Chapter 11
Modeling categorical variables: Logistic regression

James Myers

2022/5/10

1. Introduction

As we've seen, regression is a powerful technique, the secret heart of everything from *t* tests to ANOVA and beyond, but so far all of the examples have involved continuous-valued data that form a normal distribution (or can be transformed into a reasonably normal shape). But as we've also seen, not all of the data that linguists care about are continuous or normally distributed. Is there any way to apply the power of regression to categorical data, like the binary responses we tested with the binomial test, or the counts we tested with chi-squared tests?

Of course, but it took a long time to make it practical. Even though statistical methods have been around for centuries, it wasn't until the 1960s, when computers were increasing in power, that researchers really figured out how to apply regression techniques to categorical data. The trick was to think of the dependent variable in terms of a function that makes the values a bit more linear, just enough so that the logic of linear regression can be generalized to them. Thus was born **generalized linear regression**.

By far the most common type of generalized linear regression is **logistic regression** (named after its particular linearization function), which is used to analyze **binary variables**. So if your dependent variable involves accuracy (correct vs. incorrect) or binary judgments (acceptable vs. unacceptable) or the appearance of some sociolinguistic feature in a corpus (present vs. absent) or any other such binary situation, and you want to predict whether the probability of getting one or the other response depends on a bunch of independent variables (whether they are categorical or continuous, interacting or not), then logistic regression is the right tool for the job. In fact, if you're a sociolinguist, you may have heard of a software system called **Varbrul** (short for "variable rule"), which has logistic regression at its core.

This makes generalized linear models much more flexible than methods like the binomial test or chi-squared tests. With contingency tables, the *independent* variables also have to be categorical (nominal), and we can only test one-way main effects, or two-way interactions, but not both at the same time. We can't test models involving continuous independent variables, or more than two independent variables. But with a regression approach, we can do all of this.

What if your dependent variable is categorical but not binary, with three or more levels? No problem; you can use **multinomial logistic regression**. What if you want to analyze count data, rather than binary data? You can use **Poisson regression** (which I already mentioned early in this book). What if your dependent variable is **ordinal** (e.g., the levels are categorical but are ordered)? There are regression techniques for that situation too. There is even something called **generalized additive modeling** that can model data no matter how wiggly the best-fit line ends up being. And there are categorical regression tools for special situations, including word frequencies in a corpus.

You can do all of this too - in R! From now on in this book, Excel becomes a lot less useful. It may help with simple calculations or data organization when preparing a logistic regression, but the regression itself has to be run in a real statistics program like R.

2. Why go logistic?

In the old days before logistic regression, people dealing with binary data would often try to treat it as normally distributed categorical data anyway, just so they could use ordinary linear regression, maybe transforming ratios (e.g., accuracy rates) using tricks like the arcsine square root transformation, which I briefly mentioned in an earlier chapter. But now that we have tools like logistic regression, Warton and Hui (2011) are right to say that "arcsine is asinine": analyzing the binary data as binary data is the best way to go.

To explain why, let's start with a simple example involving a binary dependent variable, and show how much better logistic regression does in analyzing it than ordinary linear regression. Then we need to dive into the math of logistic regression, first explaining its strange name. You might also wonder how this new categorical data analysis technique relates to our old friend the chi-squared test; we'll discuss that too. Then we need to face a bit of sad reality: the power of logistic regression comes with a cost. Being the first major computer-dependent statistical method, it is also the first one that can "crash": you might not be able to build the fancy model you want due to computational limitations. Finally, to help our monkey brains understand what a logistic regression model is really telling us, we need to discuss how to plot them, and how to estimate effect sizes.

2.1 Treating a binary response variable with respect

Sally the sociolinguist notices that some Martians always pronounce the final consonant of words, while other Martians always delete it. She also notices that the deletion seems to be more common for richer Martians, so she calls this phenomenon "rich deletion". To study it more systematically, she collects the data from 100 Martians contained in the file **richdeletion.txt**. In this file, each row represents one Martian, showing both that Martian's income in Martian dollars (Income) and whether the Martian is a deleting type of Martian (Deletion = 1) or not (Deletion = 0).

Here's the data:

rd = read.delim("richdeletion.txt")
head(rd)

```
     Deletion  Income
  1         1   58900
  2         1   87800
  3         0   17800
  4         0   47100
  5         0   14100
  6         0    9900
```

Just to get a sense of the data, she makes a scatter plot (in this case, using the **options()** function to change R's default setting for scientific notation so it doesn't turn the incomes into stupid-looking values like 2e+04), producing Figure 1.

options(scipen=5) # Forces R to display 5-digit numbers
plot(Deletion ~ Income, data = rd) # Plot lets you use formula notation too!

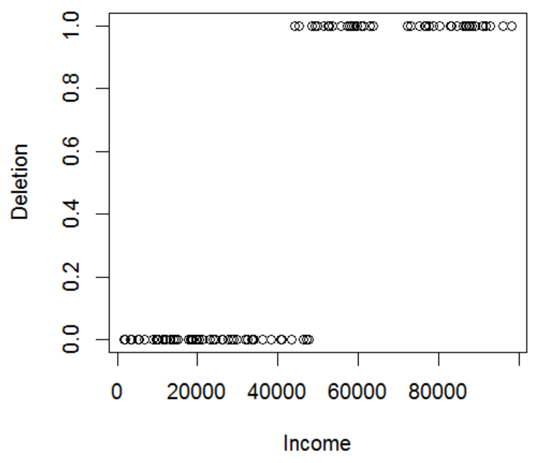

Figure 1. The relation between Deletion and Income

It looks like her hunch is on the right track: the higher a Martian's income, the more likely that Martian is to be a deleting Martian (with some overlap in the middle). But how can Sally go beyond a simple scatter plot and build a statistical model?

One thing Sally knows *not* to do in this case is to run an ordinary linear regression. Because the dependent variable is numerical, R won't stop her from doing it, but it's easy to see that it makes no sense here. This is what it would look like:

summary(lm(Deletion~Income, data = rd)) # Showing just the coefficients table below

```
Coefficients:
              Estimate  Std. Error    t value      Pr(>|t|)
(Intercept)  -0.23859   0.047692     -5.003    2.49E-06 ***
Income       1.54E-05   9.14E-07     16.820      < 2e-16 ***
```

If it's not obvious to you why this analysis makes no sense, let's add the line defined by the above regression coefficients to the plot we just made, as shown in Figure 2:

abline(lm(Deletion~Income, data = rd)) # Assumes scatter plot is still open

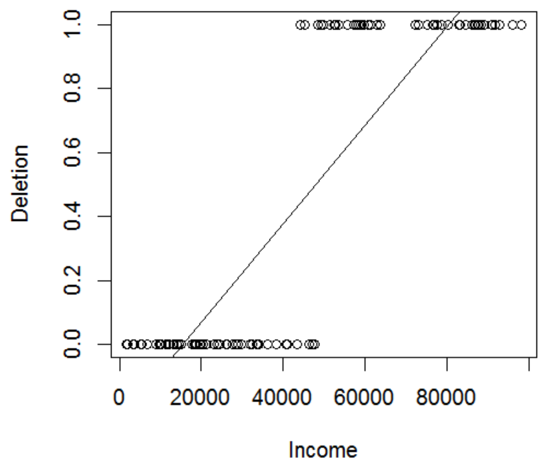

Figure 2. The relation between Deletion and Income, with a linear best-fit line

Do you see the problem now? The dependent variable is not continuous, but is always 0 or 1, with nothing between, and nothing above or below either. Doing a linear regression on this kind of data is ridiculous, since the so-called "best-fit" line not only goes infinitely far beyond both these two limits, but it doesn't even get very close to most of the dots in the scatter plot. It literally "misses the point(s)"!

Fortunately, Sally knows that her data set is just the kind where she can use logistic regression: her dependent variable is 0 vs. 1 and each data point is independent (from a different Martian). She also know how to do logistic regression in R: with the function for generalized linear models, called **glm()**. Since the logistic regression is designed for binary data, she identifies which type of generalized linear model she wants by using an argument that identifies which **family** of distribution she needs: the **binomial distribution**. You remember that distribution family, right? It's for binary data like flipping coins, or here, like deleting vs. not deleting. So Sally runs her analysis like this:

rd.glm = glm(Deletion~Income, family = "binomial", data = rd)

The quotation marks around the distribution family name are actually optional. But be sure not to forget to include the **family** argument, since otherwise R will make **glm()** act exactly like **lm()**, using the default value family = “gaussian”, i.e., with the normal distribution (named after the German mathematician Carl Friedrich Gauss [1777-1855], who worked on it, though as usual for things named after people, he wasn’t the first one to do so).

As usual with R models, running **glm()** just creates a **glm** object, in this case Sally’s logistic regression model. So she uses the **summary()** function to see what it tell us (if we don’t need to use the model object for another purpose, we could do these two steps in one line):

summary(rd.glm)

```
Call:
glm(formula = Deletion ~ Income, family = “binomial”, data = rd)

Deviance Residuals:
     Min         1Q   Median        3Q       Max
  -1.505  -0.00294  -0.00002   0.00026   1.73345

Coefficients:
              Estimate     Std. Error   z value   Pr(>|z|)
(Intercept)   -25.6798     10.88789     -2.359    0.0183 *
Income        0.000554     0.000233      2.383    0.0172 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ‘ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 137.186   on 99   degrees of freedom
Residual deviance:   12.564   on 98   degrees of freedom
AIC: 16.564

Number of Fisher Scoring iterations: 10
```

I’ll explain the details of this report later, but for now just focus on the regression coefficients table. It looks a lot like the one we got for the (invalid) ordinary linear model (with some differences to be explained later), showing separate results for the intercept and for the Income independent variable. Both of these effects are significant ($p < .05$). The coefficient for Income is also positive, consistent with Sally’s intuition that deletion is more common for richer Martians.

The coefficient for the intercept is negative, though. Can you guess what that might mean? Well, if the logistic regression model is trying to predict the probability of the Deletion value being 0 or 1, the intercept here represents the deletion probability when we ignore Income entirely. That is, it represents the default deletion rate. If deleting and non-deleting Martians were equally common in the sample, this coefficient would be zero. So the fact that it's negative suggests that there are more non-deleters (0) than deleters (1). Is that true? Yes indeed, as we can see using the **xtabs()** function:

```
xtabs(~Deletion, data = rd)
```

```
Deletion
   0  1
  56 44
```

So there you go: logistic regression. Of course, there's still a lot more to say. Like, how does this work? What do all those other things mean in the report? How do we plot the results properly (since Figure 2 is wrong)?

2.2 The math of logits

Statisticians figured out very early that linear regression doesn't make sense for a binary dependent variable, and they knew that there three things they needed for a proper model of binary data. Goal one is that the model should represent something meaningful about what we're trying to predict (the dependent variable). This means respecting binary data as binary. Goal two is that the model should express the relationship between the dependent and independent variables in a way that makes intuitive sense. This means thinking in terms of probability, or at least something related to probability. Goal three is that the model should handle the problem of how to fit infinitely long lines to a range restricted to the range 0 to 1.

Since the binomial distribution turns into the normal distribution as the sample gets larger (remember?), and since the mean of a bunch of 0s and 1s gives you a value that's exactly the same as the probability of getting a 1 (remember?), the very first attempt at regression analyses for binary variables (way back in the 1930s) was based on probabilities. This old type of model (which is still around) is called **probit** modeling. It handles goals one and two reasonably well, but it doesn't do a good job with goal three: probabilities, just like the raw data, are still stuck between 0 and 1.

In the 1960s, however, **logistic regression** was invented (though it took a little for it to become mainstream, as we'll see). By taking advantage of the increasing power of computers (another theme we'll see more of throughout the rest of this book), logistic regression was finally able to achieve all three goals (Pampel, 2000, is a short overview book; Speelman, 2014, is a short article reviewing its applications to corpus linguistics).

Logistic regression transforms the dependent variable using the **odds** (勝算比、發生比) rather than the probabilities. The odds of an event is the ratio of the probability of one outcome (here, 1) to the probability the opposite outcome (0):

Deriving the odds from probabilities: $P_1/P_0 = P_1/(1-P_1)$

Odds are used a lot in gambling. For example, you have 1:1 odds (pronounced "one to one" in English) of getting a head when you flip a coin. The advantage here is that using odds goes partway towards creating the infinite range we need for a linear model, since the maximum odds value is infinity. That is, if $P_1 = 1$, then $P_0 = 0$, and $1/0 = \infty$.

To make the regression line infinite in both directions, we use our old friend the logarithm, specifically, the **natural logarithm** (ln) based on that magic number $e = 2.718282\ldots$ (which R computes using **log()** and Excel computes using **=LN()**). The log of infinity is still positive infinity, but the log of 0 (the bottom of the odds ratio scale) is negative infinity:

```
log(Inf) # Yes, R lets you run computations on infinity!
```

```
[1] Inf
```

```
log(0)
```

```
[1] -Inf
```

So now we have the **log odds**, or the **logit**, and hence the name of our new statistical method, logistic regression:

Deriving the logit from probability: $\text{logit}(P_1) = ln\left(\frac{P_1}{1-P_1}\right)$

You can compute log odds by writing the above formula in R, or by installing the **gtools** package (Warnes et al., 2014). The function **logit(Pr)** turns the probability Pr into log odds Lo, and the function **inv.logit(Lo)** turns the log odds back into probability. For example, if your probability of getting 1 is .5, then the odds of getting 1 (instead of 0) is 1:1, or 1/1 = 1. Since the natural log (ln) of 1 is the value of y that makes $x = e^y = 1$, that means that $y = \ln 1 = 0$ (because $z^0 = 1$ no matter what z is).

```
library(gtools) # You have to install it first
logit(0.5)
```

```
[1] 0
```

```
inv.logit(0)
```

```
[1] 0.5
```

As shown in the plot in Figure 3, the logistic function has an S shape (technically, a **sigmoid** shape, named after the Greek letter for the /s/ sound, sigma, written Σ or σ... which are not S-shaped ...). This shape will prove useful later.

curve(logit(x), 0, 1)

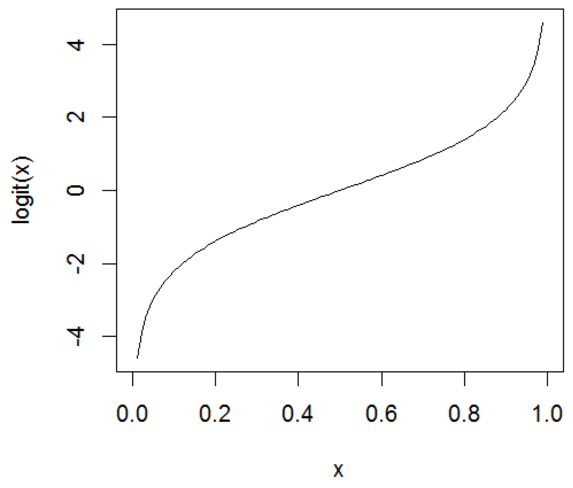

Figure 3. The S-shaped logistic function

You actually don't need the **gtools** package for this, since base R already has functions that do the same thing, and in fact they're in the familiar family of "p+distribution" for finding the area to the left of a given point and "q+distribution" for finding the point given an area. So if you rotated Figure 3 to put the logit(x) part on the bottom, you'd see the logistic distribution.

plogis(0) # Same as inv.logit(0) - try it, and other values too!
qlogis(0.5) # Same as logit(0.5) - ditto!

Because log odds have an infinite range, they provide a way to express probability information in a way that linear models can handle, even if you don't want to do a full-fledged logistic regression analysis. For example, Allerup and Elbro (1998) argue that if you want to run an ANOVA comparing differences in accuracy across experimental conditions, it's better to use log odds than the raw proportions (though I'd say it's even better just to run a logistic regression, with help of the mixed-effects techniques that we'll learn in the next chapter).

The log odds transformation also deals with goal two, expressing the relationship between the dependent and independent variables in a way that makes sense. To see this, think about some situation where you want to predict the probability for a binary variable y that's dependent on two independent variables x_1 and x_2. If those two independent variables are also independent of each other, then we can go back to basic probability theory and remember the multiplication rule. That rules says that the probability of *y* should depend on the product of x_1 and x_2, just as the probability of choosing K♥ from a deck of cards (1/52) is the same as the probability of choosing K (1/13) times the probability of choosing ♥ (1/4).

But since this is regression, we also have an intercept; we also need to use odds instead of probability. And let's do one more thing: instead of representing the intercept, x_1, and x_2 in terms of their raw coefficients (b_0, b_1, b_2), let's use the **exponential** function (**exp()** in R), that is, raise *e* to the power of those values. We still multiply them together because of the multiplication rule, though. That gives us this:

Part way to logistic regression: $\frac{P_1}{1-P_1} = (e^{b_0}) \cdot (e^{b_1 x_1}) \cdot (e^{b_2 x_2})$

Now let's take the natural logarithm (base *e*) of both sides. Not only does this convert the dependent variable into the logit, but even more amazingly, it also turns the rest of the equation into a linear equation (because **log()** is the inverse of **exp()**, and adding logarithms is the same as multiplying the original values):

Logistic regression: $ln\left(\frac{P_1}{1-P_1}\right) = b_0 + b_1 x_1 + b_2 x_2$

That's why logistic regression is a generalized *linear* model. In essence, by using the logit as a **link function**, it becomes possible to analyze binary data in a linear way. Moreover, as we've already seen, the relevant distribution for computing the *p* values is the **binomial** family, which is why you tell R to perform logistic regression by putting the **family = "binomial"** argument in the **glm()** function.

2.3 Logistic regression and chi-square

How does R compute the coefficients for the logistic regression model, and how is this related to the other categorical data analyses methods we know, like the binomial test and the chi-squared test?

If this were ordinary linear regression, R could just plug in our data (as a big matrix) and then compute the slope with a fixed formula. For example, if we observe the (*x*, *y*) pairs (1, 10) and (2, 20), what is the most likely slope (coefficient for *x*)? Well, it should be 10 (since $y = 10x$). This is derived from a formula generalized from the familiar formula for a line's slope that you learned in algebra class: $(y_2-y_1)/(x_2-x_1)$. The reason why mathematicians are sure that their formula is the optimal one is because of calculus (微積分學), which was used hundreds of years ago to find the optimal values for functions with certain well-defined properties. In the case of finding the best-fitting line in scatter plot, calculus was applied to **least-squares**

estimation to find the line that minimizes the (squared) distances from the line of each data point. R already knows the slope formula, so it doesn't need to do any calculus or any other hard work.

Due to the logistic transformation, however, it turns out that we can't estimate the model coefficients using least-squares estimation. Fortunately, the more general theory of **maximum likelihood estimation** still applies. This is the idea that the coefficients of the regression model should be the ones most likely to generate the observed data. So to find the coefficients in logistic regression, we (or rather, our computer programs) have to use an estimation algorithm, where we start with an initial guess, check the model fit (by seeing what likelihood it gives), adjust the guess slightly, check the model fit again to see if it's better, and so on, until any more changes don't make any noticeable improvement, or until we stop looping (since we don't want to get caught in an infinite loop if there's no good answer). The number of loops R makes while running **glm()** is given in the text report as "Number of Fisher Scoring iterations", so in the analysis of **richdeletion.txt** above, the report said it iterated 10 times before finally **converging** on the best-fitting model. (Yes, it's the famous Fisher again! He died before logistic regression became practical, but he had his hand in developing some aspects of the math, just as with most other kinds of statistics.)

The need for such algorithms makes logistic regression dependent on computers, which is why it wasn't practical until the 1960s. It also makes it prone to "crash": sometimes the estimation algorithm can't find any coefficients, and sometimes it gives inaccurate coefficients (because the algorithm stops too early). Fortunately, there are ways to check for dangers like this, and R will also often give you a warning.

2.3.1 Comparing logistic regression against other categorical tests

To get these basic concepts clear, let's start with the simplest possible regression model, namely one with only an intercept, like y~1 (remember?). The intercept in a logistic regression model tests whether the overall probability for response = 1 is significantly higher than the probability for response = 0, ignoring all of the independent variables.

If you think about this, you'll realize that this is exactly like the one-way chi-square test comparing two category sizes (here, the 1 vs. 0 categories), or like the binomial test. Let's analyze the same data three ways to see how the results differ.

So ... while doing fieldwork on Mars, you're trying to figure out a puzzle about the suffix *-qfx*. This appears about equally often across the hundreds of nouns in the dictionary you're compiling, with one curious exception: it seems to appear slightly more often than not on the 30 Martian words relating to cheese (Martians love cheese). Specifically, *-qfx* appears on 20 of the cheese nouns but not on the 10 other cheese nouns. Is this a significant asymmetry, or is the preference of *-qfx* for cheese just chance?

```
cheese = c(rep(1,20),rep(0,10)) # 20 1's and 10 0's
sum(cheese==0); sum(cheese==1) # Yup
```

The binomial test gives the exact probability of the null hypothesis that *-qfx* has no preference for cheese, just missing significance:

```
# two-tailed p = 0.09873715 (not quite significant!)
2*pbinom(min(sum(cheese==0),sum(cheese==1)), length(cheese), 0.5)
```

Now let's run a one-way chi-squared test, as an asymptotic estimate of the above (by the way, note how I count the number of 0s and 1s using **xtabs()**). It just misses significance.

```
chisq.test(xtabs(~cheese)) # X-squared = 3.3333, df = 1, p-value = 0.06789
```

To test the intercept-only model using logistic regression, we create the model with **glm()**, and then put this inside **summary()** to get the coefficients table, like so:

```
cheese.glm = glm(cheese~1, family="binomial") # The quotes are actually optional
summary(cheese.glm) # p = .0735: not significant again
```

```
Call:
glm(formula = cheese ~ 1, family = "binomial")

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.4823  -1.4823   0.9005   0.9005   0.9005

Coefficients:
             Estimate  Std. Error  z value  Pr(>|z|)
(Intercept)    0.6931      0.3873     1.79    0.0735
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 38.191  on 29  degrees of freedom
Residual deviance: 38.191  on 29  degrees of freedom
AIC: 40.191

Number of Fisher Scoring iterations: 4
```

So all three analyses give sort of similar results, though in this case, the **exact** binomial test is the most reliable, since this is such a small sample; the other two tests are **asymptotic** models, and so they work better in larger sample sizes.

2.3.2 Deeper links between logistic regression and the chi-squared distribution

R's text report for the logistic regression also mentions the **residual deviance**. **Deviance** is just a measure of how different two frequencies are. For example, in a chi-squared test, the test statistic χ^2 represents the **total deviance**, or the sum of all of the deviances for the expected and observed frequencies (which we want to be as big as possible, since the expected frequencies represent the null hypothesis). In a logistic regression, the residual deviance represents the deviance of your model compared with the observations. Thus unlike R^2 for ordinary linear regression, we want this measure of model fit to be as *small* as possible, since our model is supposed to fit our observations.

The residual deviance is related to the **log likelihood** (LL) of our model, in terms of the following formula:

Deviance = -2LL

Remember that likelihood here is the probability that our model is true given our data. Why do we use *log* likelihood? Because as a kind of probability, likelihood is also restricted to lie between 0 and 1, and the log makes it infinite at one end. Why do we multiply by -2? In part because ln(1) = 0 and ln(0) = $-\infty$, making worse fits negative, so multiplying by a negative number makes larger positive numbers imply a worse fit. In particular, if our model is a perfect fit, -2LL = 0 (no deviance); if our model isn't a perfect fit, -2LL > 0. But why -2 specifically? Because this measure relates to **information theory**, and a binary contrast (2) is the smallest unit of information (i.e., 1 only means something in contrast to 0).

We can see that the deviance and log likelihood values are related for the **cheese.glm** model by extracting them like so:

```
LL = logLik(cheese.glm); LL

'log Lik.' -19.09543 (df=1)

summary(cheese.glm)$deviance

[1] 38.19085

as.numeric(-2*LL) # Same as deviance

[1] 38.19085
```

In order to compute a logistic regression, programs like R are actually trying to maximize the log likelihood (and thus reduce the residual deviance), adjusting the coefficients each time it loops through the data until adjusting them doesn't make log likelihood any larger.

To get a sense of how this is computed, first look at the estimate for the intercept coefficient in the logistic regression: it's just the log odds associated with the probability that the output is 1:

```
mean(cheese) # The probability of cheese == 1
```

```
[1] 0.6666667
```

```
coef.int = summary(cheese.glm)$coefficients[,"Estimate"] # intercept coefficient only
coef.int
```

```
[1] 0.6931472
```

```
inv.logit(coef.int) # Turn log odds back into probability (in gtools package)
```

```
[1] 0.6666667
```

The *p* value shown in the logistic regression table comes from a two-tailed one-sample *z* test on the *z* value shown in the table:

```
summary(cheese.glm)$coefficients[,"Pr(>|z|)"] # Just the p value
```

```
[1] 0.07350237
```

```
logit.z = summary(cheese.glm)$coefficients[,"z value"] # Just z
2*pnorm(-abs(logit.z))
```

```
[1] 0.07350237
```

So where does this *z* value come from? Like the *t* value in ordinary (linear) regression, it is just the estimated coefficient divided by the standard error:

$z = \frac{B}{SE}$ for the coefficient *B*

```
logit.z
```

```
[1] 1.789699
```

```
logit.SE = summary(cheese.glm)$coefficients[,"Std. Error"] # Just standard error
coef.int/logit.SE
```

```
[1] 1.789699
```

The test R that uses to compute the z value is called the **Wald test** (named after Hungarian mathematician Abraham Wald [1902-1950]), which takes advantage of the relationship between the normal distribution and the chi-squared distribution (I mentioned this relationship in the chapter on chi-squared test). In the case of the Wald test, the crucial relationship looks like this (for $df = 1$):

$\chi^2 \approx \frac{B^2}{SE^2}$ for the coefficient B

You can confirm this yourself:

```
chisq.test(c(10,20))$statistic # Just the chi-squared value of the chi-squared test
```

```
X-squared
 3.333333
```

```
logit.z^2
```

```
[1] 3.203021
```

But why are we back to the stupid old z test? Didn't we learn that it makes unjustified assumptions about the population, which is why we started using the t test instead? Indeed, something funny is going on. While it's reasonable to test the usual null hypothesis that the population mean (here, coefficient) is zero, we don't actually know the population standard deviation, and this makes our standard error unreliable. Moreover, when the coefficient is large, the Wald test tends to make the standard error large as well, making their ratio (which gives z) less reliable as well.

So not only is logistic regression an asymptotic method, but it's one that relies on the z test (no df involved). That means that if you're going to do a logistic regression, it's best to use as large a sample as you can. A common rule of thumb (that also applies to ordinary linear regression) is to have at least 10 data points per predictor. Another, stricter, common rule of thumb is to have at least 100 data points (assuming you only have a small number of predictors, i.e., many fewer than 10).

2.3.3 Testing significance with likelihood ratios

Worries over the reliability of the Wald test has led to the search for alternative ways to compute the p values in a logistic regression. One of these is the same one we saw with ordinary linear regression, and though it is also an asymptotic method, it can be more stable than the

Wald test. This involves running a **likelihood ratio test**, comparing our model with one that's missing the parameter that we want to test.

Recall that for linear models, we used R's **anova()** function, and computed a kind of analysis of variance. For generalized linear models, we use exactly the same function, but tell it to run a chi-squared test comparing the model fits, using the argument **test = "Chisq"** (we have to capitalize the "C"; don't ask me why). So this time, the likelihood ratio that R computes here with **anova()** is actually a so-called **analysis of deviance**.

Here's how to use this approach to test the intercept in the intercept-only logistic regression model **cheese.glm**. The formula expressed as $y \sim 1$ is actually not the simplest possible regression; even simpler is $y \sim 0$, which predicts y from nothing! Does our intercept model do better than this no-intercept model?

```
cheese.glm0 = glm(cheese~0, family="binomial")
anova(cheese.glm0, cheese.glm, test="Chisq")
```

```
Analysis of Deviance Table

Model 1: cheese ~ 0
Model 2: cheese ~ 1
  Resid. Df  Resid. Dev  Df  Deviance  Pr(>Chi)
1        30      41.589
2        29      38.191   1     3.398   0.06528 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can report the results we get as $\chi^2(1) = 3.398$, $p > .05$. And look at that p value: $p = .06528$ is very close to the $p = .06789$ we got with the chi-squared test, showing once again the relationship between this old method and our fancy new logistic regression.

So far I've demonstrated *four* different ways to test exactly the same null hypothesis. The exact binomial test is the most reliable here, since the p value is based directly on probabilities from the observations. We can then compare this with the other three tests as in Table 1.

Table 1. Four ways to test Martian cheese

Method	*p* value
binomial test	.099
Wald test	.073
chi-squared test	.068
analysis of deviance	.065

Of course, in real life, you usually don't have much choice over which test to use. In Sally's rich deletion data set, for example, we're predicting deletion from income, a numerical variable, instead of just modeling the intercept. This makes the binomial and chi-squared tests irrelevant, since they only work with simple categorical independent variables. Our only real choice, then, would be logistic regression (with *p* values computed using the default Wald test, or via model comparison and analysis of deviance).

2.4 The problem of convergence

Suppose you look at those 30 Martian cheese nouns again and you notice something amazing: the *-qfx* suffix seems to be rarer in higher-frequency cheese words. You also suspect that maybe word length (in number of syllables) affects the probability of using the *-qfx* suffix too. Let's first create the fake data, but use column labels so we can run a full-fledged regression:

```
Suffixed = c(rep(1,20),rep(0,10)) # Dependent variable
Frequency = 1:30
```

Notice that by putting all the Suffixed words first in the list, we get a strong negative correlation between the variables Suffixed and Frequency. By the way, you might remember that this kind of correlation (between a vector of 0s and 1s and a numerical vector) is identical to an unpaired *t* test (try it: you should get *r* = -.82).

```
cor.test(Suffixed,Frequency) # This is just an unpaired t test, remember?
```

Now let's fake some word lengths, choosing them totally randomly using the **sample()** function (I first use **set.seed(1)** to make sure you and I get the same fake values):

```
set.seed(1) # So we get the same fake values
WordLength = sample(1:30)
```

Let's run a logistic regression on these exciting "facts", to see what predicts the probability (actually, log odds) of showing a suffix. Based on how we faked the data, we expect there to be a strong negative effect of Frequency on Suffixed, but no effect of WordLength.

```
cheese2.glm = glm(Suffixed ~ Frequency + WordLength, family = binomial)
```

```
Warning messages:
1: glm.fit: algorithm did not converge
2: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

This is a bad sign: R gives us two warnings. The first says that the model didn't **converge** (i.e., the algorithm failed to find a stable model before running out of loops), and the second says that the fitted probabilities were at the minimum or maximum values (0 or 1).

Oh well, a warning is just a warning, not a fatal error. Let's see what our model looks like anyway:

summary(cheese2.glm)

```
Call:
glm(formula = Suffixed ~ Frequency + WordLength, family = binomial)

Deviance Residuals:
          Min            1Q        Median            3Q          Max
 -0.000078088  -0.000000021   0.000000021   0.000000021  0.000074086

Coefficients:
              Estimate  Std. Error    t value     Pr(>|t|)
 (Intercept)   755.688 302590.587      0.002        0.998
 Frequency     -33.014  13296.782     -0.002        0.998
 WordLength     -6.309   5417.085     -0.001        0.999

(Dispersion parameter for binomial family taken to be 1)

     Null deviance: 38.190850097689    on 29    degrees of freedom
Residual deviance: 0.000000011877    on 27    degrees of freedom
AIC: 6

Number of Fisher Scoring iterations: 25
```

What? That doesn't make any sense. All of the *p* values are basically 1. This is not crazy in the case of WordLength, which is a totally random variable, but why is the intercept *p* value so far above .05, when in our intercept-only model it was pretty close to .05? And why on earth is the *p* value for Frequency so close to 1, when the correlation between Suffixed and Frequency was intentionally made so strong?

But it's not just the *p* values that are wrong: look at those crazy coefficients! In the intercept-only model, the intercept coefficient was 0.6931, which as we saw, was just the log odds for the probability of getting a suffixed word in this set (20/30). But now we get the totally ridiculous and impossible coefficient of 755.688! Moreover, the residual deviance is virtually zero, and R looped 25 times (as shown in the last line of the report), which is the default maximum (you can change it using the **control** argument; check **?glm.control** for details).

So the algorithm crashed. Why? Bizarrely, it's because the correlation between Frequency and Suffixed is *too* good: the optimization algorithm used by logistic regression cannot handle a perfect fit! Indeed, in our fake data, Frequency predicts Suffixed perfectly: Suffixed = 1 when

Frequency < 20, and Suffixed = 0 when Frequency > 20 (technically, this situation is called **complete separation**):

```
max(Frequency[Suffixed==1]) # 20
min(Frequency[Suffixed==0]) # 21
```

For a large enough real data set, this situation is not extremely common, since real life is kind of messy. But if it happens to you, there are a number of responses you could take to this problem. The simplest would be to say that no statistical analysis is really necessary here: if the sample size isn't tiny, complete separation of your output variable is itself a good argument that your model is describing a real pattern. If your thesis director or journal reviewer do not agree with such a reasonable response, you could see if you could use valid linguistic reasons to change your choice of independent variables, or to transform some of them. You could also try studying more technical solutions like some form of "penalized" regression, where the model adds a noisy variable (relevant functions are available in the R packages **glmnet** [Friedman et al., 2017] and **logistf** [Heinze et al., 2017]).

But to keep the chapter from getting too long, let's finish this example by adding a little bit of noise to the data ourselves, for example by moving one of the 1s from the start of Suffixed to the end, so now there's one low-frequency word that's suffixed, creating the even more fake vector Suffixed.f. Now we get no warning, and a reasonable-looking model:

```
Suffixed.f = c(Suffixed[2:30],Suffixed[1])
cheese3.glm = glm(Suffixed.f ~ Frequency + WordLength, family = binomial)
summary(cheese3.glm)
```

```
Call:
glm(formula = Suffixed.f ~ Frequency + WordLength, family = binomial)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.23479  -0.40883   0.07562   0.27547   2.73506

Coefficients:

              Estimate   Std. Error   z value   Pr(>|z|)
 (Intercept)   7.74575    3.32075      2.333    0.01967   *
 Frequency    -0.39387    0.14973     -2.63     0.00853   **
 WordLength    0.02213    0.08701      0.254    0.79925
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 38.191  on 29  degrees of freedom
```

```
Residual deviance: 16.032    on 27    degrees of freedom
AIC: 22.032

Number of Fisher Scoring iterations: 6
```

Now the intercept has a significant positive effect as it should, since there are more 1s than 0s. Frequency has a significant negative effect, since higher frequency means fewer 1s. WordLength has no significant effect, since it's random.

The only problem is that the coefficient for the intercept is still too high (remember that in reality it should be 0.6931), because our sample size is still too small to get completely reliable results. We can get a hint that our model is not perfect by looking at another part of R's text report, namely the **deviance residuals**. As in ordinary linear regression, these should be normally distributed (and thus symmetrically distributed) around 0, but you can see that the minimum (-1.23479) and maximum (2.73506) are not mirror images of each other: the distribution is not symmetrical.

Another clue that we may need to check our model more carefully is suggested at the end of the report, where we are told that the "Dispersion parameter for binomial family taken to be 1". **Dispersion** reflects an assumption about the variability in our data that's made by logistic regression, namely that the ratio of 0s and 1s will be consistent with an actual binomial distribution, and not some other type of distribution. If there's less variation in the data than the model assumes (i.e., less than 1), that's called **underdispersion**, and if there's more, that's **overdispersion** (which is the more common situation with real data)..

Should we worry about this? People generally don't, just as they don't worry too much about other technical assumptions about statistical tests (and we've already seen a lot of those throughout this book). If you're worried anyway, the easiest way to test if the dispersion assumption is valid for your data is is to use the **testDispersion()** function in the **DHARMa** package (Hartig, 2022; the name stands for "Diagnostics for Hierarchical (Multi-Level / Mixed) Regression Models", where the terms "hierarchical", "multi-level", and "mixed" will be explained in the next chapter). It works by resampling new (fake) samples to see how weird the real data are.

```
library(DHARMa) # You have to install it first (it comes with lots of stuff)
testDispersion(cheese3.glm, plot = F) # Default plot = T shows fake sample histogram

        DHARMa nonparametric dispersion test via sd of residuals fitted vs. simulated

data:   simulationOutput
dispersion = 0.85633, p-value = 0.792
alternative hypothesis: two.sided
```

So in this case we're just fine: with $p > .7$, we can safely assume that our data shows the expected levels of dispersion, even though the actual dispersion value (0.86) isn't exactly 1.

Finally, notice that the **glm()** report also includes **AIC**, that universal measure of model fit, the **Akaike Information Criterion**. Remember that the better the fit, the smaller this number should be. The number here, 22.032, is above zero, which means it's not as small as an ideal model would be (since AIC values can be negative, and it's the lowness that matters, not the magnitude).

2.5 Plotting a logistic regression

Since we plotted Sally's rich deletion data incorrectly before, let's do it again, properly this time. Here's her data and model if you lost them:

```
rd = read.delim("richdeletion.txt")
rd.glm = glm(Deletion~Income, family = "binomial", data = rd)
```

One way to plot logistic regression is to show the raw 0s and 1s for the dependent variable, as we did before, but instead of adding a linear fit line, we use the sigmoid curve from our logistic regression model. We can do this with the **predict()** function that we used before with linear regression. The only new thing here is that for **glm** models, **predict()** uses the **canonical** (default) link function by default for whichever generalized linear model you specific via the **family** argument; when you set **family=binomial**, it predicts the logit values by default. So if we want to predict the original 0 vs. 1 response values, we have to set **type="response"** (see **?predict.glm** for more info).

Here's how I did this to make Figure 4. The most annoying parts of the code below are caused by the fickle **curve()** function. The **predict()** function needs a data frame to predict from, but the **curve()** function needs to treat these variables as a continuous input *x* value so it can plot the corresponding *y* values in the curve. The result is the weird use of **data.frame(Income = x)**, which in turn forces us to use **attach(rd)** first so Income is in the general memory for the benefit of **curve()**. We also need the **add=TRUE** argument, since by default **curve()** creates new plots instead of adding lines to existing plots.

```
attach(rd) # Pleases the fickle curve() function
options(scipen=5) # Forces R to display 5-digit numbers
plot(Deletion ~ Income)
curve(predict(rd.glm, data.frame(Income = x), type="response"), add=T)
detach(rd) # Always do this as soon as you're done
```

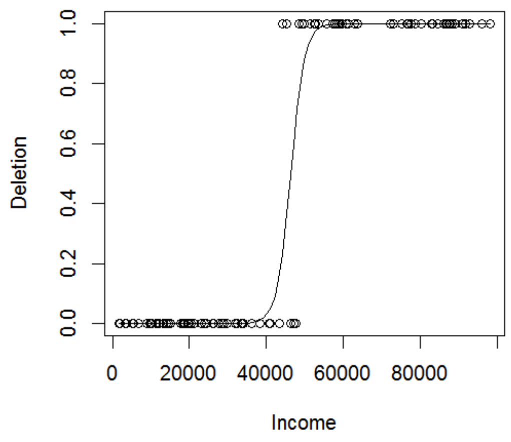

Figure 4. A sigmoid logistic fit to our raw 0s and 1s

How should we interpret such a sigmoid logistic fit line? Well, the "intercept" is indicated by the position of the rising part of the curve along the *x*-axis; here it's about in the middle, since Deletion = 0 and Deletion = 1 have almost the same frequency. The "slope" is indicated by the slope of this rising part: it's going up, so here it's positive.

However, this kind of plot has some important disadvantages. First, the dots tend to overlap, so it's often hard to tell where they cluster. For example, if there is more than one data point with the same *x* and *y* values (not uncommon, since *y* can only be 0 or 1), all those dots will overlap: one isolated dot looks exactly the same as a lot of dots all at the same point. Second, the dots always appear at only two places along the *y*-axis, so we don't get a clear sense of how the probabilities actually change as a function of *x*. Third, the sigmoid fit line doesn't show the model's intercept and slope directly.

An alternative that solves the first two problems is to convert the *y*-axis values to probabilities for several **bins** along the *x*-axis. These probabilities represent what the sigmoid curve is ultimately trying to fit. So here's how to create Figure 5 (note the change in the *y*-axis label, since we're now plotting rates or probabilities, not raw data):

```
attach(rd)
bins = cut(Income,10) # Creates factor defining 10 equal-sized bins for x
meanx = tapply(Income, bins, mean) # Computes mean for each bin
proby = tapply(Deletion, bins, mean) # Mean of 0s & 1s is probability, remember?
options(scipen=5) # Sorry, we have to do this again...
plot(meanx, proby, ylab = "Deletion rate", xlab = "Income")
curve(predict(rd.glm, data.frame(Income=x), type="resp"), add=T)
detach(rd)
```

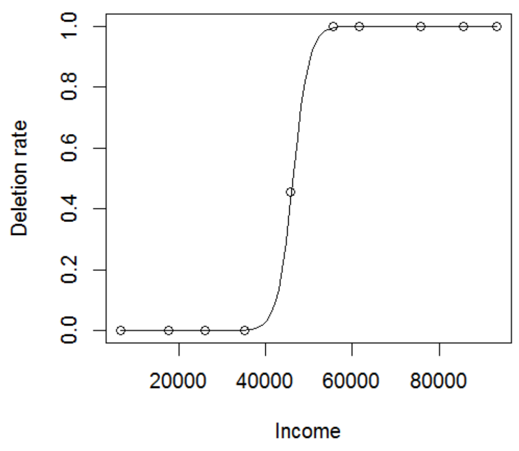

Figure 5. A sigmoid fit to probabilities

Another alternative that solves all three problems (but creates new problems of its own) is to convert the y-axis values into log odds (logits) for several bins along the x-axis. Now we can plot the fit line using ordinary linear regression, since we're applying the canonical link function directly in the graph. But by using log odds on the y-axis, we've lost an intuitive picture of the original probabilities. Computing logits for some bins may also require fudging the data a bit, since if any bin has no cases of $y = 0$ or $y = 1$, then P_0 or P_1 will be zero, making $ln(P_1/P_0)$ impossible to plot (∞ if $P_0 = 0$ or $-\infty$ if $P_1 = 0$). So in the R code I used to create Figure 6, I had to add some noise.

```
attach(rd)
bins = cut(Income, 10)
# tapply needs a one-argument logit function for vectors:
logit.bin = function(vector) {
  prob1 = mean(c(vector,0,1)) # Add noise: two extra values
  prob0 = 1-prob1 # probability of y = 0 in bin
  log.odds = log(prob1/prob0) # So neither prob should be 0
  return(log.odds)
}
meanx = tapply(Income, bins, mean) # Computes mean for each bin
logity = tapply(Deletion, bins, logit.bin)
plot(meanx, logity, ylab = "Deletion (log odds)", xlab = "Income")
abline(lm(logity ~ meanx)) # Ordinary linear regression to fit to log odds
detach(rd)
```

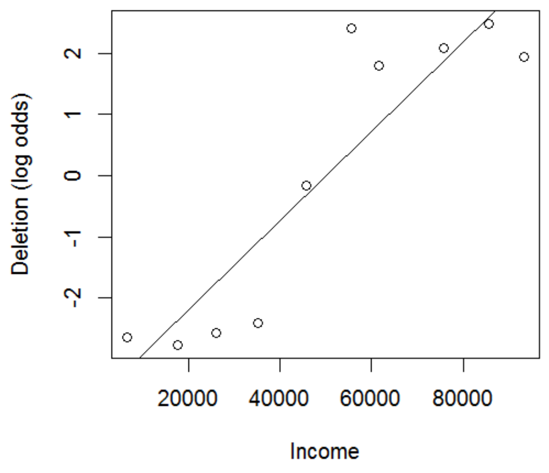

Figure 6. A linear fit to log odds

You don't have to do all this work by hand. The **effects** package essentially does a combination of the last two options: as shown in Figure 7, it plots a linear fit to the log odds, but it rescales the *y*-axis to display the probabilities (resulting in *y*-axis numbers that are not evenly spaced).

```
library(effects)
plot(allEffects(rd.glm))
```

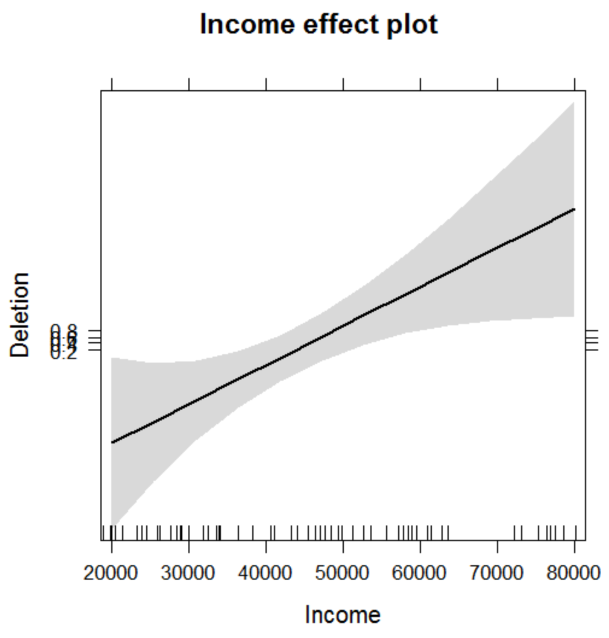

Figure 7. A linear fit to log odds, rescaled to probabilities

Note that the plot created with the **effects** package also surrounds the best-fit line with a gray envelope representing the 95% **confidence band** (this package performs the same service when plotting ordinary linear regression, yet another interesting thing that we had to leave out of our over-long multiple regression chapter). Notice that the band here does not show an identical distribution of variability all across the data set. which seems consistent with how the binned probabilities are scattered in Figure 6 as well. In particular, the model is more reliable in the middle range of values (where the confidence band is thinner). This is typical for logistic regression; indeed, as we saw, the whole algorithm crashes when our data set is too extreme (perfect).

2.6 Effect sizes for logistic regression

As we've seen, the overall fit of a logistic regression model can be expressed with residual deviance and AIC. Another way is to attempt to mimic the **coefficient of determination** R^2 that we can easily compute for ordinary linear regression. We can't use exactly the same method to compute this for logistic regression, since now the dependent variable of 0s and 1s isn't a "really" a numerical vector, but we can do something similar. Menard (2000) suggests computing what he calls $R_L{}^2$, based on the log likelihood of our full model as compared with an intercept-only model. Namely, you compute $R_L{}^2$ as the ratio of the difference in log likelihoods for the empty model LL_{empty} and the full model LL_{full}, divided by the empty model:

A kind of coefficient of determination: $$R_L{}^2 = \frac{LL_{empty} - LL_{full}}{LL_{empty}}$$

It's easy to compute this formula in R, using the **logLik()** function again (below I put it inside the **as.numeric()** function to get rid of the text that **logLik()** adds). The value we get implies that about 61% of the variability in the responses is explained by our model.

```
cheese0.glm = glm(Suffixed.f ~ 1, family = "binomial")
as.numeric((logLik(cheese0.glm)-logLik(cheese3.glm))/logLik(cheese0.glm))
```

```
[1] 0.5802248
```

If you want, you could also compute $R_L{}^2$ from the **summary(glm(...))** object. Remember that the text report given by this command shows the residual deviance, which is the log likelihood for the full model divided by -2. The report also shows the **null deviance**, which is the same as for the intercept-only model (which is why the two values are identical for the first cheese analysis we did, back in section 2.3). Thus we can also compute $R_L{}^2$ like so:

```
cheese3.LL = -2*summary((cheese3.glm))$deviance
cheese0.LL = -2* summary((cheese3.glm))$null.deviance
(cheese0.LL- cheese3.LL)/cheese0.LL
```

```
[1] 0.5802248
```

How do we quantify the effect sizes for the individual independent variables? One simple way is to adapt the method we used for ordinary linear regression, and compute something similar to **standardized coefficients**. Remember that for ordinary linear regression, we can do this by multiplying each predictor's coefficient by the ratio of that predictor's standard deviation divided by the standard deviation of the independent variable:

Standardized coefficients for ordinary linear regression: $\beta_i = b_i \left(\frac{s_{x_i}}{s_y}\right)$

Remember also that we could get the exact same results by first converting all of our variables, including the dependent variable, into *z* scores, and then running the linear regression on these *z* scores.

However, neither method works perfectly for logistic regression, since now the model is dealing with log odds, not the raw dependent variable of 0s and 1s. Indeed, if you try to compute the standard deviation for logit(Y) (where Y is your vector of 0s and 1s), you get nonsense, since by definition, logit(0) = -∞ and logit(1) = +∞.

The easiest way around this problem is to "standardize" only the independent variables, by converting them to *z* scores before running the regression. This will allow you to compare effect sizes within a model, but not necessarily between models, since this method doesn't take the variance of the dependent variable into account. In fact, this variance will change as you add or remove predictors from your model (which is part of the reason why the intercept coefficient changed so much when we went beyond our intercept-only analysis of the Martian cheese data). It's still a pretty commonly used approach, however (see, e.g., Myers, 2016).

Let's try it on the slightly-noisy Martian cheese data (predicting Suffixed.f from Frequency and WordLength). As when we used this trick with ordinary linear regression, the *p* values come out exactly the same as before, but now we get the (sort of) standardized coefficients of $\beta_{Frequency}$ = -3.4674 and $\beta_{WordLength}$ = 0.1948 (try it!).

```
cheese3.z.glm = glm(Suffixed.f ~ scale(Frequency) + scale(WordLength),
  family = "binomial")
summary(cheese3.z.glm)
```

Also like ordinary linear regression, we can test if the two coefficients are significantly different using a likelihood ratio test comparing the full model with one where the predictors are literally summed (with **I()**). Unsurprisingly, they're indeed significantly different (try it!):

```
cheese3.z.same.glm = glm(Suffixed.f ~ I(scale(Frequency) + scale(WordLength)),
  family=binomial)
anova(cheese3.z.same.glm, cheese3.z.glm, test="Chisq")
```

If you want to try to take the dependent variable into account too, Menard (2004) suggests a method that estimates the "observed" logits (which you can't literally observe, since they're -∞ and +∞, as just explained) using the predicted logits, which **predict(glm(...))** can give you (by keeping the default argument **type="link"**). His method also exploits the fact that the formula for computing standardized coefficients β in ordinary linear regression is related to the formula for the coefficient of determination R^2. As we saw above, the same author suggests using R_L^2 as the logistic regression equivalent of R^2. All of this reasoning leads to him suggest the following formula, where R_L is the square root of R_L^2, and $s_{logit(\hat{y})}$ is the standard deviation of the logits predicted by your model.

Menard's standardized coefficients for logistic regression: $\beta_i = b_i R_L \left(\frac{s_{x_i}}{s_{logit(\hat{y})}}\right)$

To apply this formula to standardize the coefficients in **cheese3.glm** (created with the raw variables, not the *z* scores), we can use the following code. It gives very different values from the simpler method above, but the difference in effect size for the two variables is still in the same direction (i.e., the magnitude of the Frequency effect is much greater than that for the WordlLength effect), and the ratio of the two standardized coefficients is the same using either method:

```
Frequency.coef = summary(cheese3.glm)$coefficients["Frequency","Estimate"]
WordLength.coef = summary(cheese3.glm)$coefficients["WordLength","Estimate"]
RL = sqrt(as.numeric((logLik(cheese0.glm)-logLik(cheese3.glm))/logLik(cheese0.glm)))
Frequency.sd = sd(Frequency)
WordLength.sd = sd(WordLength)
logit.yhat.sd = sd(predict(cheese3.glm)) # logits, not prob (type = "link" by default)
Frequency.beta = Frequency.coef * RL * (Frequency.sd/logit.yhat.sd)
WordLength.beta = WordLength.coef * RL * (WordLength.sd/logit.yhat.sd)
Frequency.beta; WordLength.beta
```

```
[1] -0.7674511
[1] 0.04311755
```

3. Extending logistic regression

Just as with ordinary linear regression, there's a lot more to say about logistic regression. Some of it involves things we've already seen with ordinary linear regression. For example, when running a logistic regression, you should still test for collinearity and try to deal with it if you find it. You may also want to try doing a stepwise regression, since the **step()** function works just as well for **glm** objects as for **lm** objects (though I don't recommend it, for the reasons I mentioned in the previous chapter). Your independent variables can also include multi-level categorical factors, encoded using dummy coding or effect coding. You can also test for interactions among the independent variables. It is also possible to handle repeated-measures data.

Here I'll address just one of these old issues, namely working with repeated-measures data. The rest of this section then turns to new issues, namely: multinomial logistic regression (when your dependent variable involves more than two categories) and Varbrul (which can do a variety of logistic regression methods, albeit in a rather non-standard format).

3.1 Repeated-measures logistic regression

Like ordinary linear regression or chi-squared tests, a logistic regression model assumes that each data point is independent of all of the others. However, the logic we used earlier for repeated measures (linear) regression can also be applied in **repeated-measures logistic regression**. Similar to the previous chapter, all we have to do is run a logistic regression on each unit (e.g., each participant in an experiment, or each speaker in a sociolinguistic database), and then run one-sample *t* tests on the coefficients across the units. This works because logistic regression coefficients, just like coefficients in ordinary linear regression, tend to be normally distributed (Pampel, 2000).

Let's see how this works, since in the next chapter we'll compare the results with mixed-effects logistic regression, which is a better way to analyze the same kind of data.

The data in **demo.txt** are the results of a real syntax experiment (for similar experiments, see Myers, 2009, 2012a), where seven Mandarin speakers were each given 20 sentences to judge as good (Judgment = 1) or bad (Judgment = 0). The sentences came in sets of four that were matched as much as possible, except for two factors: ComplexNP (1 = complex noun phrase, -1 = simple noun phrase) and Topic (1 = element extracted from the noun phrase to topic position, -1 = no extraction). Our main interest is testing whether there is an interaction between ComplexNP and Topic, which could mean that it's ungrammatical to extract topics from complex noun phrases.

First, here's the data:

demo.dat = read.delim("demo.txt")
head(demo.dat)

```
  Speaker  Item   Set  Order ComplexNP  Topic Judgment
1       1    18     5      1         1     -1        1
2       1     5     2      2         1      1        1
3       1    20     5      3        -1     -1        1
4       1    19     5      4        -1      1        1
5       1     4     1      5        -1     -1        1
6       1     2     1      6         1     -1        1
```

Note that I've already used effect coding in the data frame, which makes the interaction (if it exists) easier to interpret (it also makes the **effects** package annoying to use for plotting, but we won't be plotting anything).

So let's write some R code to create a logistic regression model on each speaker; this is basically the same code we used in the previous chapter to run repeated-measures regression, but modified for logistic regression.

```
CNP.coef = numeric(7) # Will hold ComplexNP coefficients across participants
Top.coef = numeric(7) # Will hold Topic coefficients across participants
CxT.coef = numeric(7) # Will hold interaction coefficients across participants
for (i in 1:7) { # Run logistic regressions for each participant
  demo.dat.i = subset(demo.dat, demo.dat$Speaker==i) # Participant i's data
  glm.i = glm(Judgment~ComplexNP*Topic, family=binomial, data=demo.dat.i)
  CNP.coef[i] = summary(glm.i)$coefficients["ComplexNP","Estimate"]
  Top.coef [i] = summary(glm.i)$coefficients["Topic","Estimate"]
  CxT.coef [i] = summary(glm.i)$coefficients["ComplexNP:Topic","Estimate"]
}
```

Now we'll run one-sample *t* tests on each of the three coefficients:

```
t.test(CNP.coef)$p.value # Let's just look at the p values...
```

```
[1] 0.02073655
```

```
t.test(Top.coef)$p.value # ... computed using one-sample t tests...
```

```
[1] 0.000008145483
```

```
t.test(CxT.coef)$p.value # ... just as we did in the previous chapter
```

```
[1] 0.02073655
```

The results show that all three parameters are significant, all with negative effects: complex noun phrases reduce acceptability, topicalization reduces acceptability, and both together is especially bad (an interaction, so to be sure of what's going on, we should also plot the interaction, but I'll skip that here). The fact that the two main effects are also significant shows that we're merely testing acceptability, not grammar directly: complex noun phrases and topicalization may be rare-ish or harder to process, but they're not ungrammatical in Chinese!

But what about the "language as fixed effect fallacy"? Shouldn't we also run a by-items analysis? Maybe not in this case, since the sentences were in matched sets, and as I mentioned in an earlier chapter, Raaijmakers et al. (1999) argue that we don't need to run by-item analyses if we match our items well enough. But even if we did run a by-items repeated-measures logistic regression, there is nothing like *minF'* to put it together with the by-participants analysis. To take both participants and items into account, we need to use **mixed-effects logistic regression** (which we'll introduce in the next chapter).

To end this section, let me note a clever application used by Xu et al. (2006) (who actually used mixed-effects modeling, but the logic works the same way). Xu and his friends took advantage of the fact that the sigmoid logistic function looks a lot like what you see when you give people a series of physical stimuli (like speech sounds) that start firmly in category A and gradually shift until they end up firmly in category B, and ask the people to decide if each stimulus is in category A. Xu et al. used this experimental method to study tone categories.

Now, for some types of stimuli (like speech sounds), people do not perceive the gradual shift, but instead experience categorical perception, a nonlinear change from one category to another, with only a few ambiguous items in between. In other words, if you ask people to decide if each stimulus belongs to category B, you end up with an S-shaped curve for the probability of saying "yes": flat at the top (mostly "no"), then a rise (the ambiguous portion), and then flat on the bottom (mostly "yes", since now you're in the area of category B). In other words, the identification curve in a categorical perception experiment looks a lot like Figure 5 above.

To analyze categorical perception using repeated-measures logistic regression, the dependent variable is 1 when the participant perceives category A, and 0 otherwise, and there is only one independent variable: the stimulus along the gradient continuum (which we can treat as an ordered series of numbers). Each person will thus give us a series of 0s and 1s, and our total data set will consist of many such series, each from a different person. Then we can run a separate logistic regression on each person, computing that person's intercept (where the rising part of the curve appears along the *x*-axis) and slope (how vertical this rising portion is), and then run one-sample *t* tests across the participants' intercepts and slopes. The overall intercept then tells us where people typically divide up the continuum, and the slope tells us how sharp (vertical) their categorical boundary is.

3.2 Multinomial logistic regression

What if our dependent variable has more than two categories? For example, as is well known, in the Martian language, different nouns require different classifiers depending on the noun's semantic features. So the words for "snake" and "river" take the classifier *tiao* because they are oblong and flexible, even though *tiao* nouns differ in many other ways (e.g., snakes are animals and rivers aren't). Martian also has two other "oblong" classifiers, namely *gen* for oblong thin objects like needles, and *zhi* for oblong round objects (round in diameter, i.e., cylindrical) like pens. All three classifiers face tricky situations, though; e.g., *tiao* can also be used with dogs and hearts, for which oblongness is not the most obvious feature. In this way, Martian is extremely similar to a certain Earth language that we all know and love, except that it has the advantage that it's fake so I can make up a large enough data set to make the regressions work.

How can we model classifier choice in terms of semantic features? On the one hand, this looks like a job for logistic regression: we have a categorical dependent variable (classifier choice), and we want to predict it in terms of predictor variables (in this case, binary semantic features). But on the other hand, our dependent variable is not binary, but has three possible values: *tiao*, *gen*, or *zhi*.

Don't worry! We can use **multinomial logistic regression**. Let's first discuss the basic idea, and then look at some special applications.

3.2.1 Multinomial logistic regression: the basic idea

We start by encoding the semantics of the nouns using some binary features that we think might be relevant, like flexibility, thinness, and roundness. We'll just use dummy coding, since we're not going to test for interactions, so the variable Flexible has the value 1 for flexible objects and 0 for non-flexible objects, and so on. We'll also code the three classifiers as 1, 2, and 3 (we're not implying that they go in any particular order; these are just arbitrary labels).

The coded semantic features for 100 Martian nouns, and their preferred classifiers, are shown in the file **classifiers.txt**:

```
classifiers = read.delim("classifiers.txt")
```

Let's start by doing a series of binary logistic regression analyses to predict, separately, just the choice of classifier 1 (*tiao*), classifier 2 (*gen*), and classifier 3 (*zhi*). To use **glm(... family = "binomial")**, the dependent variable must be binary, so let's divide our three-level factor Class into three two-level factors:

```
classifiers$tiao = 1*(classifiers$Class== 1)
```

```
classifiers$gen = 1*(classifiers$Class== 2)
classifiers$zhi = 1*(classifiers$Class== 3)
```

Then we show the three coefficients tables:

```
summary(glm(tiao ~ Flexible + Thin + Round, family = "binomial", data = classifiers))
summary(glm(gen ~ Flexible + Thin + Round, family = "binomial", data = classifiers))
summary(glm(zhi ~ Flexible + Thin + Round, family = "binomial", data = classifiers))
```

The results appear as in Table 3, focusing just on the feature coefficients and using stars to indicate significance (* means $p < .05$, ** mean $p < .01$, *** means $p < .001$). Since we're running three non-independent analyses on the same data (e.g., *tiao* ==1 implies *gen* == 0 and *zhi* == 0), perhaps we want to use the Bonferroni adjustment, and only count effects as significant if $p < .05/3 = .017$; that would show the same significance pattern for the three features, since all have $p < .01$.

Table 3. Three separate binary logistic regressions predicting three different classifiers

	tiao B	*gen B*	*zhi B*
Flexible	4.7943 ***	-3.0106 ***	-3.9869 ***
Thin	-1.2262	2.7192 ***	-1.9633 **
Round	0.2616	-0.6745	0.4441

For *tiao*, the positive coefficient for Flexible means that this classifier is associated with this feature. For *gen*, the positive coefficient for Thin means that this classifier is associated with this feature, but the negative coefficient for Flexibility also means that it is also associated with rigid (i.e., non-flexible) objects. For *zhi*, the feature Round turns out not to be significant, but instead the classifier has negative associations with Flexible and Thin (i.e., it goes with rigid, thick objects).

Obviously it would be a lot more satisfying if we could test all three classifiers at the same time. We can't do it with **glm()**, though; it gives us a fatal error, not merely a warning:

```
summary(glm(Class ~ Flexible + Thin + Round, family = "binomial", data = classifiers))
```

```
Error in h(simpleError(msg, call)) :
  error in evaluating the argument ‘object’ in selecting a method for function ‘summary’: y values must be 0 <= y <= 1
```

But R being R, there are other packages available that can do just what we want. One is the **nnet** package (Venables & Ripley, 2002; I'll explain the weird name later), which contains the function **multinom()**.

Let's try it on the binary dependent variable *tiao* first, just to get a feeling for its output report (binary logistic regression is just a special case of multinomial logistic regression). As usual, the most useful information is found by applying the **summary()** function to the **multinom** object (you don't need to specify the family, since this function is only for models of this type):

```
library(nnet) # You have to install it first
summary(multinom(tiao ~ Flexible + Thin + Round, data = classifiers))

# weights:  5 (4 variable)
initial  value 69.314718
iter  10 value 32.924678
iter  10 value 32.924678
final  value 32.924678
converged
Call:
multinom(formula = tiao ~ Flexible + Thin + Round, data = classifiers)

Coefficients:
               Values    Std. Err.
 (Intercept)  -1.79792   0.716416
 Flexible     4.794361   0.857558
 Thin         -1.22622   0.825574
 Round        0.261588   0.732752

Residual Deviance: 65.84936
AIC: 73.84936
```

The first part of the report tells us how it ran the optimization algorithm to find the best model, looping ("iter" = iterating) until it "converged". The last part tells us about model fit, showing our old friends residual deviance and AIC. As you can confirm, these are the same values reported by **summary(glm(tiao ~ ...))**. The coefficients report also matches the same values we got for *tiao* using **glm()** (compare with Table 3); it also reports the same standard errors (as you can confirm by yourself).

It doesn't give significance values, though, for three reasons. First is a secret reason I'll tell you later. Second, as we saw, even statisticians don't agree on the best way to compute significance for logistic regression; not everybody likes the Wald test. Third, if you want to use the Wald test to compute *p* values, you still can, just by dividing the estimates by the standard errors to get *z* values, and then finding the two-tailed *p* values from those. For example, to compute the *z* and *p* values for the intercept, do this (compare them with **summary(glm(tiao ~ ...))**):

```
int.coef = -1.79792
int.se = 0.716416
int.coef/int.se # z = -2.509603
2*pnorm(-abs(int.coef/int.se)) # p = 0.01208668
```

OK, now that we know that **multinom()** is doing what it's supposed to do for a known case, let's try it on a new case, namely our three-level factor Class:

```
summary(multinom(Class ~ Flexible + Thin + Round, data = classifiers))
```

```
# weights:  15 (8 variable)
initial  value 109.861229
iter  10 value 55.775753
final  value 55.758966
converged
Call:
multinom(formula = Class ~ Flexible + Thin + Round, data = classifiers)

Coefficients:
  (Intercept)  Flexible      Thin     Round
2    0.238963  -4.47156  2.542851  -0.63955
3     1.41512  -5.09572  -0.25016  0.053477

Std. Errors:
  (Intercept)  Flexible      Thin     Round
2    0.838664  1.025919  0.957109   0.82217
3    0.777262  1.152775  0.967931  0.819323

Residual Deviance: 111.5179
AIC: 127.5179
```

Since our dependent variable could have any number of levels, the text summary arranges the table in an expandable way, arranging the independent variables horizontally and using separate tables for the coefficients and the standard errors. Each line of these two tables gives values related the comparison of the indicated level with the reference level of 1 (the lowest value in our labels 1, 2, 3). Again, we can use the reported *B* and *SE* values to compute $z = B/SE$ and thus *p*. Typing **?summary.multinom** doesn't say this (R help is often unhelpful), but by using **attributes()** on my object I learned that we can extract the *B* and *SE* values by using the **coefficients** and **standard.errors** attributes. Since we have two comparisons (level 2 vs. level 1, and level 3 vs. level 1) and four parameters (intercept plus three predictors), each of these is a two-by-four matrix, and since R knows how to apply functions across the cells of a matrix, we can compute two-by-four matrices of *z* values and *p* values relatively easily:

```
results = summary(multinom(Class ~ Flexible + Thin + Round, data = classifiers))
coef.vals = results$coefficients
se.vals = results$standard.errors
z.vals = coef.vals/se.vals
p.vals = 2*pnorm(-abs(z.vals))
z.vals
```

```
  (Intercept)  Flexible      Thin     Round
2    0.284932  -4.35859  2.656805  -0.77788
3    1.820648   -4.4204  -0.25844   0.06527
```

p.vals

```
  (Intercept)  Flexible      Thin     Round
2    0.775696  1.31E-05  0.007889  0.436638
3     0.06866  9.85E-06  0.796065  0.947959
```

Now we can use these *p* values as-is; there's no need to do a Bonferroni correction, since they're all computed from the same model. The feature Round is still useless (*p*s > .4) The feature Thin is associated significantly more strongly with classifier 2 (*gen*) than with classifier 1 (*tiao*), with $z > 0$ and $p < .01$. Both *gen* and classifier 3 (*zhi*) disfavor the feature Flexible, with *z*s < 0 and $p < .001$.

3.2.2 Multinomial logistic regression and computational modeling of language

It turns out that multinomial logistic regression is quite commonly used in linguistics, though this is not always obvious since it shows up in different subdisciplines under very different names. Here I'll discuss two examples of such applications.

First, theoretical linguists have become interested in using multinomial logistic regression to build models of grammar learning, except they don't call it this; instead, they call such models **maximum entropy models**, or **MaxEnt** for short (see, e.g., Hayes & Wilson, 2008), or sometimes **loglinear models** (which includes any generalized linear model involving logarithms).

Entropy (熵) is a metaphor from physics, where it refers to how "disorganized" a system is (for example, water molecules in ice have low entropy, while water molecules in steam have high entropy). In **information theory**, entropy refers to how "disorganized" information is, in the sense of how unpredictable each piece is. For example, a text in human language will have low entropy (since the words are partially predictable by the grammar), while a list of random words will have high entropy. Another way to put it is that when the entropy of a text is high, each word represents a lot of information, since none is redundant (e.g., you have to memorize each word in a random word list).

What about **maximum entropy**? Again going to the physics metaphor, imagine a complex ice sculpture (low entropy) sitting on a stove. It first turns into a puddle of water (higher entropy), but the puddle still has a "shape". When the water boils away into steam (maximum entropy), each water molecule is equally distant from all of the others. As in any optimization problem, the actual maximum depends on the constraints on the system. For example, if the stove is in a closed kitchen, the water molecules may spread out through the whole room, but the steam cloud still has a shape (i.e., the room). The importance of a constraint is quantified as its **weight** (the same metaphor used in the word 重要).

The looping algorithm for computing the predictor weights in MaxEnt is similar to how the water molecules in steam spread out evenly, within the constraints of the room. In this case, the constraints come from the probabilities of the various outputs, and from the values of the predictor variables. Aside from these constraints, the weights should be given values that distribute the probabilities as evenly as possible.

But it's already been proven (Berger et al., 1996) that the MaxEnt algorithm gives exactly the same results as multinomial logistic regression, and that the weights you get are identical to the statistical model coefficients. Indeed, we've already seen this with our own eyes: **glm()** uses a logistic regression algorithm, **multinom()** uses MaxEnt, but they end up giving exactly the same weights/coefficients in the cases where their powers overlap (binary logistic regression).

This leads to a second application of multinomial logistic regression: (artificial) **neural networks**, also known as **connectionism** (聯結主義). In fact, the name of the **nnet** package is short for "neural network". An artificial neural network is a highly simplified computer model of the human brain. Just as a real brain is built out of millions of brain cells that are individually stupid, but are connected with so many other brain cells in such a complex way that the overall brain becomes smart, so an artificial neural network is built out of simple **node**s that are connected to other nodes, encoding the information not in smart nodes but in clever network patterns. In particular, when a neural network is being trained to give the right outputs when presented with a set of inputs, the network "learns" by changing the **weights** of the connections between nodes. For example, if during training, output node 23 is supposed to be active whenever input node 61 is active, then the weight of the connection between nodes 61 and 23 will become more positive, and if output node 23 is supposed to be suppressed whenever input node 97 is active, then the weight of the connection between these nodes will become negative.

To see how this relates to logistic regression, we can install another R package that draws the mini-brain for us. First, we load the necessary package:

```
library(neuralnet) # You have to install it first
```

Since the algorithm starts by assigning random weights to all the connections, let's use **set.seed(1)** to make sure that you and I get the same results:

set.seed(1) # So we get the same results

Now we use the **neuralnet()** function to define our network, with three input nodes (one per semantic feature) and three output nodes (one per classifier). The syntax looks like an ordinary formula, except that there are three output nodes, so the dependent variable is written as a sum of variables. We also have to tell the function that we only want two layers in the network, namely the input and output, without any "hidden" layers in between (which the algorithm would adjust by itself, depending on the input and output and optimal connection weights). We also need to tell it how many times to train the input-output connections, adjusting weights little by little (**rep** for "repetitions).

train.net = neuralnet(tiao + gen + zhi ~ Flexible + Thin + Round, data = classifiers,
hidden = 0, rep = 5)

Let's look at the mature (**"best"**) network, as shown in Figure 8:

plot(train.net, rep = "best")

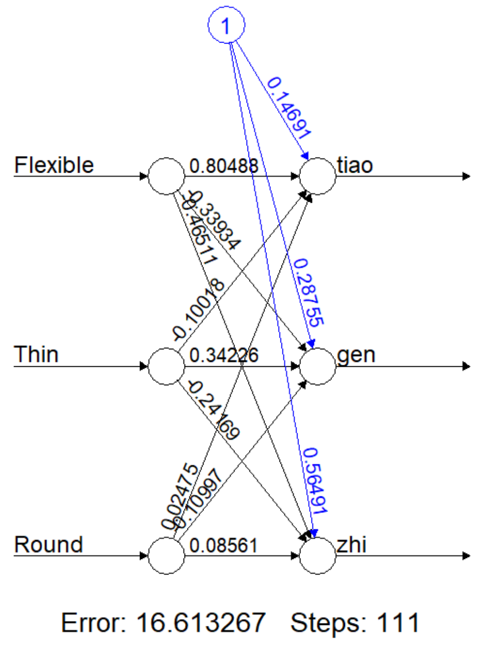

Figure 8. A neural network for Martian classifiers

The input nodes are on the left, and the output nodes are on the right. These six nodes are connected to each other in all possible ways, and each connection has a positive or negative weight. There is also a light-colored (blue) node with connections to the outputs; this is just a technical trick needed to make the algorithm work (since this node has a fixed value of 1, it works kind of like the intercept in logistic regression).

Note the connection weights on the dark (black) lines. They don't match the coefficients we got for logistic regression, either binary or multinomial, because this model is doing something a bit different from those. But since it's still doing something similar to those models as well, the weights make sense. For example, the weight of the connection from Flexible to *tiao* is very large and positive (0.80571), the weights of the connections from this feature to *gen* and *zhi* are rather large and negative (-0.33986 and -0.46497, respectively), and the weights of the connections from the useless feature Round to all outputs are small (0.02451, 0.11062, 0.08679, respectively). Thus the feature Flexible facilitates *tiao* but inhibits *gen* and *zhi*, and Round does nothing, just as we saw before.

There are lots of things we can do with our trained network, including confirming that it gives good responses for the items it was trained on and testing what output it will give for new inputs, which, after all, is how you test whether a brain (human or otherwise) has "learned" something (for that, we can use the **compute()** function, which takes old or new inputs as arguments). We can also look at earlier stages in the learning by setting the **rep** argument to a lower value, so we can study how the network weights change over the course of learning; this kind of analysis makes connectionism an interesting tool for testing theories about language acquisition (see e.g., Elman et al., 1996).

Since connectionist models are mainly used to model learning, make predictions, and find patterns in data, they fall into the category of data exploration, rather than hypothesis testing. And that's the third "secret" reason why **multinom()** doesn't give *p* values.

So as I pointed out in the first chapter and again and again since then: everything in statistics is related. Who would have guessed that drawing lines through dots (regression) is actually related to melting ice (maximum entropy) and brains (neural networks)?

3.3 Varbrul

As I mentioned earlier, logistic regression is at the heart of the widely used Labovian sociolinguistic variable-rule analyzing program VARBRUL and its descendants. VARBRUL was invented in the early 1970s (Cedergren & Sankoff, 1974) for a specific linguistic theory that claimed that mental grammars contain variable rules (Labov, 1969). The earliest version of VARBRUL actually used probit analysis, since its creators hadn't yet heard of logistic regression, which had only been invented a few years before. Since then, the grammatical theory of variable rules has lost favor, even among Labovians, but logistic regression remains a crucial analysis tool for studying linguistic variability (see Mendoza-Denton et al., 2003, for some history and math).

There's a step-by-step guide to make R's **glm()** function act like a VARBRUL program in Johnson (2008, pp. 174-180). If you have no prior experience with VARBRUL, or don't need to communicate with people who only know VARBRUL, then you don't need this.

If you *do* have prior experience with VARBRUL or want to introduce your Labovian friends to R, you don't need Johnson (2008) either, since R has a package specifically designed to imitate VARBRUL, cleverly called **Rbrul** (Johnson, 2009: not the same Johnson!). When you run it, you get step-by-step prompts on what to do, so even R-fearing sociolinguists can use it. It also goes beyond VARBRUL in doing fancy stuff like mixed-effects logistic regression (which we'll learn about in the next chapter). Unlike most of our R packages, however, it's not part of the CRAN network for R packages, but you can read about it at http://danielezrajohnson.com/rbrul.html. You can install and run the latest version by entering the following into the R window:

```
source("http://www.danielezrajohnson.com/Rbrul.R ")
```

The following command will then run it in a "shiny" package (that's what it's called) that looks and feels more like a "modern" program, not the old-fashioned text-based R window:

```
rbrul()
```

4. Weirder types of regression

Logistic regression is by far the most commonly used type of non-linear modeling, but there are others. In this section we'll look at a few of them: Poisson regression (for count data), regression with ordinal variables (either as dependent or independent variables), generalized additive models (for dependent variables that are wiggly in complex ways), Zipf-based regression for modeling corpus frequencies, and finally a few more even weirder types.

4.1 Poisson regression

The **Poisson distribution** is named after the mathematician Siméon Denis Poisson (1781-1840), who was French, so the first syllable is pronounced /pwa/, not /poi/ (and his name comes from the French word for fish, not the English word for poison!). Like the binomial distribution, the Poisson distribution relates to categorical variables, but this time to counts, rather than to binary or multinomial categories.

Counts are discrete (you can have two tokens of a word in a corpus, but not 1.5 tokens) and can't go below zero (you can't have minus two word tokens). If your counts are large (e.g., vocabulary sizes across languages), the Poisson distribution will be basically normal, but if your data set is not huge, the Poisson distribution will be categorical and positively skewed, and linear models aren't ideal (though they're still usable, if you adjust for the skew).

One weird property of Poisson distributions is that the mean is identical to the variance, so it only has one parameter, called **lambda** (λ). Of course R has the usual set of functions for Poisson distributions, so we can get a feeling for it; see the plot in Figure 9.

```
set.seed(1) # So our results match
pois.sample = rpois(n = 100000, lambda=3)
mean(pois.sample)
```

```
[1] 2.99847
```

```
var(pois.sample) # Close enough for a random sample!
```

```
[1] 3.015338
```

```
x = 1:10
plot(x, dpois(x, lambda=3))
```

```
ppois(3, lambda=3)
```

```
[1] 0.6472319
```

```
ppois(3, lambda=3, lower.tail=F)
```

```
[1] 0.3527681
```

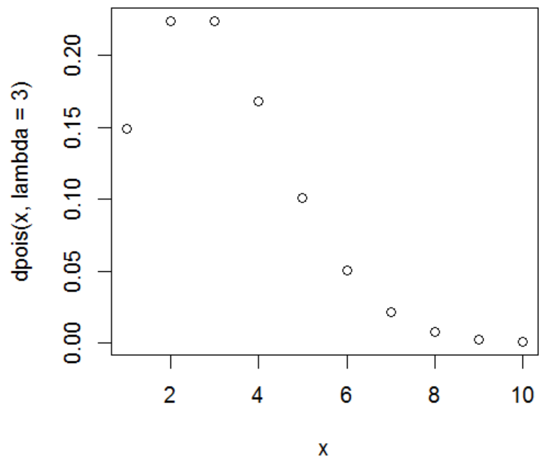

Figure 9. A Poisson distribution

You can think of Poisson regression as a yet another way to extend the chi-squared test, which also involves counts (in a contingency table). The canonical link function for Poisson regression is the logarithm, as for logistic regression. For this reason, both are also considered **loglinear models**, since they turn probabilities into a linear equation by using the logarithm.

This gives us a *fifth* way to analyze the Martian cheese "data", where 20 cheese nouns take the suffix *-qfx* and 10 don't. Note that the *p* value is far smaller than the exact *p* value given by the binomial test, which suggests that perhaps this sample is too small for this asymptotic test.

```
y = c(20,10) # Just the two counts
summary(glm(y~1, family=poisson)) # p < 2e-16
```

Now finally I can explain the Poisson analysis from Myers & Tsay (2015) that I cited at the beginning of this book. We wanted to know whether Southern Min speakers tend to reduplicate the discourse morpheme 著 "tioh8" /tiəʔ˥/ (which behaves similarly to Mandarin 對) an odd number of times, while also preferring fewer repetitions overall. So we analyzed the spoken corpus data in **DuiCounts.txt** using Poisson regression. The best-fitting model (according to analysis of deviance model comparisons using **anova(... test = "Chisq")**) proved to be a complex one involving interactions and polynomials (since the counts drop off nonlinearly):

```
dui = read.delim("DuiCounts.txt")
dui.poly.pois = glm(Count ~ I(NumSyl^2) * NumSyl * Oddness,
  family = "poisson", data = dui)
summary(dui.poly.pois) # Here are the coefficients
```

```
Coefficients:
                              Estimate   Std. Error   z value     Pr(>|z|)
(Intercept)                    3.77697    0.453464     8.329     < 2e-16 ***
I(NumSyl^2)                   -0.44533    0.088341    -5.041    4.63E-07 ***
NumSyl                        1.627676    0.371133     4.386    1.16E-05 ***
Oddness                       2.847068    0.453464     6.278    3.42E-10 ***
I(NumSyl^2):NumSyl            0.024603    0.006124     4.017    5.89E-05 ***
I(NumSyl^2):Oddness           0.242203    0.088341     2.742     0.00611 **
NumSyl:Oddness                -1.47633    0.371133    -3.978    6.95E-05 ***
I(NumSyl^2):NumSyl:Oddness      -0.012    0.006124     -1.96     0.05003 .
```

We can see what this model is doing if we plot its predictions against the real counts, as in Figure 10:

```
dui$Count.hat = predict(dui.poly.pois, type="response")
plot(dui$NumSyl, dui$Count, pch=19, # Black dots (observed)
  xlab="Number of syllables", ylab="Counts")
lines(dui$NumSyl, dui$Count, lwd=2) # Thick line (observed)
```

```
points(dui$NumSyl, dui$Count.hat, pch=0) # White squares (model)
lines(dui$NumSyl, dui$Count.hat, lwd=1) # Thin line (model)
par(lwd=1) # Thin line for legend box
legend("topright",
  lwd=c(2,1), # Thick vs. thin lines
  pch=c(19,0), # Black dots vs. white squares
  legend=c("Observed","Model"))
```

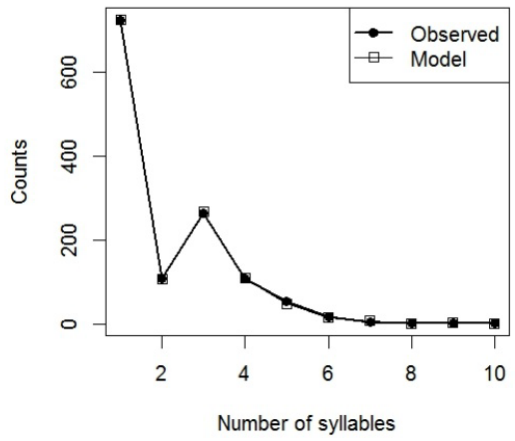

Figure 10. A perfect (too perfect?) Poisson model

As you can see, the model is a perfect fit to the real data. This may be seem good, but it's also kind of suspicious: the data are so few, and there are so many parameters, so this is very likely a case of **overfitting**; that is, we probably can't generalize from this data set. But at least we've shown that a model with oddness and number of syllables works.

Just for practice, let's check if the dispersion assumption is valid here (as with logistic regression, the assumed value is 1).

```
library(DHARMa) # Only need to do this once per session
testDispersion(dui.poly.pois, plot = F)
```

```
        DHARMa nonparametric dispersion test via sd of residuals fitted vs. simulated

data:   simulationOutput
dispersion = 0.053408, p-value = 0.008
alternative hypothesis: two.sided
```

Uh-oh. Indeed, it looks like our analysis was too good to be true: our model was so complex that its powers swamped the relatively small data set, resulting in a serious case of underdispersion. As noted earlier, overdispersion is more common in real life, since real data tends to be noisier than models expect.

For yet another (underdispersed) use (or misuse?) of Poisson regression, see the R code in the MiniCorp program (Myers, 2012b), which tests the reliability and ranking of phonological constraints in word lists (http://personal.ccu.edu.tw/~lngmyers/MiniCorp.htm). Several more linguistic applications, from somebody who knows a lot more about Poisson regression than I do, are reviewed in chapter 13 of Winter (2020). In particular, Winter suggests that if you do encounter overdispersion (the more likely scenario), you can switch over to something called **negative binomial regression**, which doesn't make the Poisson assumption that the mean and variance be identical (i.e., the dispersion assumption); it's available via the **glm.nb()** function in the **MASS** package (Venables & Ripley, 2002 *Modern applied statistics with S*).

4.2 Regression with ordinal variables

Besides the continuous-valued or nominal variables that we've been working with in this book, regression models can also handle **ordinal** variables. As you may recall from the very beginning of the book, these are variables that are composed of discrete categories that occur in a specific order. For example, lexical access by humans is affected not just by lexical frequency (based on a corpus), but also subjective familiarity (based on people's intuitions). The familiarity scores usually come from asking a bunch of people to rate each word on an ordinal scale, say 1 = least familiar to 7 = most familiar. We have no way to know if the "real" distance from familiarity = 1 to familiarity = 2 is the same as from familiarity = 2 to familiarity = 3. All we know is the order: 3 is higher than 2, which is higher than 1.

This is a such a subtle distinction that it's hard to tell if it really makes a practical difference. But if you want to be strict about it, it is indeed possible to code an ordinal variable to represent the order, without also implying equal distances between the levels.

To try this out, let's look at the data in **freqdur.txt** yet again, which you may remember has the results of a fake experiment collecting spoken durations in milliseconds, as well as lexical frequency counts, mean subjective familiarity scores, and mean subjective age of acquisition (AoA) scores for real English words. All of these values can be treated as continuous, but let's turn the two subjective scores into integers (from 1 to 6) to be closer reflections of their original ordinal nature (i.e., what each participant gave the experimenter, before the values were averaged together).

```
fd = read.delim("freqdur.txt")
fd$AoA.ord = floor(fd$AoA) # Convert to integers
fd$Fam.ord = floor(fd$Fam) # Convert to integers
fd$AoA.ord = as.factor(fd$AoA.ord) # Convert to factor
fd$Fam.ord = as.factor(fd$Fam.ord) # Convert to factor
```

Here are the original levels for these two factors:

```
levels(fd$AoA.ord)
[1]  "1" "2" "3" "4" "5" "6"
levels(fd$Fam.ord)
[1]  "1" "2" "3" "4" "5" "6"
```

One way to treat these as ordinal factors to use **Helmert coding** (named after German scholar Friedrich Robert Helmert, 1843-1917, who studied "geodesy" [大地測量學], and no, I'd never heard of it before either). This coding scheme is like effect or sum coding in using values that sum to zero, but it does it in a sequential way, from "first" to "last" level (according to the ordering you want).

```
fd$AoA.ord.h = fd$AoA.ord # Prepare for Helmert coding
fd$Fam.ord.h = fd$Fam.ord # Ditto
contrasts(fd$AoA.ord.h) = contr.helmert(levels(fd$AoA.ord.h))
contrasts(fd$Fam.ord.h) = contr.helmert(levels(fd$Fam.ord.h))
```

This is what Age of Acquisition looks like now (ditto for Familiarity):

```
contrasts(fd$AoA.ord.h)
  [,1] [,2] [,3] [,4] [,5]
1   -1   -1   -1   -1   -1
2    1   -1   -1   -1   -1
3    0    2   -1   -1   -1
4    0    0    3   -1   -1
5    0    0    0    4   -1
6    0    0    0    0    5
```

Since our dependent variable (Dur) is continuous, let's do an ordinary linear regression, but with these ordinal variables as the independent variables:

```
summary(lm(Dur ~ log(Freq) + AoA.ord.h + Fam.ord.h, data = fd))
```

Among regression report stuff, we get this long coefficients table:

```
             Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)   252.874    2.28286   110.77   <2e-16
log(Freq)    -0.85010    0.53879   -1.578   0.1148
AoA.ord.h1    1.18439    2.32400    0.510   0.6104
AoA.ord.h2    0.09957    0.91143    0.109   0.9130
AoA.ord.h3    0.42067    0.52258    0.805   0.4209
AoA.ord.h4    0.75505    0.42800    1.764   0.0779
AoA.ord.h5    0.71604    0.51511    1.390   0.1647
Fam.ord.h1   -5.84849    5.34241   -1.095   0.2738
Fam.ord.h2    -1.55746     1.93729    -0.804    0.4215
Fam.ord.h3    -0.77988     1.06387    -0.733    0.4636
Fam.ord.h4    -0.32551     0.71897    -0.453    0.6508
Fam.ord.h5    -0.62202     0.69976    -0.889    0.3742
```

This is testing the mean of all the levels up to the indicated level against the next level in the order. For example, the marginally significant effect for AoA.ord.h4 (p = .0779) tests the mean of levels 1-4 against level 5.

Another totally different way to treat a factor as ordinal is to use **polynomial** (多項式) coding, which links each level to each part of a polynomial function ($y = b_0 + b_1x + b_2x^2 + b_3x^3 + ...$), which allows each level to act as if it's "beyond" all of the previous levels. Why? Because the $y = b_1x$ part is linear, the $y = b_2x^2$ part is curved a bit, and the $y = b_3x^3$ part is even wigglier, and so on. By adding this wiggliness, the specific distances between points along the independent variable become less important, and their overall order has a greater impact.

You can convert a factor to polynomial coding in three different ways, but they all have the same effect:

```
fd$AoA.ord.p = fd$AoA.ord # Prepare for polynomial coding
fd$Fam.ord.p = fd$Fam.ord # Ditto
# Method 1
fd$AoA.ord.p = factor(fd$AoA.ord, ordered = T)
fd$Fam.ord.p = factor(fd$Fam.ord, ordered = T)
# Method 2
fd$AoA.ord.p = ordered(fd$AoA.ord)
fd$Fam.ord.p = ordered(fd$Fam.ord)
# Method 3
contrasts(fd$AoA.ord.p) = contr.poly(levels(fd$AoA.ord.p))
contrasts(fd$Fam.ord.p) = contr.poly(levels(fd$Fam.ord.p))
```

No matter how you do it, the contrasts end up looking like this, again illustrating with Age of Acquisition, where "L" = "linear", "Q" = "quadratic" [二次函數], "C" = "cubic" [三次方程], and then they run out of technical terms and just show the powers directly as they should have done in the first place.

```
contrasts(fd$AoA.ord.h)
             .L          .Q          .C          ^4           ^5
[1,] -0.5976143   0.5455447   -0.372678   0.1889822  -0.06299408
[2,] -0.3585686  -0.1091089   0.5217492  -0.5669467   0.31497039
[3,] -0.1195229  -0.4364358   0.2981424   0.3779645  -0.62994079
[4,]  0.1195229  -0.4364358  -0.2981424   0.3779645   0.62994079
[5,]  0.3585686  -0.1091089  -0.5217492  -0.5669467  -0.31497039
[6,]  0.5976143   0.5455447    0.372678   0.1889822   0.06299408
```

Now let’s run our ordinary linear regression on Dur using these things:

summary(lm(Dur ~ log(Freq) + AoA.ord.p + Fam.ord.p, data = fd))

```
              Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)    252.874      2.2829   110.77   <2e-16
log(Freq)      -0.8501      0.5388   -1.578    0.115
AoA.ord.p.L     5.3303      3.5689    1.494    0.135
AoA.ord.p.Q     0.8873      2.9796    0.298    0.766
AoA.ord.p.C     0.4523      2.2154    0.204    0.838
AoA.ord.p^4    -1.4912      1.6119   -0.925    0.355
AoA.ord.p^5     0.3802      1.2513    0.304    0.761
Fam.ord.p.L    -6.6420      7.2869   -0.912    0.362
Fam.ord.p.Q     4.8530      5.9392    0.817    0.414
Fam.ord.p.C    -5.5443      4.3310   -1.280    0.201
Fam.ord.p^4     1.9267      2.8250    0.682    0.495
Fam.ord.p^5    -1.3674      1.7611   -0.776    0.438
```

Note that the *p* values are totally different from what we got with Helmert coding, since here “ordinal” has a totally different meaning, namely, the higher the power (from L to Q to C and beyond), the “wigglier” the effect is. For a real-life linguistic application of this this kind of analysis (in logistic regression, in fact), see Johnson (2008, pp.170-174).

What if the dependent variable is itself ordinal? For example, what if we want to analyze whether lexical frequency helps predict subjective familiarity in one speaker, where familiarity is on that ordinal scale? We could pretend that this is a continuous variable and do linear regression (that’s what most people do), but what if we want to be strict, and treat it as an ordinal variable?

Well, we can use something called **ordinal logistic regression**. This is a generalization of logistic regression, where instead of comparing binary responses of 1 vs. 0, we compare ordinal responses of 1 vs. higher, 2 vs. higher, and so on. A function for computing it is in the **MASS** package (Venables & Ripley, 2002):

library(MASS) # As usual, you have to install it first

So let’s take the data in the data frame **fd** again, but this time try to predict the ordinal variable **Fam.ord** from **logFreq**, which we create using **log(Freq)**:

fd$logFreq = log(fd$Freq)

The **MASS** function we want is called **polr()**, which stands for **proportional odds logistic regression**:

```
polr.all = polr(Fam.ord ~ logFreq, data = fd)
summary(polr.all)
```

```
Re-fitting to get Hessian

Call:
polr(formula = Fam.ord ~ logFreq, data = fd)

Coefficients:
            Value   Std. Error    t value
 logFreq  1.222549    0.044242   27.63346

Intercepts:
                Value    Std. Error   t value
         1|2   -3.7506      0.4116    -9.1123
         2|3   -1.1413      0.1384    -8.2467
         3|4    0.8544      0.1057     8.0850
         4|5    3.4360      0.1381    24.8753
         5|6    7.5443      0.2280    33.0847

Residual Deviance: 3281.896
AIC: 3293.896
```

Note that while it runs separate analyses for the overall model (in the “Coefficients” table), as well as for each ordered level (in the “Intercepts” table), it only gives the test statistic for each: just the *t* values, not the *p* values. This is because the Wald test is particularly unreliable for ordinal logistic regression, and it’s not even clear what the *df* should be. If the sample is very large (as it is here: **nrow(fd) == 1689**), we can treat the *t* values as *z* scores (remember that *t* distributions turn into the normal distribution as the sample size grows). In this case, all of our $|t| > 1.96$, so they all have two-tailed $p < .05$.

If you want to avoid all Wald test problems, we can use a likelihood ratio test comparing our full model with one that is missing log frequency, which reports the chi-squared-distributed likelihood ratio statistic (as “LR stat.”); try it!

```
polr.int = polr(Fam.ord ~ 1, data = fd) # Remove Freq (just intercept remains)
anova(polr.int, polr.all) # df = 1, LR stat. = 1085.159, p = 0 # Not literally zero!!
```

So for this analysis we could report the following analysis of the effect of the continuous variable of log frequency on the ordinal variable familiarity: $\chi^2(1) = 1085.16, p < .001$.

4.3 Taking Zipf into account

One kind of categorical data is particularly troublesome in linguistics: word frequencies, or more generally, frequencies of lexical classes, like the numbers of monosyllabic words, or irregular verbs, or Chinese nouns that take the *tiao* classifier, and so on. These are problematic because word frequencies form an extremely skewed distribution: most words in a corpus are rare. This is related to Zipf's law, which we mentioned early in this book: if you rank words from most to least frequent, word #2 will be about 1/2 as frequent as word #1, and so on. By the time you get to word #100, the frequencies are already going to be extremely tiny.

This fact has caused us problems already; it's why we log-norm lexical frequencies, even though it doesn't totally eliminate the skew. But as Baayen (2001) points out, it causes an even more fundamental problem: it kills the dream that we can make reliable claims based on ordinary sampling logic. That is, whereas a *t* test or even a logistic regression can give useful results even for samples with just a hundred or so data points, this would far too small a corpus to make word frequency claims, since most words are too rare. It's even worse than this, though, since Zipf's law says that even if you do have a gigantic corpus, most of the words in it will *still* be rare! There's just no way to get a large enough corpus to quantify everything about it.

Fortunately, we can get a bit closer if we use regression models that explicitly take Zipf's law into account. R has a package for just this purpose: **zipfR** (Evert & Baroni, 2007), which is designed to create yet another kind of loglinear model, this time called **Large-Number-of-Rare-Events** (**LNRE**) modeling.

To give you a quick feeling for how it works, let's apply it to the tiny corpus we played with way back in chapter 2: **Jabberwocky_OnlyWords.txt**. First we'll load it in as a string vector using **readLines()**, just as we did before (**read.table()** would be more awkward here):

```
jabberwocky = readLines("Jabberwocky_OnlyWords.txt")
head(jabberwocky) # Remember this?
```

```
[1] "jabberwocky" "twas"        "brillig"     "and"        "the"        "slithy"
```

Remember that we can compute the frequency table using **table()**. This gives us a sense of Zipf's law, since most words have a frequency of just 1:

```
table(jabberwocky)
```

```
jabberwocky
          all          and         arms           as       awhile
            2           14            1            2            1
         back bandersnatch      beamish       beware         bird
            1            1            1            2            1
...
```

We can get an even clearer sense of the abstract power of Zipf's law if we apply **table()** twice, to show us the frequencies of every observed frequency. This is called a **frequency spectrum**, showing the **token frequencies** (top row) and the **type frequencies** (bottom row).

jabberwocky.tabtab = table(table(jabberwocky))
jabberwocky.tabtab

```
  1   2   3   6   7  14  19
 56  28   3   1   1   1   1
```

This reveals that there are 56 word types with a token frequency of 1, and 28 words with a frequency of 2: exactly half, just as Zipf's law predicts! By the same law, there should be 56/3 = 18.66667 words with a frequency of 3, but in reality there are only three such words, and all of the remaining frequencies only have at most one word. So in this tiny sample, Zipf's law works for the highest-frequency words, but fails for rarer words.

Now let's see what the **zipfR** package can do with this tiny corpus. First we start it up (after installing it):

library(zipfR)

Then we use **zipfR**'s **spc()** function to convert our hand-made frequency spectrum into a proper spectrum object, which contains the same information, but coded in a way that other **zipfR** functions can use. The table above actually uses character strings for the token frequencies (as shown by the **names()** function), so I use the **as.numeric()** function to convert them to numbers. The **spc()** function's **m** argument is for the frequency classes (here, the token frequencies), and the **Vm** argument is for the sizes of these classes (here, the type frequencies):

jabberwocky.spc = spc(m=as.numeric(names(jabberwocky.tabtab)),
  Vm = as.numeric(jabberwocky.tabtab))

Here's what we created:

jabberwocky.spc

```
     m  Vm
1    1  56
2    2  28
3    3   3
4    6   1
5    7   1
6   14   1
7   19   1

     N   V
   167  91
```

The top part shows the same numbers we got with **table(table())**, and the bottom sums them up: **N** is the number of tokens (i.e., **length(jabberwocky)**), and **V** is the vocabulary size (i.e., **length(unique(jabberwocky))**). Now we can run a Zipf-based regression model using the **lnre()** function. This function can run several types of related models, but let's just use the simplest one: the **Zipf-Mandelbrot** model, named after Zipf and the Polish/French/American mathematician Benoit Mandelbrot (1924-2010). Math nerds may know him from the amazing (and beautiful) Mandelbrot set (https://en.wikipedia.org/wiki/Mandelbrot_set). To use this model, we set the first argument to "zm":

```
jabberwocky.zm = lnre("zm",jabberwocky.spc)
```

To see what's inside it, we just enter its name (we don't need **summary()**):

```
jabberwocky.zm
```

```
Zipf-Mandelbrot LNRE model.
Parameters:
 Shape:           alpha  =  0.0000001075863
 Upper cutoff:        B  =  0.01650917
 [ Normalization:     C  =  60.57236          ]
Population size: S = Inf
Sampling method: Poisson, with exact calculations.

Parameters estimated from sample of size N = 167:
                V      V1     V2     V3     V4     V5
  Observed:  91.00  56.00  28.00    3.0   0.00   0.00 ...
  Expected:  97.47  56.73  23.06   10.5   4.52   1.77 ...

Goodness-of-fit (multivariate chi-squared test):
          X2   df             p
    13.49343    3 0.003682427
```

Look at the parameters table. The row of observed frequencies are the same that we have in **jabberwocky.tabtab** or **jabberwocky.spc**, where V is the total vocabulary size, and V1, V2, and so on are the numbers of words with a frequency of 1, 2, and so on. The row of expected frequencies are the values generated by the Zipf-Mandelbrot model. This model does a not-bad job with the two highest frequencies (V1 and V2), but it does terribly starting with V3. This makes perfect sense, since as we saw ourselves, the first highest frequencies follow Zipf's law very well, but after that it fails, just like this model.

This lack of fit is quantified in the goodness-of-fit chi-squared test at the end of the report: $\chi2(3) = 13.49$, $p = .003$. This high chi-squared value is bad news: the two rows of values (observed and expected) are quite unlikely to differ just by chance, so the model is a bad fit. Of course this is partly because our corpus is so tiny, but in my experience working with larger corpora, the fit always fails at some point. So even the **zipfR** package can't completely eliminate the bothersome implication of Zipf's law.

But assuming the model isn't a total failure, we can still use this model for useful things. For example, imagine you are transcribing the speech of a child acquiring English as a first language, and this "Jabberwocky" poem was the only data you have from this child (yes, a very weird child). So this kid produced only 167 word tokens (in your records anyway), during which there were 91 different word types. What does this predict for larger corpora, say one with 10,000 word tokens? How many word types would we expect? In other words, how large is this kid's actual vocabulary, way beyond just this one poem?

To find out, we can use the **lnre.vgc()** function, which uses a fitted LNRE model to estimate a **vocabulary growth curve**, which reflects the increase in word types (V) as we increase the number of word tokens (N) in our sample (corpus). We'll compute it from our fitted Zipf-Mandelbrot model, starting from the first word token up to 10,000 word tokens (way beyond what we've actually recorded), probing the growth curve at 20 points along the way:

jabberwocky.vgc = lnre.vgc(jabberwocky.zm, seq(1, 10000, length=20))

To see what we've modeled, let's just plot it, as in Figure 11, which shows the increase in vocabulary size as the projected corpus size (N) increases (E[V(N)] = Expected Vocabulary size as a function of the Number of tokens).

plot(jabberwocky.vgc)

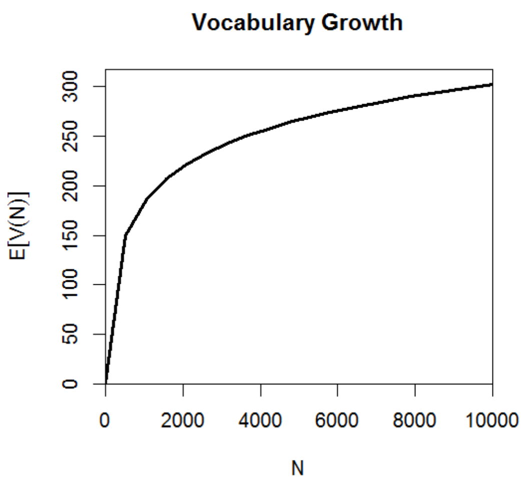

Figure 11. Predicted growth curve for "Jabberwocky"

The model thus predicts that the growth curve gets less steep. That is, if "Jabberwocky" is good representative of this kid's language production, and if this model is a good model of this tiny corpus (both are big "if"s), then we can expect that if we recorded 10,000 words from this kid, there would only be about 300 different words in total. If the kid is young enough, maybe 10,000 is even a good guess for the total number of word tokens ever produced; in that case, we can guess that the kid's actual vocabulary size is 300.

This kind of logic (albeit using much larger corpora) can also be applied to testing the productivity of various morphological processes; see Evert and Baroni (2007) and Baayen (2008) for examples. In particular, they show that affixes that seem relatively rare in the corpus may actually be more productive than more familiar affixes, simply because they appear in more rare words (and thus are more likely to be applied to create new words in natural language production).

4.4 Generalized additive models

Given the variety of models we've seen in this chapter, you might think that generalized linear models are so generalized that there's no way they could be generalized even more, but they can! All of the above models involve well-established functions and distributions, but with the increasing power of computers, we don't even need to be limited by these mathematical constraints anymore. Using **generalized additive modeling** (**GAM**) we can analyze the relationship between dependent and independent variables even if the shape of this relationship is arbitrarily wiggly, rather than a single neat function. As in all types of models, the individual independent variables are still added up (hence the name), but each one of them is now subject to a different, arbitrary function that can make the overall model as non-linear as your data demand. Interactions can also be analyzed in such models, but this gets complicated, so you'll have to read about GAM yourself, in tutorials like Clark (2013) (relatively easy), Tremblay and Newman (2014) (somewhat harder), and Wood (2006) and Baayen et al. (2017) (hardest). For linguists maybe the best not-hard introduction is Winter and Wieling (2016), who apply GAM (and other models) to language change.

Baayen et al. (2017) motivate the increasing use of GAMs in psycholinguistics and related fields by referring to Plato's allegory of the cave, where Plato described objects in the observable world as being mere shadows on a cave wall projected from "ideal" objects that we cannot see. In this case, the cave is a cage of functions that statisticians have tended to use mainly to make the calculations simpler. But generalized additive modeling, it is claimed, can let research escape this cage (or cave) and see the real world (it is claimed). They apply this technique to modeling the complex up-and-down changes in responses across the many trials of a long psycholinguistic experiment, which can obscure the experimental contrasts that you care about. Another linguistic example of this new technique is given by Tremblay and Newman (2014), who use it to bring some order to the complex patterns in event related potentials (ERPs) studied in brainwave research.

The most widely used GAM package for R is **mgcv** (Woods, 2006), which stands for Mixed GAM Computation Vehicle (sorry, that's what it stands for). Let's load it up:

```
library(mgcv) # You have to install it first, remember?
```

Let's try it out on the data set in **RTacc.txt**, which shows the accuracy rates and reaction times for a set of items in some sort of experiment:

```
rtacc = read.delim("RTacc.txt")
```

As Figure 12 shows, the relationship between the two variables is not linear, with accuracy rates starting out flat for the faster responses, and then falling in a more linear fashion. Presumably this shape arises because responses are both more accurate and faster (lower RT values) for easier items, but the accuracy rate has hit a ceiling for the easiest items (it's far from the floor of 0 accuracy, though).

```
plot(Acc ~ RT, data = rtacc)
```

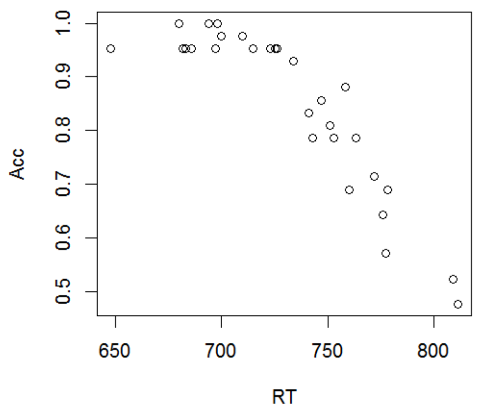

Figure 12. A nonlinear scatter plot

We can replot this with a smoothed loess line to make the overall trend easier to see, as shown in Figure 13, but this is not a regression model that can be used to test hypotheses; it's just a form of descriptive data exploration.

```
scatter.smooth(rtacc$RT,rtacc$Acc) # Smoooooth!
```

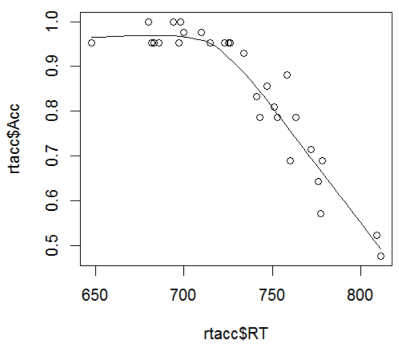

Figure 13. A nonlinear scatter plot with a loess line

With GAM, however, we can include a smoothing function inside our equation. These functions are called **splines** (樣條函數), which act like the splines (雲規) used by engineers to fit weird shapes by "pushing" and "pulling" at various points. For example, we can use **cubic splines**, which are a bunch of little wiggly cubic equations ($y = b_0 + b_1x_1 + b_2x_2^2 + b_3x_3^3$) overlapped on each other, or use **thin plate splines**, which overlap wiggly lines created by analogy with a bending thin metal plate (which is what the physical-world engineering spline is). The latter type of spline is the default for the **gam()** function, and that's good enough for our demonstration.

To make this work, all we have to do is put the independent variable that we want to smooth inside the **s()** function ("s" for "smooth"), and the **gam()** function will do the rest:

rtacc.gam = gam(Acc ~ s(RT), data = rtacc)

Before looking at the **summary()** of this GAM object, let's plot it, as in Figure 14.

plot(rtacc.gam)

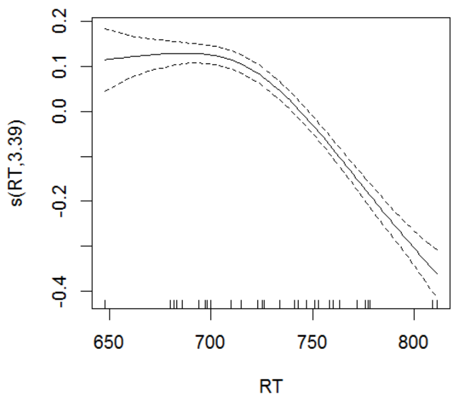

Figure 14. A smoothed fit to our nonlinear Acc ~ RT data

It’s shaped a lot like our loess line, which means it’s capturing the essence of our data, instead of trying to force it to match some specific function. It also shows the 95% confidence band, in case we need that, and the distribution of the *x* values (the little marks at the bottom).

Now let’s see what the statistical analysis looks like for this model:

summary(rtacc.gam)

```
Family: gaussian
Link function: identity

Formula:
Acc ~ s(RT)

Parametric coefficients:
              Estimate  Std. Error  t value  Pr(>|t|)
 (Intercept)  0.848267   0.007574       112  <2e-16 ***
---
Signif. codes:   0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
          edf  Ref.df       F  p-value
 s(RT)  3.391   4.188  84.49  <2e-16 ***
---
Signif. codes:   0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =   0.925     Deviance explained = 93.3%
GCV = 0.002016   Scale est. = 0.0017209    n = 30
```

Explaining all of this would take a lot more space than we have in this already very long chapter, but at least we can see that the model is pretty good: the adjusted R^2 is .925 (this measure is valid here because it’s partly a linear model), the deviance explained is 93.3% (this measure is valid here because it’s partly *not* a linear model), and the smoothed RT predictor is highly significant ($p < .0001$). Of course, as the report says, this *p* value only represents the approximate significance, and in fact “edf” stands for “estimated *df*”, since one cost of throwing out all of our usual functions and distributions and using splines instead is that we can only get approximate values. We don’t even get a coefficient for RT, as we would for other types of regression, since the line, being composed of overlapping little bits, can be wiggly to an undefined extent, and so one coefficient, or even the many coefficients for polynomial lines, can’t define it. But that’s OK: we can see from the plot what the overall shape looks like and how its various portions are shaped (flat on the left and falling on the right).

One thing you might have noticed is that the *y* axis in the GAM plot isn’t the same as in the original scatter plot: in the raw data, it’s Acc (our dependent variable), ranging from around 0.5 to around 1.0, but in the GAM plot it’s called “s(RT, 3.39)” and it ranges from around -0.4

to around 0.2. That's because the plot here is just showing the smoothed part of the model, not any other stuff, such as the intercept.

The simplest way to get the model's predicted trend line in plot-friendly format is to use the **predict()** function, telling it to predict within the original response scale (Acc). To get a nice curve for the line, we'll generate these predictions for values in the RT range that are closer together than the actual RT values:

```
rtacc.fit = data.frame(RT = seq(min(rtacc$RT),max(rtacc$RT),1)) # 648, 649, 650, ...
rtacc.fit$Acc = predict(rtacc.gam, newdata=rtacc.fit, type = "response")
```

Now we can just add this line to a regular scatterplot, giving us Figure 15:

```
plot(Acc ~ RT, data = rtacc) # Regular scatter plot
lines(Acc ~ RT, data = rtacc.fit) # The GAM trend line
```

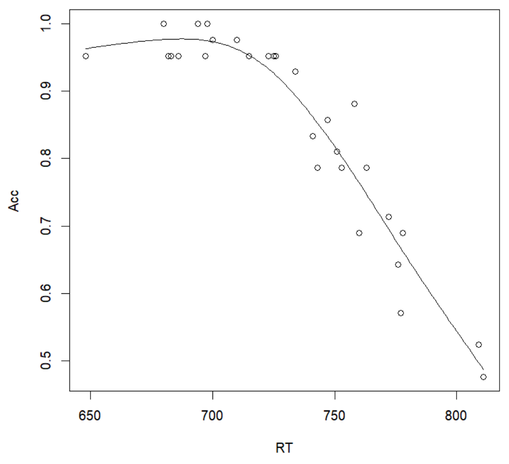

Figure 15. Quick'n'dirty GAM plot with original *y*-axis scale, plus original data points

If we want to show that 95% confidence band, we can derive them ourselves using the predicted standard errors (SE), and then using the critical value of 1.96 for the normal distribution (in the hopes that everything is roughly normal with large enough sample sizes). As shown by the match between Figure 16, generated below, and Figure 14, I guess this is how **plot.gam()** works too:

```
rtacc.fit$SE = predict(rtacc.gam, newdata=rtacc.fit, type = "response", se.fit=T)$se.fit
library(ggplot2) # Cuz it's got a built-in confidence band thingie
ggplot(data = rtacc.fit, mapping = aes(x = RT, y = Acc)) +
  geom_line() + geom_ribbon(mapping = aes(ymin = Acc-SE*1.96, ymax= Acc+SE*1.96),
  alpha = .2) + labs(y = "Accuracy")
```

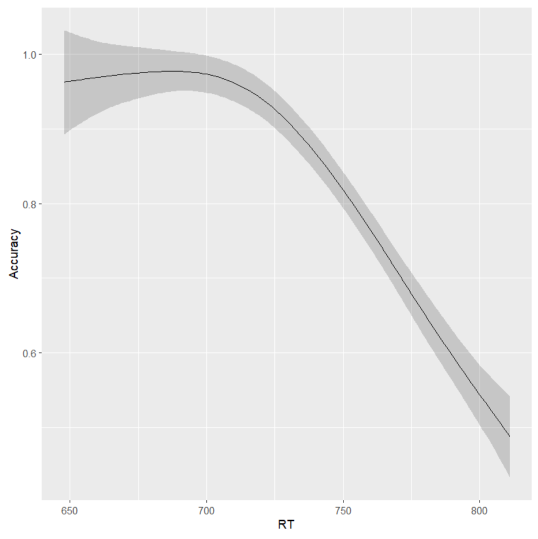

Figure 16. Quick'n'dirty GAM plot with original *y*-axis scale, plus 95% confidence band.

Plotting all of this together (original *y*-axis, original data points, and 95% confidence band) will be left to the interested reader.

Before finishing this section, I should say that even though GAM is very useful (and becoming ever more popular), you should be aware that its ability to model any wiggly curve comes with a big limitation: it can't **extrapolate** (外推), that is, predict values outside the range of the actual data (see the problems that arise when you try to extrapolate using such models anyway: https://fromthebottomoftheheap.net/2020/06/03/extrapolating-with-gams/). The basic problem should be obvious: if the wiggliness has no formula, there's no way to know what wiggles you'll get before or after what you see in your plot. This makes GAM quite different from linear regression, where we can expect the line to stretch out the same way beyond our plot. Nevertheless, you can still **interpolate** (插值), that is, make predictions for values that are missing from your data but are still within your data's range.

4.5 Even more nonlinear models

Let me end with quick mentions of three other nonlinear tools in R that you might want to try some day. The first is the base package function **nls()**, which stands for **nonlinear least squares**. As the name suggests, it generalizes the metric used by linear regression to find the best-fitting straight line, but the clever thing is that it does so for any crazy function you want to use to model your dependent variable in terms of your independent ones, even if your function doesn't use a built-in structure that's linear, polynomial, exponential, logistic, or something else with a special-purpose R function. It does this by looping through possible coefficient values until changing them doesn't make the fit any better. The following fake R code illustrates how you might use it to predict some variable **y** from the variable **x** in terms of some function you invented called **myfunction()**, which also takes the model constants **a** and

b (like the coefficients in an ordinary model), where you think **a** = 0, **b** = 1 are reasonable starting point (maybe **a** is added and **b** is multiplied in your function):

```
mymodel.nls = nls(y~myfunction(x,a,b), start = list(a = 1, b = 1), data=mydata)
```

The second type of model is called **gamma regression**, which uses yet another family of distributions (we'll see it again in the chapter on Bayesian statistics) that has the useful property of being as skewed or symmetrical as you like, depending on its parameters. Thus it can be used to model not-completely-normal phenomena like reaction times; Lo & Andrews (2015) recommend this approach since it doesn't distort the data the way the usual lognorming method does. This fake R code gives you some idea of how it might be used:

```
myrtmodel.gamma = glm(RT ~ Something, family=Gamma(link=identity), data=mydata)
```

The third (and final!) fancy nonlinear model that I'll mention is called **quantile regression**, a generalization of generalized additive modeling. Ordinary linear regression, similar to other parametric tests like the *t* test and ANOVA, is focused on the mean and variance, but there are many other interesting aspects of a distribution that we might want to test, especially perhaps if it's not normally shaped, such as the median, the minimum, or the maximum. Or maybe our data looks perfect for a linear model, except that it seriously violates the homoscedasticity assumption, with widely varying variance in the dependent variable as a function of the independent variable. A specific linguistic application might be to to study a child's median scores on some difficult task over many weeks of testing. Well, with quantile regression, as implemented in the **qgam** package (Fasiolo et al., 2021), you can do just that:

```
qgam(Score ~ Week, qu = 0.5, data = mydata) # qu = quantile as a value from 0 to 1
qgam(Score ~ Week, qu = 0.75, data = mydata) # Top quartile of scores
```

The creators of the package have posted a rich (but technical) tutorial at this website: https://cran.r-project.org/web/packages/qgam/vignettes/qgam.html. Note that the full name of the package has the word "Bayesian" in it (see final textbook chapter for more on this).

5. Conclusions

Well, this was another long chapter. That's only to be expected when your topic is a generalization of the already long previous chapter. Generalized linear models use various tricks to extend the power of regression to non-linear situations. By far the most common type is (binary) logistic regression, which analyzes binary dependent variables, which are widely used in linguistic research, from accuracy in experiments to the presence or absence of linguistic markers in sociolinguistics. The trick used by logistic regression is the logit, or log odds, which both expands the *y*-axis range from 0-to-1 to minus-infinity-to-plus-infinity, and captures the insight that adding independent variables is like multiplying probabilities. Despite the novelty, much of what we learned about (linear) multiple regression still applies to logistic regression, just in modified form: we can have more than one predictor, we have to watch out for collinear predictors, we can code the factors in various ways, we can test for interactions, we can test both numerical and categorical predictors, we should check to see if our model fits the data well (using log likelihood and residual deviance), we should plot our model in intuitive and useful ways (perhaps with log odds on the *y*-axis), we can compare models using likelihood ratio tests (computed with analysis of deviance instead of analysis of variance), we can run repeated-measures analyses with hand-written R code, and we can estimate effect sizes by standardizing the coefficients (except that it's harder to take the dependent variable into account). There are also new issues; in particular, when working with logistic regression you also have to watch out for problems caused by the algorithm itself (it may crash or give impossible values). We also saw that logistic regression can be generalized beyond binary variables, with multinomial logistic regression showing up (in disguise) in many places, from computer models of the brain to variable rule analysis. Yet logistic regression is only one member of the family of generalized linear models, which also includes Poisson regression, ordinal logistic regression, Large Numbers of Rare Events modeling and a hot new method called generalized additive modeling. All of these techniques rely on powerful computers to build and test models by trying this and that until the models work, instead of plugging numbers into fixed equations as in traditional linguistic methods like *t* tests, chi-squared tests, ANOVA, and linear regression. The same increase in computer power has also made the rest of this book's statistical techniques possible, as we will see.

References

Agresti, A. (2007). *An introduction to categorical data analysis*, second edition. Wiley-Interscience.

Albert, A., & Anderson, J. A. (1984). On the existence of maximum likelihood estimates in logistic regression models. *Biometrika, 71* (1), 1-10.

Allerup, P., & Elbro, C. (1998). Comparing differences in accuracy across conditions or individuals: An argument for the use of log odds. *The Quarterly Journal of Experimental Psychology, 51A* (2), 409-424.

Baayen, R. H. (2001). *Word frequency distributions*. Springer.

Baayen, H., Vasishth, S., Kliegl, R., & Bates, D. (2017). The cave of shadows: Addressing the human factor with generalized additive mixed models. *Journal of Memory and Language, 94*, 206-234..

Berger, A., Della Pietra, S. A., & Della Pietra, V. J. (1996). A maximum entropy approach to natural language processing. *Computational Linguistics, 18* (4), 381-392.

Cedergren, H., & Sankoff, D. (1974). Variable rules: Performance as a statistical reflection of competence. *Language, 50*, 333-355.

Clark, M. (2013). Generalized additive models: Getting started with additive models in R. Center for Social Research, University of North Dame, Notre Dame, IN, USA, 13.

Elman, J. L., Bates, E. A., Johnson, M. H., Karmiloff-Smith, A., Parisi, D., & Plunkett, K. (1996). *Rethinking innateness: A connectionist perspective on development*. MIT Press.

Evert, S., & Baroni, Marco (2007). zipfR: Word frequency distributions in R. *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, Posters and Demonstrations Sessions* (pp. 29-32). Prague, Czech Republic.

Fasiolo, M., Wood, S. N., Zaffran, M., Nedellec, R., & Goude, Y. (2021). qgam: Bayesian nonparametric quantile regression modeling in R. *Journal of Statistical Software, 100*(9), 1-31.

Friedman, J., Hastie, T., Simon, N., & Tibshirani, R. (2017). glmnet. R package.

Hartig, F. (2022). DHARMa: Residual Diagnostics for Hierarchical (Multi-Level / Mixed) Regression Models. R package.

Hayes, B., & Wilson, C. (2008). A maximum entropy model of phonotactics and phonotactic learning. *Linguistic Inquiry, 39*, 379-440.

Heinze, G., Ploner, M., Dunkler, D., & Southworth, H. (2017). logistf. R package.

Johnson, K. (2008). *Quantitative methods in linguistics*. Wiley.

Johnson, D. E. (2009). Getting off the GoldVarb standard: Introducing Rbrul for mixed-effects variable rule analysis. *Language and Linguistics Compass, 3*(1), 359-383. Software available at http://www.danielezrajohnson.com/rbrul.html.

Labov, W. (1969). Contraction, deletion, and inherent variability of the English copula. *Language, 45*, 715-762.

Lo, S., & Andrews, S. (2015). To transform or not to transform: Using generalized linear mixed models to analyse reaction time data. *Frontiers in Psychology, 6*.

Mehta, C. R., & Patel, N. R. (1995). Exact logistic regression: Theory and examples. *Statistics in Medicine, 14*(19), 2143-2160.

Menard, S. (2000). Coefficients of determination for multiple logistic regression analysis. *The American Statistician, 54*(1), 17-24.

Menard, S. (2004). Six approaches to calculating standardized logistic regression coefficients. *The American Statistician, 58* (3), 218-223.

Mendoza-Denton, N., Hay, J., & Jannedy, S. (2003). Probabilistic sociolinguistics: Beyond variable rules. In R. Bod, J. Hay, & S. Jannedy (Eds.) *Probabilistic linguistics* (pp. 97-138). Cambridge, MA: MIT Press.

Myers, J. (2009). The design and analysis of small-scale syntactic judgment experiments. *Lingua, 119*, 425-444.

Myers, J. (2012a). Testing adjunct and conjunct island constraints in Chinese. *Language and Linguistics, 13* (3), 437-470.

Myers, J. (2012b). Testing phonological grammars with lexical data. In J. Myers (Ed.) *In search of grammar: Empirical methods in linguistics* (pp. 141-176). Language and Linguistics Monograph Series 48. Taipei, Taiwan: Language and Linguistics.

Myers, J. (2016). Knowing Chinese character grammar. *Cognition, 147*, 127-132.

Myers, J., & Tsay, J. (2015). Trochaic feet in spontaneous spoken Southern Min. In Hongyin Tao, Yu-Hui Lee, Danjie Su, Keiko Tsurumi, Wei Wang, & Ying Yang (Eds.), *Proceedings of the 27th North American Conference on Chinese Linguistics, Vol. 2*, 368-387. Los, Angeles: UCLA.

Pampel, F. C. (2000). *Logistic regression: A primer*. Sage.

Raaijmakers, J. G. W., Schrijnemakers, J. M. C., & Gremmen, F. (1999). How to deal with “the language-as-fixed-effect fallacy”: Common misconceptions and alternative solutions. *Journal of Memory and Language, 41*, 416-426.

Speelman, D. (2014). Logistic regression: A confirmatory technique for comparisons in corpus linguistics. In D. Glynn & J. A. Robinson (Eds.) *Corpus methods for semantics: Quantitative studies in polysemy and synonymy* (pp. 487-534). Amsterdam: John Benjamins.

Tremblay, A., & Newman, A. J. (2015). Modeling nonlinear relationships in ERP data using mixed-effects regression with R examples. *Psychophysiology, 52*(1), 124-139.

Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with S* (4th edition). Springer.

Warnes, G. R., Bolker, B., & Lumley, T. (2014). gtools. R package available at http://cran.r-project.org/web/packages/gtools/index.html

Warton, D. I., & Hui, F. K. (2011). The arcsine is asinine: The analysis of proportions in ecology. *Ecology, 92*(1), 3-10.

Winter, B. (2020). *Statistics for linguists: An introduction using R*. Routledge.

Winter, B., & Wieling, M. (2016). How to analyze linguistic change using mixed models, Growth Curve Analysis and Generalized Additive Modeling. *Journal of Language Evolution, 1*(1), 7-18.

Wood, S. (2006). *Generalized additive models: An introduction with R*. CRC Press.

Xu, Y., Gandour, J. T., & Francis, A. L. (2006). Effects of language experience and stimulus complexity on the categorical perception of pitch direction. *Journal of the Acoustical Society of America, 120*(2), 1063-1074.

Zamar, D., McNeney, B., & Graham, J. (2007). elrm: Software implementing exact-like inference for logistic regression models. *Journal of Statistical Software, 21*(3).