplyr::bind_rows(av_pop |> dplyr::mutate(country = 'Sweden'),
                          av_pop |> dplyr::mutate(country = 'Kazakhstan'))

new_df$pred_deaths = predict(m3, newdata = new_df, type = 'response')
new_df |>
  dplyr::group_by(country) |>
  dplyr::summarise(pred_deaths = sum(pred_deaths),
            pop = sum(pop),
            rate = 1e3 * pred_deaths / pop)
```

```
# A tibble: 2 x 4
  country    pred_deaths      pop  rate
  <chr>            <dbl>    <dbl> <dbl>
1 Kazakhstan      69641. 6542164. 10.6
2 Sweden          43118. 6542164.  6.59
```

Passing `type = response` to `predict` tells R to return predictions on the y scale (counts, rather than log counts).
The mortality rate in Sweden vs Kazakhstan is $6.59/10.6 = 0.622$, so mortality rates in Sweden are $1 - 0.622 = 37.8\%$ lower than Kazakhstan. This is very similar to the coefficient from the model. The model approach is generally better than calculating DSRs by hand - it's easier to do, allows you to adjust for other confounders, and you get confidence intervals for free (there are formulas for DSR confidence intervals, but they can be a bit involved).

How are these models fit?

We can go through the steps we used to fit the logistic regression model by hand in the previous section if we wanted to. Instead of working through implementations for specific GLMs, we're now going to derive the way how R fits all GLMs - an algorithm called Iteratively Reweighted Least Squares.

Suppose the response vector y follows some 1-parameter exponential family distribution. The density f has the form

$$\ln f(y) = \eta(\theta)T(y) - A(\theta) + c(y)$$

For simplicity, assume that the distribution is in *canonical form*. This means that η, T are both the identity function and we have the relationship $Ey = A'(\theta)$, $Vy = A''(\theta)$. The log likelihood is then

$$\ln L(y) = \ln \prod_{i=1}^{n} f(y_i) = \sum_{i=1}^{n} \ln f(y_i) = \sum_{i=1}^{n} \theta_i y_i - A(\theta_i) + c(y_i)$$

As usual we will model the natural parameter with a linear predictor, $\theta = X\beta$. Writing things in terms of β gives

$$\ln L(\beta) = \sum_{i=1}^{n} x_i^T \beta \, y_i - A(x_i^T \beta) + c(y_i)$$

We can differentiate this twice to get formulas for the elements of the score function and the Hessian

$$\partial_{\beta_j} \ln L(\beta) = \sum_{i=1}^{n} x_{i,j}^T \, y_i - x_{i,j}^T \, A'(x_i^T \beta) \quad \implies \quad \nabla_\beta \ln L(\beta) = X^T (y - A'(X\beta))$$

$$\partial_{\beta_k} \partial_{\beta_j} \ln L(\beta) = -\sum_{i=1}^{n} x_{i,j}^T \, x_{i,k}^T \, A''(x_i^T \beta) \quad \implies \quad H = -X^T D X, \;\; D = \text{diag}(A''(X\beta))$$

Using the relationship between the derivatives of A and the mean and variance simplifies this to

$$\begin{aligned} \nabla_\beta \ln L(\beta) &= X^T (y - Ey) \\ H &= -X^T D X, \;\; D = \text{diag}(Vy) \end{aligned}$$

If $y \sim Bern(p)$ then these formulas reduce to the ones we derived back in the logistic regression section. There is a relationship between the mean and the natural parameter θ given by the link function $g(Ey) = \theta$. Since we're modelling $\theta = X\beta$ we can write the mean in terms of the linear predictor as $Ey = g^{-1}(X\beta)$. We can put these into the usual Newton Raphson update step to estimate β

$$\beta_{(k+1)} = \beta_{(k)} - H^{-1}(\beta_{(k)}) \nabla_\beta \ln L \left(\beta_{(k)}\right)$$

This requires very slight modification to our previous code. We need to add in the inverse link function and a function to calculate the variance, then use the formulas for the score function and Hessian. R stores these function inside the `family` class of functions we've been passing to `glm`. Here's all the things stored in `poisson()`:

```
names(poisson())
```

```
 [1] "family"     "link"       "linkfun"    "linkinv"    "variance"
 [6] "dev.resids" "aic"        "mu.eta"     "initialize" "validmu"
[11] "valideta"   "simulate"
```

Which means we can just add a `family` argument and then access the inverse link and variance functions through that. One extra modification is needed to deal with models which include offsets. The linear predictor $X\beta$ is replaced by $X\beta + offset$, which is an easy enough change. This code fits any GLM which has a `family` function already built into R:

```
glm_byhand = function(X, Y, family, offset = NULL, max_iter = 25, tol = 1e-10){
  # X = design matrix, y = response vector, family = family class function

  beta = rep(0, ncol(X))
  if (is.null(offset)) offset = 0

  for (i in 1:max_iter){
    beta_old = beta

    mu = family$linkinv(X %*% beta_old + offset)
    D  = diag(as.numeric(family$variance(mu)))
    H  = -crossprod(X, D %*% X)
    score = t(X) %*% (y - mu)

    beta = beta_old - solve(H, score)
    if (sqrt(crossprod(beta - beta_old)) < tol) break
  }
  return(beta)
}
```

Let's use this model to fit `m1` and `m3`

```
X = model.matrix(m1)
y = medpar$los
glm_byhand(X, y, poisson())
```

```
                     [,1]
(Intercept)      2.4273769
as.factor(age)2 -0.2541625
as.factor(age)3 -0.2013314
as.factor(age)4 -0.2343959
as.factor(age)5 -0.2666340
as.factor(age)6 -0.2291997
as.factor(age)7 -0.2752798
as.factor(age)8 -0.3230793
as.factor(age)9 -0.3124464
type2            0.2357979
type3            0.7250714
```

```
X = model.matrix(m3)
y = mortality$deaths
coefs_byhand = glm_byhand(X, y, poisson(), offset = log(mortality$pop))

coefs_byhand[2] # log(DSR_swe / DSR_kaz), should be -0.479
```

```
[1] -0.4794215
```

This exactly matches the results we got from using the `glm` command.

The code we just wrote is actually a very inefficient implementation of iteratively reweighted least squares (IRLS). To derive the IRLS algorithm, just put the equations for the score function and Hessian into the update rule

$$\begin{aligned}\beta_{(k+1)} &= \beta_{(k)} - H^{-1}(\beta_{(k)})\nabla_\beta \ln L\left(\beta_{(k)}\right) \\ &= \beta_{(k)} + \left[X^T DX\right]^{-1} X^T(y - Ey)\end{aligned}$$

D and Ey both depend on $\beta_{(k)}$ but we're not writing that out (just to save on clutter). The $y - Ey$ term almost looks like a Wald statistic, except it isn't divided by the variance. Since $D = \text{diag}(Vy)$ and remembering that 'dividing' in matrix algebra becomes 'multiply by the inverse matrix', we define the vector $a = D^{-1}(y - Ey)$ because it looks like a Wald statistic, and write the update step in terms of a

$$\beta_{(k+1)} = \beta_{(k)} + \left[X^T DX\right]^{-1} X^T Da$$

Now notice that we can write $\beta_{(k)}$ in the following, slightly strange looking, way

$$\beta_{(k)} = \left[X^T DX\right]^{-1} \left[X^T DX\right] \beta_{(k)}$$

Then the update step becomes

$$\beta_{(k)} = \left[X^T DX\right]^{-1} X^T D(a + X\beta) = \left[X^T DX\right]^{-1} X^T Dz$$

This is the normal equation from linear regression, $\hat{\beta} = (X^T X)^{-1} X^T z$, *weighted* by the variance matrix D. This is IRLS. We can implement IRLS by slightly modifying our code

```
irls = function(X, y, family, offset = NULL, max_iter = 25, tol = 1e-10){
  beta = rep(0, ncol(X))
  if (is.null(offset)) offset = 0

  for (i in 1:max_iter){
    beta_old = beta

    mu  = family$linkinv(X %*% beta_old + offset)
    D   = diag(as.numeric(family$variance(mu)))
    a   = solve(D, y - mu)
    z   = a + X %*% beta_old
    XtD = crossprod(X, D)

    beta = solve(XtD %*% X, XtD %*% z)
    if (sqrt(crossprod(beta - beta_old)) < tol) break
  }
  return(beta)
}
```

This saves on computing the score function, and there are very efficient algorithms for solving the normal equations, so this implementation is more efficient than our previous one.

Model fit, checks, and performance

Fit and performance

We've fit a bunch of models so far, now we turn to quantifying model performance and fit. The models are fit through maximum likelihood, so R calculates the (estimated) maximum log likelihood when fitting models. This suggests that some sort of comparison of the maximum log likelihoods may be a good starting point for comparing models. If we have two nested models - that is, model 1 has parameters $\psi = (\psi_1, ..., \psi_a)$, $\tau = (\tau_1, ..., \tau_b)$ and model 2 has parameters ψ (so model 2 is a special case of model 1 where $\tau = 0$) - then the profile log likelihood ratio is distributed chi squared

$$-2(\ln L(\psi = \hat{\psi}, \tau = 0) - \ln L(\psi = \hat{\psi}, \tau = \hat{\tau})) \sim \chi^2_a$$

Which can be used to see if adding extra parameters to a model leads to significant improvements in model fit. Formally this is testing the null hypothesis that the simpler model is correctly specified, and small p-values are evidence against the null. These (profile) log likelihoods are the log likelihoods evaluated at the MLE for the parameters, they're exactly the maximum log likelihoods returned by R - it's no extra work to use these to construct the log likelihood ratio and test. As an example, the code below fits two models to the `cattle` data, here we're testing to see if the model with an interaction term has better fit than the model without:

```
cattle = read.csv('practicals/data/cattle.csv')
m1 = glm(cbind(infect, cattle - infect) ~ as.factor(group) + dfactor,
         data = cattle, family = binomial())
m2 = glm(cbind(infect, cattle - infect) ~ as.factor(group) * dfactor,
         data = cattle, family = binomial())

lr = -2 * (logLik(m1) - logLik(m2))[1]          # 0.258. [1] makes R store lr as a number
df_resid = df.residual(m1) - df.residual(m2)    # m1 has 3 parameters, m2 has 4, 4 - 3 = 1
1 - pchisq(lr, df_resid)                        # 0.611
```

```
[1] 0.611753
```

There is no evidence that the interaction term leads to a significantly better fit. In this case, since there's only one extra parameter, the Wald test gives a similar p-value (the two tests are asymptotically equivalent). If you have multiple extra parameters then the likilihood ratio test is testing the hypothesis that at least one of the coefficients is non-zero, which is a better test to do compared to looking at each coefficient individually because it avoids multiple testing.

The absolute best model is the so-called *saturated model*, where every observation has its own parameter. In this case the model can exactly fit the data, so will be the best fit possible (and also the largest possible log likelihood). This gives a baseline to compare against when constructing the likelihood ratio. Here's the likelihood of the saturated model for the cattle data

```
cattle$id = 1:nrow(cattle)
m_sat = glm(cbind(infect, cattle - infect) ~ as.factor(id),
            data = cattle, family = binomial())
logLik(m_sat)
```

```
'log Lik.' -11.90887 (df=10)
```

Using the saturated log likelihood in the likelihood ratio gives the *scaled deviance* of the model:

$$S = -2 \ln \left(\frac{L_m}{L_s} \right) = -2(\ln L_m - \ln L_s)$$

The deviance is just the scaled deviance multiplied by the dispersion parameter, $D = \phi S$. Large values of the deviance (scaled or unscaled) mean that your model has a much larger log likelihood than the best possible model, so your model is much worse than the best possible model. It's a 'badness of fit' measure - smaller values are better. The formula for the deviance is a log likelihood ratio, so $D \sim \chi^2_{n-p}$ where p is the number of parameters in the model. This makes sense, every model is nested inside saturated model, because you can set some of the parameters of the nested model to zero to get any other model. The `deviance` function calculates deviance for a model:

```
deviance(m1)
```

```
[1] 2.401511
```

```
# Calculate deviance by hand
-2 * (logLik(m1) - logLik(m_sat))[1]
```

```
[1] 2.401511
```

```
# Compare to a chi squared with 10 - 3 = 7 DOF for p-value
1 - pchisq(deviance(m1), df.residual(m1))
```

```
[1] 0.934329
```

There is no evidence of poor model fit.

Deviance is not a good measure of fit for individual data, because as the sample size n gets larger so does the number of parameters needed to fit the saturated model. in the large sample limit the saturated model will have an infinite number of parameters, which invalidates the set up needed for constructing the profile log likelihood ratio. This doesn't happen for grouped data, where increasing the number of observations will increase the size within each of the groups but keep the number of groups constant.
Since the issue comes from having individual data, it feels like a reasonable idea to group the data first and then calculate deviance (or some other summary statistic) on the grouped data. This is the idea behind the *Hosmer-Lemeshow* test - first the individual level data is grouped into g groups, then the expected number of events in each group is calculated using the model, and that is compared to the observed number of events in each group. The grouping produces a $2 \times g$ table of counts, so a natural test statistic is the chi-squared. Hosmer & Lemeshow showed that the test statistic is distributed chi-squared with $g - 2$ degrees of freedom

$$\chi^2 = \sum_{i=1}^{g} \frac{(y_i - n_i \hat{p}_i)^2}{n_i \, \hat{p}_i \, (1 - \hat{p}_i)} \sim \chi^2_{g-2}$$

Large values indicate that the predicted number of events are very different to the observed number of events, so large values indicate poor model fit. This tests the null hypothesis that the model is correctly specified, and a small p-value is evidence against the null.

This feels like an intuitive test, but it turns out that the Hosmer-Lemeshow test isn't very good. It depends on an arbitrary grouping of the data, so you will get different p-values if you change the number of groups or if you change the criteria used to form the groups. As an example, the code below runs the Hosmer-Lemeshow test on the `lbw` data for a range of different group sizes:

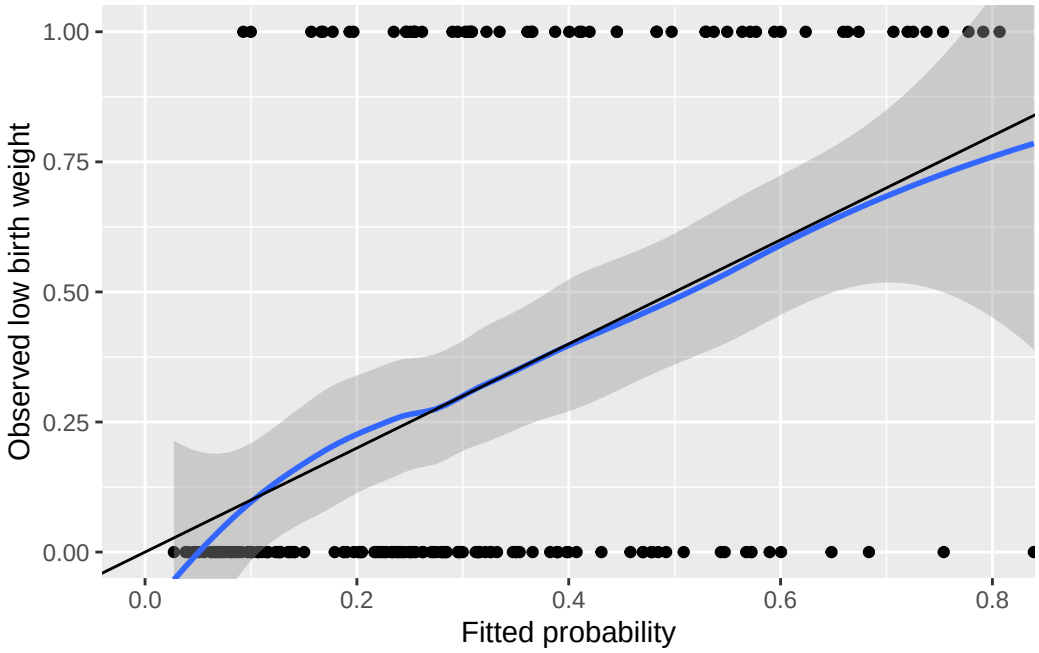
```
lbw = haven::read_dta('practicals/data/lbw.dta')
m = glm(low ~ age + lwt + as.factor(race) + smoke + ptl + ht + ui,
        data = lbw, family = binomial())

lbw$p = predict(m, lbw, type = 'response')
p_vals = vector('double', 10)
for (i in 1:10){
  lbw$grp = dplyr::ntile(lbw$p, n = i + 2)
  df_agg = lbw |>
    dplyr::group_by(grp) |>
    dplyr::summarise(p_hat = mean(p),
                     n = dplyr::n(),
                     y = sum(low)) |>
    dplyr::mutate(chi_sq = (y - n * p_hat)^2 / (n * p_hat * (1 - p_hat)))

  chi_sq = sum(df_agg$chi_sq)
  p_vals[i] = 1 - pchisq(chi_sq, i)
}
p_vals
```

```
[1] 0.8358710 0.9655871 0.4869830 0.9901576 0.9721686 0.9700237 0.7214799
[8] 0.2904041 0.6386188 0.8766098
```

The p-values vary a lot - there is a strong dependence on the number of groups. In all cases the p-value is larger than 0.05, so we do not reject the null that the model is correctly specified (that is, the test doesn't find any evidence that the model is incorrectly specified).

The Hosmer-Lemeshow test looks at how close the model predictions are to the observed data. This is a concept called *calibration*. A model is said to be "well calibrated" if the model predictions are close to the observed values. A better approach to assessing calibration is to use *flexible calibration curves*, which compares the predicted probabilities from your model against the observed empirical probabilities in the data. The idea is that, for each value of the fitted probability you can calculate the observed probability, and plotting these two quantities is a visual check of calibration. If the fitted probabilities agree with the observed probabilities, then the points will lie on the line $y = x$. The easiest way to so this is using a `lowess` smoother:

```
library(ggplot2)
ggplot(lbw) +
  geom_point(aes(x = p, y = low)) +
  geom_smooth(aes(x = p, y = low)) +
  geom_abline()
```



The calibration curve (blue) is close to the best fit line for all values of the fitted probability, which means the fitted probability is close to the observed probability for all values of the fitted probability - the model has good calibration.

Calibration by itself isn't very helpful. For example suppose you have a disease with a prevalence of 10%, and you're trying to build a predictive model to identify individuals at risk of the disease. A 'model' which predicts a probability of 10% for any individual will have perfect calibration at the population level, but will be absolutely useless at identifying high-risk individuals. You want a model which is well calibrated *and* explains a large amount of the variation in the data. The usual summary statistic for variance explained in linear regression is $R^2 = 1 - RSS_m/RSS_n$, where RSS_m is the residual sum of squares from the model and RSS_n is the residual sum of squares from the null (intercept only) model. The concept of a 'residual' doesn't immediately translate over to GLMs. To see this think about the output from a logistic model - the model can be used to estimate $P(y = 1|X)$ and the data contains observations of $y = 1$ or $y = 0$. The probabilities can be converted into predictions of $y = 1$ by applying a cut-off, saying that any predicted probability greater than some value is a prediction that $y = 1$, but this depends on the choice of cut-off value so isn't a consistent measure. Instead people usually look at *pseudo* R^2 measures, the most popular one being McFaddens index

$$R^2 = 1 - \frac{\ln L_m}{\ln L_n}$$

This has roughly the same interpretation as the usual R^2 measure, larger values are better and show that the model is explaining a large amount of the variation in the data. We can calculate the pseduo R^2 of our model as follows:

```
m_null = glm(low ~ 1, data = lbw, family = binomial())
r_sq = 1 - logLik(m) / logLik(m_null)
r_sq[1]
```

```
[1] 0.1415764
```

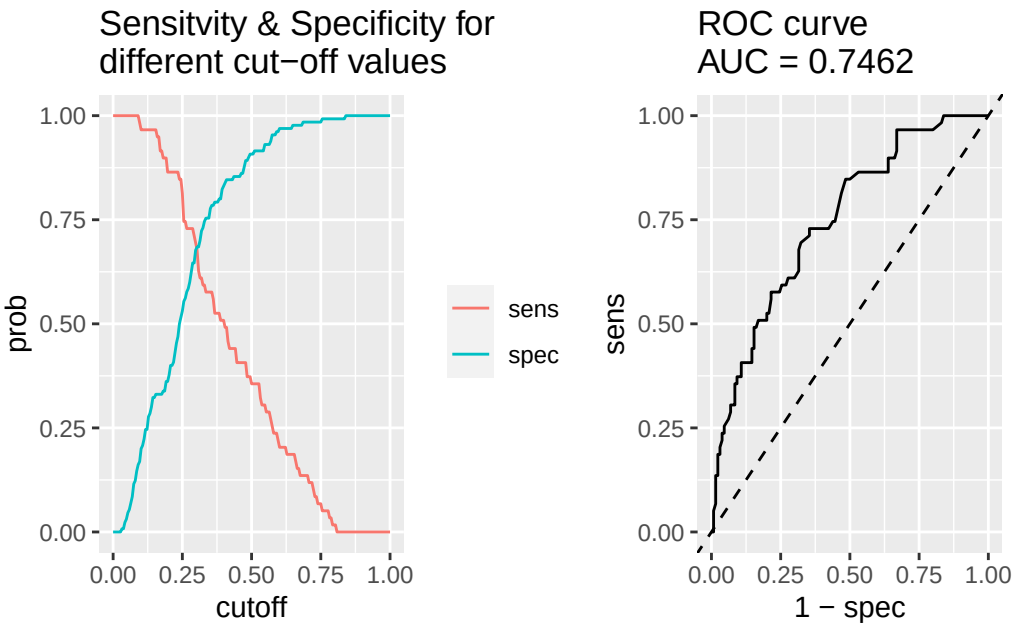
Alongside calibration, another important feature is the *discrimination* of the model. This is the ability of the model to correctly identify cases ($y = 1$) and non-cases ($y = 0$), and is measured through sensitivity and specificitiy. If we say any individual with a predicted probability $\hat{p}_i > p$ is classed as a case ($\hat{y}_i = 1$) then the sensitivity and specificity of the model can be written as

$$\text{sensitivity} = P(\hat{y}_i = 1 | y_i = 1) = \frac{\sum_{i=1}^{n} [\hat{p}_i > p,\, y_i = 1]}{\sum_{i=1}^{n} [y_i = 1]}$$

$$\text{specificity} = P(\hat{y}_i = 0 | y_i = 0) = \frac{\sum_{i=1}^{n} [\hat{p}_i \leq p,\, y_i = 0]}{\sum_{i=1}^{n} [y_i = 0]}$$

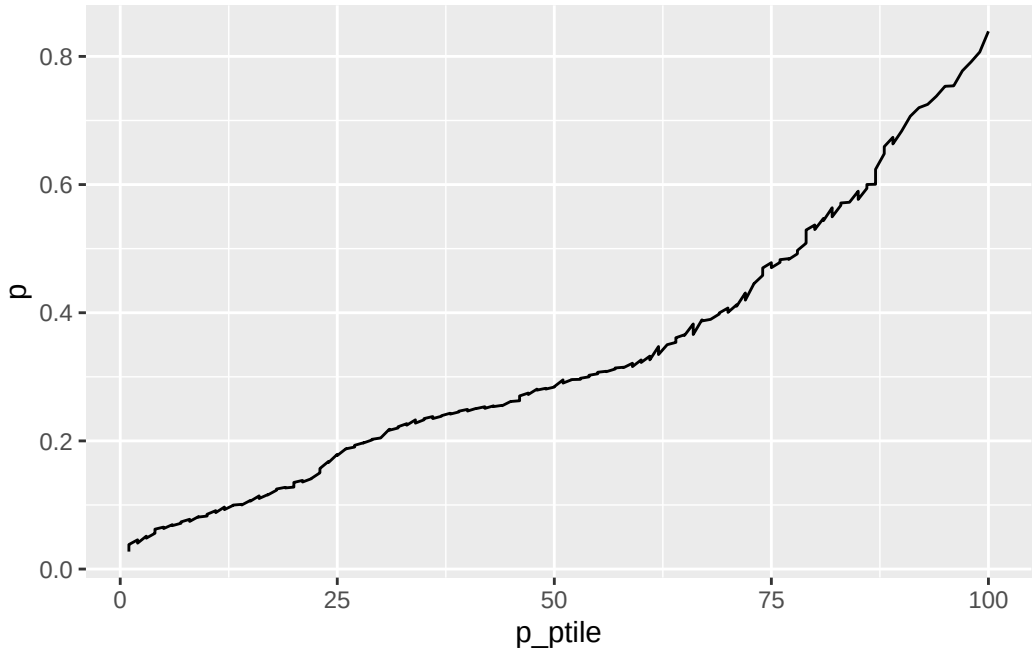
Where the square brackets are indicator variables (= 1 if the condition inside the brackets is true). These values depend on the cut-off value p, and this dependence can be shown in a *ROC plot*. We do it by hand here, but there is the `ROCR` package which does the heavy lifting for you. The `ROCR` package should be used in practice, no need to reinvent the wheel!

```
cutoff = seq(0, 1, length.out = nrow(lbw))
sens = vector('double', length(cutoff))
spec = vector('double', length(cutoff))

for (i in 1:length(cutoff)){
  c = cutoff[i]
  sens[i] = sum((lbw$p > c) & (lbw$low == 1)) / sum(lbw$low == 1)
  spec[i] = sum((lbw$p <= c) & (lbw$low == 0)) / sum(lbw$low == 0)
}

df = data.frame(cutoff = cutoff, sens = sens, spec = spec)
```



One drawback about sensitivity and specificity is that they are conditional on the outcome, which is no help in risk prediction. We want to know the probability of someone getting the disease *before they have the disease*, so we're conditioning on information we don't have. A different option is to construct *predictiveness curves* which sounds very fancy, but it's just the predicted probabilities plotted against its percentiles:

```
lbw$p_ptile = dplyr::ntile(lbw$p, 100)
ggplot(lbw, aes(p_ptile, p)) +
  geom_line()
```



Model checks

Observational data, marginal effects, collapsability

Observational data is tricky to work with due to *confounding*. People self-select into weight-management programs, people from more deprived areas are more likely to smoke, and people who go to the opera tend to live longer. The self-selection introduces *bias* into the data - the people who voluntarily enter the weight-management program will be different in some way to the general public, so directly comparing these people to the general population will be incorrect. Similarly people in more deprived areas aren't smoking more *because* they live in deprived areas, there will be other social factors which are influencing their decision to smoke. And going to the opera doesn't make you live longer, again there are other factors (e.g. having enough money to go to the opera) which are confounding the association.

These confounding variables need to be controlled for. This introduces two issues - the obvious one is what we don't measure all the confounders? This is called *unmeasured confounding* and is a issue which needs to be addressed in all observational work. The other one is deciding which confounders we should control for. This is often done using a *DAG*, where you draw the assumed causal associations between the variables, then there are rules (something called *do-calculus*) which say what variables need controlling for. The `daggity` package does the do-calculus work for you, and you can draw nice DAGs using `diagrammeR`.

Say you fit a model $g(EY|X) = X\beta$ or equivalently $EY|X = g^{-1}(X\beta)$. This is a *conditional model* - it produces predictions for specific values of X, i.e. for individuals. This is useful if you're interested in risk prediction for example, where you would take information about a specific individual, calculate a risk score, and then assign a treatment based on that risk score. It's less useful if you're interested in population effects, for example the impact of increasing the smoking age would have on lung cancer prevalence. Suppose there's some variables you're interested in - call them X - and some nuisance parameters Z you want to adjust for. You fit the conditional model as usual to get $EY|X, Z = g^{-1}(X\beta_1 + Z\beta_2)$, and then take the expectation (marginalise) over the nuisance parameters

$$EY|X = E_Z[EY|X, Z] = \int_Z dz\, EY|X, Z\, f_Z(Z)$$

This is a marginal model which estimates the prevalence of Y in the population (after adjusting for X). Marginal effects are useful for policy decisions.
Your model will estimate $EY|X, Z$, and the empirical distribution $\widehat{f_Z}$ can be estimated from the data, so all these terms are computable. You can do marginalisation by hand or by using the `margins` package

Cohort, case control, and matched studies - logisitic regression part 2

Multinomial & ordinal models - logistic regression part 3